N° d'ordre: 2974

THÈSE

Présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Geoffroy VALLÉE

Équipe d'accueil : IRISA/PARIS
École Doctorale : Mathématiques, Informatique, Signal et Électronique et Télécommunications
Composante universitaire : IFSIC

Titre de la thèse :
Conception d'un ordonnanceur de processus adaptable pour la gestion globale des ressources dans les grappes de calculateurs : mise en œuvre dans le système d'exploitation Kerrighed

soutenue le 3 mars 2004 devant la commission d'Examen

M. :	Thierry	PRIOL	Président
MM. :	Raymond	NAMYST	Rapporteurs
	Pierre	SENS	
MM. :	Jean-Yves	BERTHOU	Examinateurs
	Jean-Yves	BLANC	
	Yves	DENNEULIN	
	Christine	MORIN	

À la mémoire de mes grands-parents,
À mes parents,
À Aurélia,

Remerciements

Je remercie vivement Thierry Priol, directeur de recherches à l'INRIA, qui me fait l'honneur de présider ce jury.

Je tiens également à remercier ma directrice de thèse Christine Morin, directeur de recherches à l'INRIA, et mon encadreur industriel de ma bourse CIFRE Jean-Yves Berthou, chef de groupe à EDF R&D, pour leur disponibilité, la qualité de leurs conseils, leur soutien. Je remercie Christine pour sa confiance, la grande qualité de son encadrement. Grâce à Christine, j'ai pu bénéficier d'un encadrement scientifique de grande qualité, tout à conjuguant les contraintes impliquant par ma bourse CIFRE. J'ai ainsi pu bénéficier d'un contexte de travail de très grande qualité au sein du laboratoire de recherche qui m'a accueilli, et plus précisement au sein de l'activité de recherche Kerrighed dans laquelle s'inscrit mes travaux.. Je remercie Jean-Yves pour l'autonomie qu'il m'a accordé, et la qualité de son encadrement industriel qui m'a permis de découvrir le milieu industriel avec ces contraintes et ces objectifs. Les qualités personnelles de Christine et Jean-Yves m'ont permis de bénéficier d'une bourse industrielle qui s'est déroulée dans les meilleures conditions envisageables, aussi bien au sein du laboratoire de recherche que du centre de recherche et développement d'EDF.

J'adresse mes plus vifs remerciements à Raymond Namyst, professeur des universités à l'université de Bordeaux 1, et Pierre Sens, Professeur des universités à l'université de Paris 6, qui ont accepté d'être rapporteurs de mon travail de thèse. Je les remercie tout particulièrement de leur disponibilité pour effectuer leur rapport.

Je remercie Jean-Yves Blanc, architecte de systèmes informatique à la Compagnie Générale de Géophysique, et Yves Denneulin, maître de conférence à l'ENSIMAG, d'avoir accepté d'être examinateurs de ma thèse.

Je tiens également à remercier sincèrement tous les membres de l'équipe PARIS de l'IRISA et toutes les personnes d'EDF R&D avec qui j'ai pu travailler. Je tiens particulièrement à remercier Pascal Gallard, Gaël Utard, David Margery, Louis Rilling, Ramamurthy Badrinath et Renaud Lottiaux, membres de l'activité de recherche Kerrighed, ainsi que Éric Fayolle, Ivan Dutka-Malen et Hugues Prisker, d'EDF R&D, qui en plus d'être des collaborateurs de qualité sont devenus des véritables amis.

Je remercie également Laurent Lefèvre, chargé de recherches à l'INRIA et collaborateur technique de ma thèse, pour son soutien et la pertinence de ces remarques lors de nos discussions amicales.

Pour finir, je remercie de tout mon cœur mes parents et Aurélia pour leur soutien moral durant ces trois années de thèse, même pendant mes longues périodes de mauvaise humeur, de découragement ou d'indisponibilité. Sans le soutien de mes parents tout au long de mes études, et sans la présence d'Aurélia, les travaux présentés dans ce document n'auraient pas pu aboutir.

TABLE DES MATIÈRES

1 Introduction **1**
1.1 Contexte . . . 1
1.2 Objectifs . . . 2
1.3 Démarche et contributions . . . 3
1.4 Plan du document . . . 5

I Gestion globale des processus dans les grappes **7**

2 Terminologie et définitions **11**

3 Ordonnancement sur grappe **15**
3.1 Ordonnancement statique . . . 17
3.2 Ordonnancement dynamique . . . 19
3.2.1 Politique d'initiative à l'expéditeur . . . 20
3.2.2 Politique d'initiative au récepteur . . . 20
3.2.3 Politique symétrique . . . 21
3.2.4 Politique aléatoire . . . 21
3.3 Partage temporel et ordonnanceurs de threads . . . 22
3.3.1 Gang-scheduling et co-scheduling . . . 22
3.3.2 Co-scheduling flexible . . . 24
3.3.3 Cas particulier des systèmes de threads . . . 25
3.4 Ordonnancement adaptable . . . 26
3.5 Résumé . . . 28

4 Mécanismes de gestion globale des processus **29**
4.1 Introduction . . . 29
4.2 Déploiement d'applications sur grappe . . . 30
4.2.1 Création distante de processus . . . 30
4.2.2 Duplication de processus . . . 31
4.2.3 Gestion de groupes de processus . . . 31
4.3 Migration de processus . . . 31
4.3.1 Gestion des identificateurs de processus . . . 32

4.3.2 Extraction du processus 33
4.3.3 Transfert du processus 34
4.3.4 Exécution du processus sur le nœud de destination 36
4.4 Création et restauration de points de reprise 38
4.5 Niveau de mise en œuvre 40
4.6 Résumé 41

II Gestion globale des processus dans le système Kerrighed 43

5 Le système d'exploitation Kerrighed 47
5.1 Principes de conception 47
5.2 Architecture générale du système Kerrighed 48
5.3 Gestion globale de la mémoire 49
5.4 Synchronisation distribuée de processus 50
5.5 Gestion globale des flux de données 51
5.6 Gestion globale de données noyau 51
5.7 Gestion globale des processus 52
5.8 Communications haute performance 52
5.9 Outils pour la création de services système distribués 52
5.10 Résumé 53

6 Ordonnancement global adaptable 55
6.1 Objectifs 55
6.2 Approche proposée 57
6.3 Ordonnanceur global modulaire 57
6.4 Mécanismes de gestion globale des processus 59
6.4.1 Définitions 59
6.4.2 Les processus fantômes 59
6.4.3 Interface Pthread 61
6.5 Environnement de développement 61
6.6 Résumé 61

7 Conception d'un ordonnanceur global 63
7.1 Définition des composants 63
7.1.1 Les gestionnaires d'ordonnancement global 64
7.1.2 Les analyseurs locaux 66
7.1.3 Les sondes système 67
7.2 Communications entre composants 69
7.3 Configuration et gestion dynamique 72
7.3.1 Configuration des gestionnaires d'ordonnancement global 74
7.3.2 Configuration des analyseurs locaux 76
7.3.3 Configuration des sondes 76

7.3.4 Politiques auto-adaptables 78
7.4 Résumé 78

8 Mécanismes de gestion des processus 79
8.1 Processus fantôme 79
8.1.1 Introduction 79
8.1.2 Architecture 80
8.1.3 Création du processus fantôme 85
8.1.4 Gestion globale d'un identifiant unique de processus 90
8.1.5 Gestion globale des signaux associés aux processus 91
8.2 Déploiement, migration et points de reprise 91
8.2.1 Déploiement de tâches 91
8.2.2 Migration de processus 94
8.2.3 Création de points de reprise 97
8.3 Résumé 100

9 Exemple d'ordonnanceur global 101
9.1 Prototype des composants 101
9.1.1 Prototype d'une sonde 101
9.1.2 Prototype d'un analyseur local 102
9.1.3 Prototype d'un gestionnaire d'ordonnancement global 103
9.2 Ordonnanceur statique 103
9.2.1 Description de la sonde utilisée 104
9.2.2 Description de l'analyseur local utilisé 105
9.2.3 Description du gestionnaire d'ordonnancement utilisé 106
9.3 Ordonnanceur dynamique 109
9.3.1 Description des sondes utilisées 109
9.3.2 Description des analyseurs locaux utilisés 110
9.3.3 Description du gestionnaire d'ordonnancement global utilisé 111
9.4 Résumé 116

III Mise en œuvre et évaluation 117

10 Élément de mise en œuvre 121
10.1 Le système d'exploitation LINUX 121
10.2 L'ordonnanceur global 121
10.2.1 Éléments de mise en œuvre 121
10.2.2 Mécanisme de configuration dynamique de l'ordonnanceur global . . 131
10.3 Les mécanismes de gestion globale des processus 132
10.3.1 Accès aux mécanismes de KERPROC 132
10.3.2 Mise en œuvre du mécanisme de processus fantôme 135
10.3.3 Mécanisme de migration de processus 142

10.3.4 Mécanisme de duplication de processus dans Kerrighed 142
10.3.5 Mécanisme de création distante de processus 142
10.3.6 Mécanisme de création de points de reprise 143
10.4 Gestion des threads dans KERRIGHED 144
10.4.1 Gestion des threads dans KERRIGHED 144
10.4.2 Création d'un threads 146
10.4.3 Terminaison d'un thread 147
10.4.4 Interface générale propre aux applications constituées de threads . . 148
10.5 Résumé 148

11 Évaluation de performances 149
11.1 Plate-forme d'expérimentation 149
11.2 Description des applications de test 149
11.3 Évaluation de la gestion globale de processus 151
11.3.1 Évaluation du mécanisme de processus fantôme 151
11.3.2 Création distante de processus 152
11.3.3 Duplication de processus 153
11.3.4 Migration de processus 154
11.3.5 Création de points de reprise 158
11.3.6 Restauration d'un point de reprise 159
11.4 Évaluation de l'ordonnanceur global 162
11.4.1 Description des charges applicatives utilisées 162
11.4.2 Description des politiques utilisées 163
11.4.3 Résultats 164
11.5 Évaluation du support Posix thread 168
11.6 Résumé 169

12 Conclusion 171
12.1 Bilan 171
12.2 Perspectives 174

Bibliographie 177

A Fichiers XSL-T de l'ordonnanceur adaptable 187
A.1 Feuille XSL-T générant fichier scheduler_loader.c 187
A.2 Feuille XSL-T générant fichier analyzer_loader.c 190
A.3 Feuille XSL-T générant fichier probe_loader.c 191

B Fichiers générés 193
B.1 Prototype du fichier scheduler_loader.c 193
B.2 Prototype du fichier analyzer_loader.c 193
B.3 Prototype du fichier probe_loader.c 194

1 INTRODUCTION

Les grappes de calculateurs[20, 69] sont désormais couramment utilisées comme alternative aux machines parallèles et ont prouvé leur intérêt, notamment avec l'utilisation de nouvelles technologies comme les réseaux gigabits[18, 46, 79]. De nombreux systèmes pour grappe tentent de faciliter leur utilisation en offrant des mécanismes cachant en partie la distribution des ressources.

Néanmoins, l'utilisation et l'administration des grappes de calculateurs reste complexe. Cette complexité est dûe au manque de logiciels pour gérer la distribution de toutes les ressources au sein de la grappe. Il est par exemple toujours difficile de retirer ou d'ajouter des nœuds sans interruption des services et des applications en cours d'exécution au sein de la grappe, alors que le matériel utilisé dans les grappes peut être sujet aux défaillances. De plus, sauf dans le cas de l'utilisation de grappes pour l'exécution d'une seule application adaptée à l'architecture distribuée des grappes, il est toujours difficile d'utiliser efficacement les ressources disponibles (*e.g.* problème de déploiement, problème d'exécution des applications parallèles à mémoire partagée alors que les mémoires sont physiquement distribuées).

Pour faciliter la programmation et l'utilisation des grappes, une approche est de fournir un système dont toutes les ressources sont gérées globalement, de manière transparente pour l'utilisateur et le programmeur. C'est l'approche retenue dans les systèmes à image unique (« *Single System Image* », SSI)[21] qui ont pour but d'offrir une vision de machine à mémoire partagée de la grappe en offrant au programmeur une interface connue facilitant le développement et l'exécution d'applications sur grappe. Lorsqu'il offre l'interface d'un système d'exploitation traditionnel, un système à image unique facilite considérablement le portage des applications sur les grappes de calculateurs.

1.1 Contexte

Avant l'apparition des grappes de calculateurs[83], en 1995, des travaux de recherche ont cherché à utiliser la puissance de calcul non utilisée des stations de travail distribuées au sein d'une entreprise ou d'une institution et interconnectées par un réseau local. Dans ce type d'architecture, appelé *Network Of Workstation* (NOW)[3], les calculateurs sont utilisés de manière interactive par un utilisateur privilégié.

En 1996, un nouveau concept proche des NOW a été introduit : les architectures BEOWULF [83] qui offrent des outils pour la programmation d'applications parallèle. Parmi ces outils, distribués et parallèles, les systèmes BEOWULF fournissent une bibliothèque

et un environnement de programmation PVM[84], des outils fondés sur le standard MPI [85]. Ces outils permettent de développer des applications parallèles ou distribuées mais ne facilitent pas l'administration de l'ensemble des machines.

L'apparition des systèmes BEOWULF a marqué l'apparition d'une nouvelle architecture dédiée au calcul, comme peut l'être une machine parallèle classique, la grappe de calculateurs.

> Définition 1 (Grappe de calculateurs) *On désigne par grappe de calculateurs un ensemble de calculateurs indépendants interconnectés par un réseau. Les calculateurs sont géographiquement regroupés au sein d'un local et appartiennent tous à la même organisation.*

Aujourd'hui les grappes ont trois principales utilisations : (i) la grappe est utilisée pour exécuter une application dédiée (*e.g* des application MPI ou PVM [84]) ou (ii) la grappe est utilisée pour exécuter plusieurs applications de différents utilisateurs ; le déploiement et la gestion de l'exécution de ces applications est alors géré par un système de batch comme sur certaines machines parallèles.

Dans tous ces cas d'utilisation, la gestion des ressources est effectuée par des outils ou des intergiciels. En revanche, aucun service de gestion des ressources de la grappe n'est mis en œuvre au sein du système d'exploitation alors que le rôle d'un système d'exploitation est de virtualiser les ressources et de gérer le partage des ressources entre les applications. Pour gérer efficacement les ressources d'une grappe de calculateurs, un système d'exploitation spécifique est une approche intéressante pour une utilisation supportant la multiprogrammation et l'exécution de différentes classes d'applications scientifiques. L'utilisation d'un système d'exploitation pour grappe permet de masquer la distribution des ressources, de partager les ressources et de garantir un fonctionnement même en cas de défaillance d'un nœud de la grappe en offrant des services de haute disponibilité. Un système qui remplit ces propriétés est ce que l'on appelle un système à image unique. À ce jour, aucun système ne remplit pleinement toutes ces propriétés même si des tentatives existent (*e.g.* GENESIS [40], MOSIX [13], OPENMOSIX [11], OPENSSI [92]).

1.2 Objectifs

Les travaux présentés dans ce document s'inscrivent dans le contexte de l'activité de recherche KERRIGHED[1] menée au sein du projet INRIA PARIS de l'IRISA et qui vise à concevoir et réaliser un système à image unique. Ils portent plus particulièrement sur la conception et la mise en œuvre d'un ordonnanceur global adaptable et sont réalisés dans le cadre d'une convention CIFRE co-financée par EDF R&D à Clamart (groupe SINETICS) et en collaboration avec le projet INRIA RESO.

KERRIGHED met en œuvre les deux premières propriétés de système d'exploitation SSI par une gestion globale de l'ensemble des ressources logiques d'un système d'exploitation (processus, mémoire, flux de données, fichiers) qui exploite les ressources physiques

[1] Kerrighed est une marque déposée

disponibles dans les différents nœuds de la grappe (processeurs, mémoires, disques, interfaces réseau).

L'objectif de mon travail de thèse est la conception du sous-système de gestion globale des processus. La gestion globale des processus est essentielle au sein d'un système d'exploitation pour grappe car elle conditionne l'utilisation non seulement de la ressource processeur mais aussi des ressources mémoire des différents nœuds et du réseau. La gestion globale des processus est réalisée par un ordonnanceur global de processus. Ce dernier est en charge du placement des processus sur les nœuds de la grappe lors du lancement des applications et de la répartition de la charge au cours de l'exécution des applications de manière à garantir une bonne utilisation des ressources et par conséquent de bonnes performances pour les applications. L'un des axes de mon travail a été de définir une architecture d'ordonnanceur global permettant de configurer la politique d'ordonnancement global. L'ordonnanceur global, pour garantir une utilisation efficace des ressources, doit s'adapter à trois facteurs : les variations de charge de la grappe (*e.g.* les systèmes sont en majorité très utilisés le jour et peu la nuit), l'utilisation des ressources faite par les applications (*e.g.* utilisation d'une grande quantité de mémoire) et enfin les changements de configuration de la grappe (*e.g.* ajout et retrait de nœuds).

La littérature abonde de politiques d'ordonnancement global des processus que ce soit pour le placement des processus à leur création ou l'équilibrage de charge au cours de l'exécution d'un ensemble d'applications.

L'implantation dans les NOW ou les grappes d'un ordonnanceur global nécessite la mise en œuvre de mécanismes de gestion globale des processus efficaces permettant la création de processus à distance et la duplication de processus pour le déploiement d'applications parallèles, la migration de processus pour déplacer un processus entre deux nœuds d'une grappe.

De nombreuses applications scientifiques sont des applications de longue durée. Ces considérations nous ont amené à nous intéresser aux mécanismes de reprise d'applications. Ces mécanismes permettent d'éviter le redémarrage d'une application à son début en cas de défaillance d'un nœud. L'un de nos objectifs a été de concevoir et mettre en œuvre des mécanismes de recouvrement arrière.

L'objectif de mon travail a donc été la conception et la mise en œuvre au sein de KERRIGHED d'un ordonnanceur global configurable permettant de changer dynamiquement la politique d'ordonnancement global et de mécanismes de gestion de processus efficaces pour la création à distance, la duplication, la migration et la sauvegarde et restauration de point de reprise de processus. Le sous-système de gestion globale de processus de KERRIGHED a été conçu de manière à supporter des charges de travail constituées de différents types d'applications : séquentielles, parallèles fondées sur les paradigmes de communication par échange de message ou par mémoire partagée.

1.3 Démarche et contributions

Mon travail s'est organisé autour de quatre axes :

- la conception et la réalisation des mécanismes de gestion de processus efficaces,
- la conception d'une architecture d'ordonnanceur global modulaire et configurable et son implantation au sein de KERRIGHED,
- la conception et la réalisation d'un environnement de développement de politiques d'ordonnancement global,
- l'évaluation des sous-systèmes de gestion globale des processus de KERRIGHED à l'aide de charges de travail comprenant des applications industrielles fournies par EDF R&D.

Chronologiquement, mon travail a d'abord porté sur la conception et la réalisation des mécanismes. En effet, au moment où j'ai commencé mes travaux de thèse, un premier prototype du système GOBELINS, l'ancêtre du système KERRIGHED, était en cours de finalisation. La principale contribution du système GOBELINS était son système de gestion globale de la mémoire dans une grappe fondé sur le concept de conteneur. J'ai donc étudié les mécanismes de gestion globale des processus et j'ai proposé et mis en œuvre des mécanismes de création à distance, de duplication et de migration tirant profit du concept de conteneur.

Les systèmes actuels tels que MOSIX ne permettent ni le déploiement de threads d'applications parallèles fondées sur le paradigme de mémoire partagée sur plusieurs nœuds d'une grappe ni le déplacement de ces threads entre deux nœuds en cours d'exécution. Ces travaux ont donc abouti à la mise en œuvre d'un support *pthread* complet sur KERRIGHED. Mes travaux alliés aux travaux antérieurs de Renaud Lottiaux sur les conteneurs, permettent de prendre en compte les applications à mémoire partagée au sein de l'ordonnanceur global.

J'ai ensuite conçu un ordonnanceur global dont la modularité permet sa configuration dynamique. Il s'agit de lever une autre limitation des systèmes actuels tels que MOSIX dans lesquels la politique d'ordonnancement globale est figée et ne peut donc être modifiée qu'au prix de lourdes modifications au sein du noyau. Grâce à l'architecture modulaire que j'ai proposée pour l'ordonnanceur de KERRIGHED, il est possible de choisir la politique d'ordonnancement à la configuration mais aussi de la changer dynamiquement alors que des applications sont en cours d'exécution soit à la demande de l'administrateur soit automatiquement lorsque les conditions de charge évoluent. Cette caractéristique de l'ordonnanceur global de KERRIGHED nous a conduit au qualificatif d'ordonnanceur adaptable.

La possibilité de configurer l'ordonnanceur global de KERRIGHED encourage au développement et à l'expérimentation de différentes politiques d'ordonnancement. Afin de faciliter la tâche des programmeurs système, nous avons conçu et réalisé un environnement de développement offrant des primitives de base et le squelette de politiques et génèrant les modules à intégrer à KERRIGHED.

Enfin l'ensemble des mécanismes que j'ai implantés dans KERRIGHED ont été évalués expérimentalement et quelques politiques d'ordonnancement global ont été mises en œuvre. Dans le cadre de cette évaluation, nous avons exécuté des applications industrielles fournies par EDF R&D au-dessus de KERRIGHED. Le support *pthread* a été validé par l'installation d'un compilateur OPENMP [26] ciblant les *pthread* sur KERRIGHED. J'ai mis en œuvre l'ensemble des mécanismes que j'ai proposés dans la version 0.72 du système KER-

RIGHED fondé sur un noyau LINUX 2.2.13 qui est diffusé sur le site web de KERRIGHED (http ://www.kerrighed.org). Ils ont été depuis portés avec l'aide de David Margery dans la version 0.80 de KERRIGHED fondée sur le noyau LINUX 2.4.24.

1.4 Plan du document

Ce document est divisé en trois parties. La première partie présente un état de l'art relatif à la gestion globale des processus dans les grappes. Le premier chapitre est consacré à l'ordonnancement global des processus. Le second chapitre a trait aux techniques de gestion de processus utilisées pour mettre en œuvre un mécanisme de migration ou de création de point de reprise de processus.

Dans la seconde partie, nous présentons nos contributions relatives à la gestion globale des processus dans le système d'exploitation à image unique KERRIGHED. Le chapitre 5 présente les principales fonctionnalités du système d'exploitation pour grappe KERRIGHED, cadre de nos travaux. Le chapitre 6 présente l'architecture générale de l'ordonnanceur global proposé dans le système KERRIGHED et le chapitre 7 détaille la conception de l'ordonnanceur adaptable, dynamiquement configurable au sein du système KERRIGHED. Le chapitre 8 présente la conception d'un mécanisme unique de gestion globale des processus indépendamment de leur localisation, qu'il est ensuite possible d'étendre pour mettre en œuvre simplement des mécanismes de migration, de duplication, de création à distance ou encore de création de point de reprise de processus. Le chapitre 9 présente des exemples d'ordonnanceur global mis en œuvre au sein de KERRIGHED.

La dernière partie du document présente quelques éléments de mise en œuvre de nos propositions dans le système KERRIGHED et une évaluation des mécanismes réalisés. Le chapitre 10 décrit la mise en œuvre au sein de KERRIGHED des différents mécanismes étudiés durant cette thèse, et qui permettent de compléter les fonctionnalités du système à image unique KERRIGHED. Le chapitre 11 présente les résultats détaillés des performances de KERRIGHED sur une grappe de PCs, notamment en utilisant les applications industrielles fournies par EDF. Cette évaluation a été plus particulièrement effectuée avec des applications séquentielles et parallèles par mémoire partagée, les mécanismes de gestion globale des flux de données n'ayant pas encore été intégrés au prototype utilisé dans le cadre de cette thèse.

En conclusion, le chapitre 12 dresse un bilan de notre étude et présente les perspectives ouvertes par notre travail.

Première partie

Gestion globale des processus dans les grappes

L'ordonnanceur est l'élément d'un système d'exploitation qui gère l'allocation des processeurs d'une machine aux processus. La politique d'ordonnancement qu'il met en œuvre a un impact important sur les performances des applications.

Dans une grappe de calculateurs, deux niveaux d'ordonnancement sont à considérer : l'ordonnanceur local propre à chaque nœud de la grappe et l'ordonnanceur global. Le rôle de l'ordonnanceur global d'une grappe est de répartir les processus des applications sur les différents nœuds d'une grappe. Afin de garantir une bonne utilisation des ressources au cours de l'exécution d'un ensemble d'applications, l'ordonnanceur global peut être amené à déplacer des processus en cours d'exécution. Déplacer un processus d'un nœud à l'autre repose sur la mise en œuvre d'un mécanisme de migration de processus qui requiert une coopération entre les nœuds de la grappe. D'autres mécanismes tels que la création de processus à distance ou la duplication de processus doivent être implantés pour mettre en œuvre un ordonnanceur global. L'efficacité de ces mécanismes est essentielle au fonctionnement de l'ordonnanceur global d'une grappe.

Cette première partie du document est consacrée à la gestion globale des processus dans une grappe. Le chapitre 2 définit les termes utilisés dans la suite du document. Dans le chapitre 3, nous nous intéressons aux politiques d'ordonnancement global des processus dans une grappe. Dans le chapitre 4, nous présentons les mécanismes de gestion des processus indispensables à la mise en œuvre d'un ordonnanceur global.

2 TERMINOLOGIE ET DÉFINITIONS

Il est utile de préciser certains termes avant de présenter l'état de l'art sur la gestion des processus dans les grappes. L'objectif d'un ordonnanceur est d'exécuter au mieux les tâches qui lui sont soumises.

Définition 2 (Ordonnanceur) *L'ordonnanceur d'un système est un composant qui organise au mieux l'exécution de différentes tâches qui lui sont soumises pour qu'elles soient toutes exécutées correctement.*

La définition d'*exécution correcte* dépend du système considéré. Nous la précisons plus loin.

Définition 3 (Tâche) *Une tâche est un programme à exécuter. Une tâche peut être un programme séquentiel ou parallèle. Elle est constituée d'un ou plusieurs processus.*

Définition 4 (Processus) *Le processus est l'entité qui représente l'exécution d'un programme séquentiel dans un système d'exploitation. La représentation d'un processus dans le système d'exploitation comprend les registres, l'espace d'adressage et la description du processus.*

On distingue deux modèles de programmation parallèle. Dans le modèle à échange de messages, les processus d'une tâche parallèle communiquent uniquement par envoi explicite de messages. Dans le modèle à partage de données, la communication de données entre processus est effectuée en partageant entre ces processus des zones de mémoire contenant les données.

Définition 5 (Thread) *Nous appelons thread un processus d'une tâche parallèle reposant sur le modèle à partage de données. Les threads d'une tâche parallèle partagent leur espace d'adressage.*

Un thread peut être mis en œuvre par un processus (au sens de la définition 3) comme c'est le cas dans le système LINUX [28] pour les threads POSIX [45]. Cependant, certains systèmes mettent en œuvre les threads au niveau utilisateur[5, 48, 38, 65]. Dans ce dernier cas, un processus (au sens de la définition 3) peut servir de support à l'exécution de plusieurs threads. Sauf mention contraire, nous considérons dans la suite de ce document qu'un thread est mis en œuvre par un processus.

Définition 6 (Ordonnancement local) *Nous appelons ordonnanceur local l'ordonnanceur du système d'exploitation d'un nœud de la grappe. L'ordonnanceur local a pour rôle d'allouer le ou les processeurs d'un nœud aux processus à exécuter sur ce nœud.*

Définition 7 (Ordonnancement global) *L'ordonnanceur global d'une grappe a pour rôle de répartir les processus à exécuter sur les différents nœuds d'une grappe.*

L'affectation d'un processus à un nœud est effectuée lors de la création du processus au début de l'exécution d'une tâche. Cette affectation peut être modifiée au cours de l'exécution d'une tâche.

Définition 8 (Migration de processus) *La migration de processus est le déplacement d'un processus entre deux nœuds de la grappe pendant son exécution, de manière préemptive.*

Quelque soit le niveau d'ordonnancement considéré (local ou global), un ordonnanceur met en œuvre une politique d'ordonnancement.

Définition 9 (Politique d'ordonnancement) *Une politique d'ordonnancement définit l'ensemble des règles qui régissent l'allocation des processeurs aux processus.*

Définition 10 (Politique d'ordonnancement local) *La politique mise en œuvre par l'ordonnanceur local est appelée politique d'ordonnancement local.*

Définition 11 (Politique d'ordonnancement global) *La politique mise en œuvre par l'ordonnanceur global est appelée politique d'ordonnancement global.*

Définition 12 (Duplication de processus) *La duplication de processus consiste à créer un ou plusieurs processus identiques à un processus existant sur le même nœud ou sur un nœud distant.*

La duplication de processus est notamment utilisée pour déployer les threads ou processus d'une application parallèle sur une grappe.

L'exécution d'un processus peut être interrompue, soit volontairement soit à la suite de la défaillance du nœud sur lequel il s'exécute. La sauvegarde de points de reprise peut être décidée par le programmeur ou le système (*e.g.* par l'ordonnanceur global) afin de permettre la reprise ultérieure de l'exécution du processus.

Définition 13 (Point de reprise de processus) *Un point de reprise de processus est l'ensemble des informations caractérisant l'état d'un processus à l'instant donné à partir duquel il est possible de reprendre l'exécution du processus.*

Un point de reprise doit être rangé sur un support stable afin de pouvoir le restaurer en cas de défaillance.

Définition 14 (Restauration d'un point de reprise) *La restauration d'un point de reprise consiste à créer un* **nouveau processus** *actif à partir d'un point de reprise. Le système d'exploitation alloue alors un nouvel identifiant de processus, l'ancien ayant pu être réutilisé par le système.*

Définition 15 (Support de stockage stable) *Un support est stable s'il offre des propriétés d'accessibilité, d'inaltérabilité et d'atomicité[51].*

La propriété d'accessibilité est assurée si une donnée stockée sur le support reste accessible

quelle que soit la défaillance survenue. La propriété d'inaltérabilité est assurée si une donnée stockée sur le support ne se trouve pas altérée par une défaillance. Enfin, la propriété de mise à jour est assurée si une donnée stockée sur le support reste dans son état inital si une mise à jour ne réussit pas totalement.

Plusieurs terminologies sont utilisées pour classifier les différents types d'ordonnanceur. Les ordonnanceurs peuvent être classifiés selon les critères suivants :

1. les ordonnanceurs qui placent les processus à leur création sans tenir compte de l'état des nœuds de la grappe,
2. les ordonnanceurs qui placent les processus à leur création en tenant compte de l'état des nœuds de la grappe,
3. les ordonnanceurs qui placent les processus à leur création ou déplacent les processus pendant leur exécution en tenant compte de l'état des nœuds de la grappe.

Les travaux effectués sur les ordonnanceurs ont conduit à différentes terminologies. Les ordonnanceurs qui placent les processus à leur création sans tenir compte de l'état des nœuds sont appelés *ordonnanceurs statiques* alors que les ordonnanceurs prenant en compte l'état des nœuds de la grappe sont appelés *ordonnanceurs dynamiques* [24]. Les ordonnanceurs pouvant déplacer les processus pendant leur exécution sont appelés *ordonnanceurs dynamiques préemptifs* alors que ceux qui n'agissent qu'à la création des processus sont appelés *ordonnanceur dynamiques non préemptifs* [80]. Une autre classification existe où le classement n'est plus effectué en fonction du fait que l'état des nœuds soit pris en compte ou non, mais est effectué en fonction de la capacité à déplacer un processus pendant son exécution. Une politique qui répartit les processus sur les différents nœuds de la grappe à leur création, que ce soit en prenant ou non en compte l'état des nœuds, est appelée politique d'*ordonnancement statique*[81]. Une politique qui peut décider de déplacer des processus pendant l'exécution d'une tâche est alors appelée une politique d'*ordonnancement dynamique*. Dans la suite du document, cette dernière terminologie est utilisée.

L'ordonnancement global au sein d'une grappe est un mécanisme important pour utiliser efficacement les ressources disponibles. L'ordonnancement global se décompose en deux fonctionnalités principales : la **politique d'ordonnancement** et les **mécanismes permettant de manipuler les processus** au sein de la grappe. La séparation de ces deux fonctionnalités comme dans CHARLOTTE [7] permet de découpler les problématiques.

3 ORDONNANCEMENT DE TÂCHES DANS UNE GRAPPE

Un ordonnanceur est chargé de garantir une exécution correcte des tâches qui lui sont soumises. La définition de l'exécution correcte d'une tâche dépend du rôle qu'on assigne au système. Par exemple, dans les systèmes temps-réels, chaque tâche doit être exécutée et terminée dans un intervalle de temps donné. L'ordonnanceur doit faire en sorte que toutes les tâches soient exécutées dans les intervalles de temps qui leur sont impartis. Pour les calculateurs parallèles et les grappes, l'exécution correcte est différente : l'objectif est de fournir une puissance de calcul élevée pour exécuter le plus rapidement possible des tâches gourmandes en puissance de calcul. L'ordonnanceur doit donc minimiser le temps de séjour des tâches dans le système. Plus généralement, dans le domaine du calcul de haute performance, le souci est d'exploiter au mieux les ressources disponibles pour exécuter le plus rapidement possible des tâches. Cela nécessite un environnement multiprogrammé. En effet, si une seule tâche s'exécute sur un système, il est fréquent qu'elle n'exploite pas pleinement l'ensemble des ressources du système. En environnement multiprogrammé, plusieurs tâches peuvent s'exécuter en même temps, et certaines peuvent donc exploiter des ressources que d'autres n'utilisent pas au même moment. Le temps d'exécution de plusieurs tâches à la fois peut ainsi être inférieur au temps d'exécution résultant des exécutions l'une après l'autre de ces tâches. La mise en œuvre de la multiprogrammation nécessite l'utilisation d'un ordonnanceur global. La mission de cet ordonnanceur doit cependant être précisée, en tenant compte du modèle de programmation que le système offre aux utilisateurs (qu'on pourrait appeler architecture virtuelle), de l'architecture physique du système, et des fonctionnalités que le système doit assurer aux utilisateurs.

Les grappes sont des architectures distribuées. Chaque nœud possède sa propre mémoire, ses propres disques et ses propres ressources réseau. Deux modèles de programmation d'applications parallèles existent, (i) le modèle à échange de messages et (ii) le modèle par partage de données. Le modèle de programmation par échange de messages est particulièrement adapté aux architectures dont les ressources sont distribuées. En effet, avec ce modèle de programmation, ce sont les programmeurs qui gèrent la distribution des données lors de la programmation de l'application parallèle, les échanges s'effectuant par échanges de messages explicites via le réseau. Le rôle de l'ordonnanceur se limite donc au déploiement des tâches à leur création et éventuellement au déplacement de processus de la tâche parallèle pour équilibrer la charge de la grappe. Le modèle de programmation par partage de données n'est en revanche pas adapté aux architectures distribuées comme

les grappes. Le programmeur ne dispose pas dans une architecture distribuée d'une vision globale des mémoires des différents nœuds. L'ordonnanceur doit donc disposer d'une gestion globale des ressources de la grappe, non seulement pour le déploiement de la tâche mais également pour équilibrer la charge. Les systèmes à image unique sont des systèmes proposant une gestion globale et transparente des ressources et sont donc intéressants pour l'exécution de tâches parallèles sur grappe.

Une politique d'ordonnancement répartit les tâches au sein de la grappe suivant deux modèles de partage des ressources, par partage de temps ou d'espace. Tout d'abord, l'ordonnanceur peut agir sur le partage de temps. Lorsque plusieurs applications s'exécutent au sein d'un système, celui-ci alloue des tranches de temps sur les processeurs pour l'exécution de chaque processus, donnant ainsi l'illusion que les tâches s'exécutent en même temps. L'ordonnanceur peut également agir sur le partage spatial en modifiant la répartition des processus en exécution au sein de la grappe. Alors que le partage temporel est sous la responsabilité de l'ordonnanceur local (l'ordonnanceur local attribue des tranches de temps sur un processeur pour l'exécution d'un processus), le partage spatial est sous la responsabilité de l'ordonnanceur global (l'ordonnanceur global place les processus au sein de la grappe, à leur création ou pendant leur exécution). Néanmoins, l'ordonnanceur global peut agir sur le partage temporel en modifiant la liste des processus s'exécutant sur un nœud. De plus, alors que pour le partage spatial les processus ne partagent pas les ressources des nœuds de la grappe, le déplacement d'un processus pendant son exécution peut créer un partage temporel des ressources en plaçant sur un même nœud des processus de tâches différentes. Dans la suite du document, nous considérons que le partage spatial est principalement à la charge de l'ordonnanceur global alors que le partage de temps est à la charge de l'ordonnanceur local. Le paragraphe 3.3 décrit néanmoins comment l'ordonnanceur global peut influencer le partage temporel des ressources au sein de la grappe.

Les grappes peuvent accueillir des **charges de travail variées** : des applications séquentielles ou parallèles qui peuvent avoir des accès aux ressources également très variés (*e.g.* utilisation d'une grande quantité de mémoire, utilisation importante de la ressource processeur). Il est donc difficile de prévoir la charge d'une grappe. La **nature de la grappe** peut également influer sur la répartition des tâches sur les différents nœuds. En effet, alors qu'une grappe **homogène**[1] permet de définir simplement des fonctions d'ordonnancement efficaces pour le déploiement d'une application (non prise en compte du paramètre d'hétérogénéité des nœuds). Une grappe constituée de nœuds **hétérogènes** peut poser des problèmes d'ordonnancement. Sans politique d'ordonnancement prenant en compte cette hétérogénéité, des nœuds exécutent les tâches plus rapidement que d'autres ; certain nœuds pourront alors être surchargés alors que d'autres seront sous utilisés. La **fréquence de soumission des tâches** peut également influencer les performances d'une politique d'ordonnancement. En effet, une politique fondée sur une estimation probabiliste de la charge des nœuds peut ne pas être efficace pendant une période de soumission extrême de tâches. La politique peut alors placer sur un nœud de nombreuses tâches alors que d'autres

[1]Nous parlons dans ce paragraphe d'homogénéité ou d'hétérogénéité en terme de puissance, *i.e.* chaque nœud dispose des mêmes types de ressource.

nœuds peuvent être sous-utilisés. Enfin, quelque soit le type d'application en cours d'exécution, la nature de la grappe ou encore la fréquence de soumission de nouvelles tâches, les grappes de calculateurs nécessitent régulièrement des actions d'administration en vue de prévenir/résoudre des problèmes sur certains nœuds. Dans ce cas, le retrait puis l'ajout programmé de nœuds est un mécanisme intéressant, nécessitant de pouvoir déplacer les applications en cours d'exécution. Tous ces facteurs rendent les ordonnanceurs adaptables intéressants, car ils permettent de personnaliser la politique d'ordonnancement par rapport à la charge de travail en cours d'exécution. La charge de travail diffère selon les applications qui la constituent en fonction de leur profil d'utilisation des ressources.

Le paragraphe 2 présente la terminologie et quelque définitions utilisés le reste du document. Les paragraphes 3.1 et 3.2 présente les ordonnanceurs partageant l'espace. Le paragraphe 3.1 présente les ordonnanceurs statiques alors que le paragraphe 3.2 présente les ordonnanceurs dynamiques. Le paragraphe 3.3 présente les ordonnanceurs partageant le temps et les ordonnanceurs spécialisés pour l'exécution de threads. Le paragraphe 3.4 présente les ordonnanceurs adaptables. Enfin, nous donnons un résumé dans le paragraphe 3.5.

3.1 Ordonnancement statique

Définition 16 (ordonnancement statique) *L'ordonnancement statique est le placement des tâches à leur création, sans qu'il soit possible de les déplacer durant leur exécution par la suite. Ce placement peut être effectué en considérant l'état des nœuds ou non.*

L'ordonnancement statique peut être effectué de deux façons différentes : placement simple, ou placement intelligent. Le placement simple est un placement qui ne tient pas compte de l'état des nœuds, des ressources disponibles sur chacun d'entre eux.

Les premiers systèmes offrant un ordonnanceur statique sont issus des machines parallèles avec les systèmes de *batch*. Le principe d'un système de batch est de placer sur les nœuds de la grappe les tâches qui lui sont soumises. Pour cela, celui-ci s'autorise à différer l'exécution de la tâche pour la déployer à un moment jugé plus opportun afin de garantir de bonnes performances. Pour différer l'exécution des tâches, le système de batch utilise des files d'attentes. Avec les systèmes de batch, plusieurs tâches peuvent s'exécuter parallèlement souvent en utilisant un partitionnement de l'espace. Une tâche soumise est déployée sur un sous-ensemble des nœuds réservé à son usage exclusif pour qu'elle s'exécute au plus vite (en principe un processus d'une tâche par processeur). Lorsqu'un processus se termine, il est remplacé par le processus d'une autre tâche dès que le système de batch estime que les conditions nécessaires au lancement d'une nouvelle tâche sont réunies. Le placement des processus d'une tâche peut être effectué en tenant compte de l'état des nœuds de la grappe ou de manière statique (sans connaître l'état des nœuds). Aujourd'hui des systèmes de batch plus génériques existent. Par exemple, NQS [49] et PBS [43] permettent le partage de nœuds entre plusieurs tâches afin de rendre le système plus flexible et d'utiliser plus efficacement les ressources des nœuds.

PBS est un système de batch qui permet de partager les ressources d'un nœud entre les processus de tâches différentes. Pour cela, PBS organise le placement des tâches en fonction de l'utilisation des ressources des nœuds de la grappe. PBS est composé de trois principales parties :

- Le *Job Executor* mis en œuvre par un démon sur chaque nœud et appelé **MOM**. Ce démon est chargé de recevoir les tâches à exécuter, et de fournir des informations système sur le nœud.
- Le *Job Server* (également appelé **serveur**) s'exécute sur le nœud principal de PBS. Ce démon envoie les tâches en attente vers des nœuds de calcul et récupère les informations système envoyées par chacun des nœuds.
- Le *Job Scheduler* qui met en œuvre la politique d'ordonnancement.

La mise en œuvre d'une politique d'ordonnancement statique fondée sur l'état des nœuds est difficile à mettre en œuvre. Les politiques d'ordonnancement fournies avec PBS ne permettent pas d'offrir des performances satisfaisantes pour l'exécution de tout type de tâches. Par exemple, l'une des politiques de base de PBS est une politique FIFO (les premières tâches soumises sont les premières à être lancées) mais le mécanisme de prévention des famines peut modifier l'ordre de soumission défini par une politique FIFO stricte. Cependant l'architecture modulaire de PBS, et le language spécialisé[2] offert avec le *Job Scheduler*, permettent de mettre en œuvre d'autres politiques d'ordonnancement. Cette fonctionnalité a permis de mettre en œuvre plusieurs politiques dont la politique MAUI [17]. La politique MAUI est fondée sur l'attribution de priorités aux tâches soumises (cette attribution peut être ajustée par des paramètres). Dans un premier temps, MAUI lance le plus possible de tâches prioritaires, puis fait une réservation pour le lancement de la prochaine tâche prioritaire afin que celle-ci puisse être lancée le plus rapidement possible. Ensuite, MAUI tente de trouver des tâches de faible priorité pouvant être exécutées pendant les tranches de temps libres restantes. Cette approche garantit une date de lancement pour les tâches importantes et évite les temps d'attente trop longs pour les petites tâches.

D'autres systèmes offrent un ordonnancement statique. C'est notamment le cas dans les grappes BEOWULF qui offrent des outils pour la programmation d'applications parallèles. Parmi ces outils, distribués et parallèles, se trouvent la bibliothèque et l'environnement de programmation PVM[84], ou encore des outils fondés sur le standard MPI. Dans de tels environnements de programmation, le placement des tâches se limite à une répartition homogène des tâches en fonction du nombre de nœuds disponibles. Les systèmes BEOWULF fournissent des outils MPI et PVM pour développer des applications fondées sur le paradigme de programmation par échange de messages qui est particulièrement bien adapté à la mise en œuvre d'applications pour grappe (gestion explicite des accès aux ressources de la grappe). De tels systèmes n'offrent donc pas un placement efficace pour tous les contextes d'utilisation de la grappe (*e.g.* un nœud utilisé directement par un utilisateur pour la soumission directe d'applications concurrentes). Si par exemple, un nœud

[2]Le *Batch Scheduling Language* - BaSL - est utilisable avec les languages de programmation Tcl ou encore le langage C.

se trouve chargé, celui-ci n'est pas pour autant évité lors du placement de nouvelles tâches. C'est un inconvénient des approches dans lesquelles l'environnement de programmation a la charge de traitements comme l'ordonnancement normalement dévolus au système d'exploitation.

3.2 Ordonnancement dynamique

> Définition 17 (Ordonnancement dynamique) *L'ordonnancement dynamique consiste à pouvoir déplacer des tâches en cours d'exécution afin de pouvoir utiliser au mieux les ressources du système, et donc d'exécuter les tâches le plus efficacement possible en évitant de ralentir leur exécution par un blocage dû à une attente de résolution d'une requête d'accès à une ressource. L'ordonnancement dynamique permet donc non seulement de soulager une ressource du système ayant une importante utilisation, mais aussi prévenir un stress d'utilisation trop important d'une ressource.*

L'ordonnancement préemptif permet de déplacer des processus pendant leur exécution. Le déplacement de processus permet d'équilibrer la charge dynamiquement et de libérer un nœud de la grappe (*e.g.* pour un arrêt programmé du nœud). Cela permet de modifier le partage spatial des ressources mais cela peut également modifier le partage temporel d'un nœud. En effet, le déplacement d'un processus peut amener un processus sur un nœud sur lequel un processus d'une autre tâche s'exécutait déjà. Dans ce cas, plusieurs tâches se retrouvent sur un nœud effectuant ainsi un partage temporel.

Pour effectuer un partage spatial efficace, le système doit être capable de déplacer un processus d'un nœud à l'autre en plus des mécanismes de placement de processus. Plusieurs politiques sont possibles [59], les principales communément décrites dans la littérature étant :

- **Une politique d'initiative à l'expéditeur.** La politique d'ordonnancement est activée lorsqu'un nœud est surchargé. Le nœud (l'**expéditeur**) déplace alors des processus s'exécutant localement vers d'autres nœuds. Cette politique est adaptée pour les grappes ayant une charge moyenne peu élevée (*i.e.* dont les ressources sont globalement peu utilisées).
- **Une politique d'initiative au récepteur.** La politique d'ordonnancement est activée lorsqu'un nœud est sous-utilisé. Le nœud (le **récepteur**) reçoit alors à sa demande des processus s'exécutant sur d'autres nœuds, *a priori* plus chargés. Cette politique est adaptée pour les grappes ayant une forte charge (*i.e.* dont les ressources sont globalement très utilisées).
- **Une politique symétrique.** Ce type de politique est une combinaison des deux politiques précédentes. Cette politique est adaptée pour les grappes ayant une charge moyenne ni trop importante, ni trop faible (*i.e.* dont les ressources sont globalement moyennement utilisées).
- **Une politique aléatoire.** Les nœuds utilisés pour l'ordonnancement d'un processus/thread sont choisis aléatoirement.

L'ordonnancement dynamique repose donc sur un mécanisme essentiel : la migration de processus. La réalisation d'un tel mécanisme est abordée dans le chapitre 4.

3.2.1 Politique d'initiative à l'expéditeur

Dans les politiques d'initiative à l'expéditeur, un nœud décide de déplacer un processus qui s'exécute localement vers un nœud distant en vue d'utiliser les ressources de la grappe de manière plus efficace.

C'est notamment le cas de MOSIX (et de OPENMOSIX qui est un projet issu de MOSIX), un système d'exploitation pour grappe visant à offrir une vision unique de la grappe. MOSIX est une extension du noyau Linux offrant un mécanisme d'ordonnancement de processus pour grappe. Pour cela, l'ordonnancement global met en œuvre une politique d'ordonnancement probabiliste fondée sur une vision partielle de l'utilisation des processeurs de la grappe. Les informations relatives à la charge processeur sont extraites du système local (pour cela, le noyau Linux a été modifié) pour être ensuite envoyées vers un sous-ensemble de nœuds.

L'ordonnanceur global de MOSIX mesure une moyenne de la longueur de la file des processus prêts sur une longueur de temps au moins comparable au temps nécessaire pour déplacer un processus. Parallèlement à cette gestion de la ressource processeur, MOSIX repousse au plus tard la nécessité de paginer la mémoire sur disque en s'assurant en priorité que la mémoire physique libre sur chaque nœud ne passe pas en dessous d'un certain seuil [12]. Dès qu'elle passe sous ce seuil sur un nœud, l'ordonnanceur tente de déplacer un processus de ce nœud vers un nœud moins chargé en mémoire, même si ceci introduit un déséquilibre dans la charge processeur.

Si l'ordonnanceur détecte un déséquilibre, celui-ci peut déplacer des processus après une phase de négociation avec un nœud cible déterminé par la politique d'ordonnancement probabiliste.

Les travaux effectués dans MOSIX ont été en partie repris dans le système d'exploitation pour grappe OPENSSI. OPENSSI est construit par intégration de divers composants issus du monde académique et industriel pour offrir un système à image unique dont le code est disponible pour grappe LINUX (sous licence GPL).

3.2.2 Politique d'initiative au récepteur

Dans les politiques d'initiative au récepteur, un nœud demande à un autre nœud un processus lorsque celui-ci est peu chargé par rapport aux autres nœuds. Cette approche est intéressante pour les grappes dont les ressources sont très utilisées : les nœuds ayant une utilisation importante de leurs ressources n'ont pas à gérer en plus l'ordonnancement des processus, minimisant ainsi les ressources utilisées pour effectuer cet ordonnancement. Ce sont les nœuds sous utilisés qui demandent la migration de processus.

3.2.3 Politique symétrique

Dans les politiques symétriques, les deux approches d'initiative à l'expéditeur et au récepteur sont utilisées : un nœud peut décider de déplacer un processus, tout comme il peut demander la migration d'un processus s'exécutant sur un nœud distant.

Ce type de politique est donc particulièrement adapté pour les systèmes visant à tirer profit de réseaux de station de travail. Dans ce type d'architecture, le but est d'utiliser les machines non utilisées par l'utilisateur habituel (appelé propriétaire) pour exécuter des tâches dont le temps d'exécution est long. Une machine sous-utilisée peut donc demander un processus, et une machine dont le propriétaire redevient actif peut se débarasser des processus externes (n'appartenant au propriétaire) pour que la machine puisse être utilisée en mode interactif par son propriétaire. C'est notamment le cas dans le système SPRITE [31] : une machine qui se trouve non utilisée signale au système qu'elle peut accueillir des tâches externes, tandis que si l'utilisateur de la machine revient, le nœud se débarrasse des tâches externes afin de ne pas pénaliser l'utilisateur de la machine.

3.2.4 Politique aléatoire

Dans la politique aléatoire, un nœud souhaitant déplacer un processus s'exécutant localement choisit un nœud distant de manière aléatoire [50]. Des travaux ont montré que l'utilisation d'une telle politique permet un gain substantiel de performance dans certains cas.

L'ordonnancement statique (non préemptif) est fondé sur la création distante de processus et il est donc peu coûteux, peu d'informations étant à envoyer sur le nœud distant (référence sur le fichier source), et le surcoût d'exécution impliqué par la création distante est plus faible que dans le cas de la migration. Cette technique est donc particulièrement bien adaptée aux applications ayant un temps d'exécution court. De plus, des études théoriques ont montré qu'un placement efficace des tâches lors de leur création permet dans de nombreux cas d'équilibrer la charge globale de la grappe, réduisant grandement l'intérêt de la migration et donc de l'ordonnancement dynamique[32].

Cependant, l'ordonnancement statique ne permet pas de répondre à tous les besoins d'ordonnancement au sein d'une grappe. Si des nœuds de la grappe se trouvent ajoutés (*e.g.* en cas d'ajout de nœuds), les applications en cours d'exécution ne pourront pas tirer profit des ressources disponibles sur les nœuds sous-utilisés. Dans ce cas, un mécanisme d'ordonnancement dynamique peut permettre d'améliorer l'exécution globale des applications. De la même manière, un ordonnancement global peut être intéressant pour certains types d'applications telles que celles ayant un temps d'exécution long. Dans ce cas, le coût de la migration de tâche est négligeable par rapport au temps d'exécution des tâches.

L'ordonnancement dynamique est également un mécanisme indispensable pour l'ajout et le retrait de nœuds. En cas de retrait, le système doit être capable de déplacer l'ensemble des applications en cours d'exécution vers d'autres nœuds, alors que dans le cas d'ajout de nœuds, le système doit être capable de tirer profit le plus rapidement possible des ressources mises à disposition pour garantir une exécution efficace des applications.

3.3 Partage temporel et ordonnanceurs de threads

En plus d'agir sur le partage spatial des ressources, l'ordonnanceur peut agir sur le partage temporel. De nombreux travaux ont été menés pour améliorer la performance des applications parallèles en intervenant uniquement sur le partage temporel, *i.e.* sur l'ordre et le temps d'exécution des différents processus (ou threads) de l'application parallèle (ces politiques n'utilisent donc pas de mécanismes pour déplacer les processus). Le système d'-exploitation alloue successivement pour chaque processeur des tranches de temps pendant lesquelles un processus s'exécute. Ce partitionnement temporel permet d'exécuter plusieurs processus sur un même processeur, en donnant l'impression à un utilisateur qu'ils s'exécutent tous en même temps. Or, si les processus d'une application parallèle (par passage de message ou mémoire partagée) ne sont pas exécutés de manière cohérente sur les différents processeurs disponibles, il est possible qu'un ensemble de processus initie une communication ou une synchronisation avec d'autres processus de l'application alors que ceux-ci sont en attente d'une tranche d'exécution sur un processeur. Dans ce cas, la tranche de temps allouée au processus/thread qui a initié une communication ou une synchronisation est perdue, dégradant la performance globale de l'application. Il est donc possible, en agissant localement sur chaque nœud sur l'ordonnancement des processus d'une application parallèle sur les différents processeurs d'augmenter les performances globales de l'application. Une de ces politiques est la politique de « *gang-scheduling* » qui vise à optimiser l'exécution des processus communiquant entre eux en minimisant les temps d'attente lors de communications.

3.3.1 Gang-scheduling et co-scheduling

Lorsque deux processus s'échangent un message, si le processus récepteur est en cours d'exécution sur un processeur alors que l'émetteur attend que le système lui alloue un temps d'exécution sur un autre processeur, le processus récepteur attend l'arrivée du message pendant le temps qui lui est assigné alors que ce message n'arrivera que lorsque le processus émetteur pourra s'exécuter sur un processeur. Le temps alloué à l'exécution du processus récepteur est donc gaspillé en attente inutile. Pour résoudre ce problème Ousterhout a proposé que des processus qui communiquent et s'exécutent sur des processeurs distincts soient ordonnancés en même temps [68]. Cette technique, dite de « *gang scheduling* », consiste pour une application parallèle à simuler au mieux une exécution sur une machine parallèle où elle serait seule. C'est d'ailleurs sur ce modèle que sont programmées les tâches parallèles, étant donné qu'il est impossible de tenir compte des autres tâches dans la programmation.

3.3.1.1 Principe du *gang scheduling*

Le « *gang scheduling* » consiste à effectuer un changement de contexte simultané sur plusieurs processeurs pour exécuter en même temps les processus d'une même tâche parallèle [35]. La mise en œuvre se fait en définissant une matrice ayant autant de colonnes qu'il

y a de processeurs, en représentant une tranche de temps par ligne. Les identificateurs des processus d'une même tâche se trouvent dans les colonnes correspondant aux processeurs sur lesquels ils s'exécutent respectivement, et sur la même ligne. Pendant une même tranche de temps, les processeurs exécutent les processus d'une même ligne, et au changement de contexte suivant, ils exécutent tous les processus de la ligne suivante. Lorsqu'une case de la ligne de la tranche de temps courante ne contient pas de numéro de processus, le processeur correspondant exécute un processus indépendant des tâches parallèles présentes dans la matrice, en attendant de passer à la ligne suivante.

Cette combinaison du partitionnement des nœuds et du partage de temps conduit à un repartitionnement de l'ensemble des processeurs à chaque changement de contexte. Il est cependant coûteux d'effectuer un changement de contexte simultané sur l'ensemble des processeurs à chaque nouvelle tranche de temps. Pour cela, une variante dite de *contrôle hiérarchique distribué* [34] est fondée sur l'observation qu'une synchronisation de changement de contexte entre plusieurs groupes de processeurs n'est nécessaire que si l'un des groupes de la tranche courante peut comprendre un processeur qu'il ne comprenait pas auparavant. Le contrôle hiérarchique distribué consiste donc à organiser les tranches de temps de manière à ne nécessiter une synchronisation entre tous les processeurs pour le changement de contexte qu'un minimum de fois. Pour cela, les premières tranches contiennent les tâches de plus grande taille (en nombre de nœuds utilisés) qui nécessitent une synchronisation entre tous les nœuds, puis les tranches suivantes partitionnent l'espace des processeurs en deux pour exécuter des tâches plus petites, et ainsi de suite dans chaque partition jusqu'à avoir parcouru toutes les lignes (figure 3.1). Ensuite, le cycle reprend.

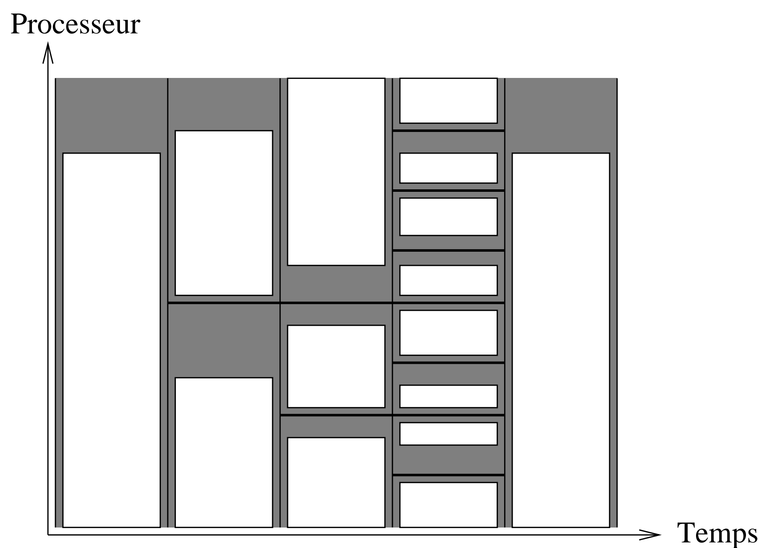

FIG. 3.1 – Contrôle hiérarchique distribué

3.3.1.2 Détermination des groupes

Le but de l'ordonnanceur de co-scheduling flexible est de simuler au mieux pour une application parallèle une exécution sur une machine parallèle où elle serait seule. L'ordonnanceur doit donc savoir quels sont les processus appartenant à une même tâche. L'approche la plus simple pour le système est que la tâche indique au système quels sont les processus qu'elle comprend et qui doivent donc être exécutés dans les mêmes tranches de temps. L'inconvénient de cette approche est justement qu'elle oblige les tâches à fournir au système les identités des processus qu'elles comprennent. Si on veut pouvoir bénéficier du *gang scheduling* dans un cadre plus général, c'est au système de déterminer quels sont les processus qui communiquent entre eux et doivent donc être ordonnancés en même temps. Ces variantes sont parfois appelées « *co-scheduling* »[3]. Le système détermine des affinités entre processus en surveillant principalement le trafic de messages échangés.

3.3.2 Co-scheduling flexible

La politique d'ordonnancement de co-scheduling flexible[37] vise à optimiser l'utilisation des ressources disponibles dans une grappe de calculateurs en analysant le temps passé par le processus dans une attente pour une communication bloquante (une librairie MPI a été modifiée pour calculer et stocker ces temps d'attente). Ce temps permet de classer les différents processus parmi trois groupes :

1. *CS* (« *co-scheduling* ») : les processus communiquent souvent et nécessitent d'être ordonnancés en groupe (« *gang-scheduling* ») à cause de leurs besoins en synchronisation.
2. *F* (« *frustrated* ») : les processus ont suffisamment de besoins en synchronisation mais à cause d'un ordonnancement non efficace au sein de la grappe, ils ne peuvent en général pas utiliser tout leur temps processeur alloué.
3. *DC* (« *don't care* ») : les processus se synchronisent rarement et peuvent donc être synchronisés indépendamment. Une autre classe appelée *RE* (« *rate-equivalent* ») permet de classer les tâches demandant peu de synchronisations mais nécessitant un temps processeur similaire pour tous leurs processus. Cependant la détection des processus de classe *RE* ne pouvant pas s'effectuer uniquement à l'aide d'informations locales durant leur exécution, ils sont également classés comme *DC* à cause de leurs faibles besoins en synchronisation.

À chacune de ces classes est associée une politique d'ordonnancement :

1. les processus *CS* sont ordonnancés suivant une politique de « *co-scheduling* » et ne peuvent pas être préemptés.
2. les processus *F* sont ordonnancés suivant une politique de « *co-scheduling* » mais peuvent être préemptés lorsque les synchronisations ne sont pas efficaces.

[3]Historiquement, Ousterhout a proposé des algorithmes de *co-scheduling* et le *gang scheduling* a été introduit plus tard [35].

3. les processus *DC* sont ordonnancés sans restriction sur la politique utilisée.

Enfin, un algorithme permet de classer les différents processus au sein des classes, après analyse des différents temps passés en attente de communication. La classe d'un processus est calculée en fonction de son temps moyen d'attente en communication et de son temps moyen d'utilisation du processeur par communication. Il a été montré que cette politique d'ordonnancement est très intéressante pour les applications communicantes de type MPI. Cette politique d'ordonnancement a été comparée à différentes politiques dont le « *gang scheduling* » et une politique de premier arrivé, premier servi (« *first-come-first-served* »). Les résultats montrent que la politique de co-scheduling flexible offre de très bonnes performances pour les applications ayant un grain fin de parallélisme, aussi bien que pour des charges de travail provoquant des déséquilibres importants de charge.

Cette politique est donc très intéressante même si elle nécessite une légère modification de l'ordonnanceur local des nœuds (pour que l'ordonnanceur local attribue des tranches de temps processeur aux processus qui soient plus adaptées à la politique de co-scheduling flexible que ne l'est la durée par défaut des tranches de temps allouées par le système non modifié), ainsi que de la librairie MPI utilisée pour acquérir les informations système nécessaires à la politique d'ordonnancement.

3.3.3 Cas particulier des systèmes de threads

Nous avons vu dans le paragraphe 3.2 que l'ordonnancement dynamique de processus peut créer un partage temporel des ressources dans le cas où un processus est déplacé sur un nœud de la grappe accueillant déjà un processus d'une autre tâche. Pour les systèmes gérant des threads, cette affinité entre le partage temporel et spatial est encore plus importante.

Dans un système de gestion de threads, les threads communiquent par partage de données. Ce partage de données crée des communications implicites si les threads ne s'exécutent pas sur le même nœud. Ces communications implicites sont dûes à la gestion des pages mémoire par le système de mémoire virtuelle partagée mis en œuvre pour que les threads d'une tâche parallèle puissent accéder aux données partagées quelque soit leur localisation. Contrairement à une machine disposant d'une unique mémoire centrale où les accès sont peu coûteux, les communications entre nœuds créées par le partage de données sont coûteuses, les réseaux utilisés au sein des grappes étant encore aujourd'hui bien moins rapides que les bus mémoires des systèmes modernes.

L'ordonnancement de threads sur grappe doit donc répondre à deux contraintes :

1. trouver la meilleure distribution des threads au sein de la grappe afin de garantir une utilisation efficace des ressources distribuées de la grappe (*i.e.* trouver le meilleur partage spatial),
2. minimiser les communications entre threads afin de ne pas pénaliser l'exécution des tâches parallèles (*i.e.* trouver le meilleur partage temporel).

Ces contraintes sont souvent contradictoires. L'ordonnancement de threads consiste donc à trouver le meilleur compromis entre ces deux contraintes.

Pour illustrer cette contradiction, le graphe 3.2 montre le problème de modélisation des communications sur une grappe : les sommets de gauche représentent les processus, tandis que les sommets de droite représentent les données partagées (il est inutile de représenter les données non partagées propres à un nœud). Chaque nœud est représenté par une région

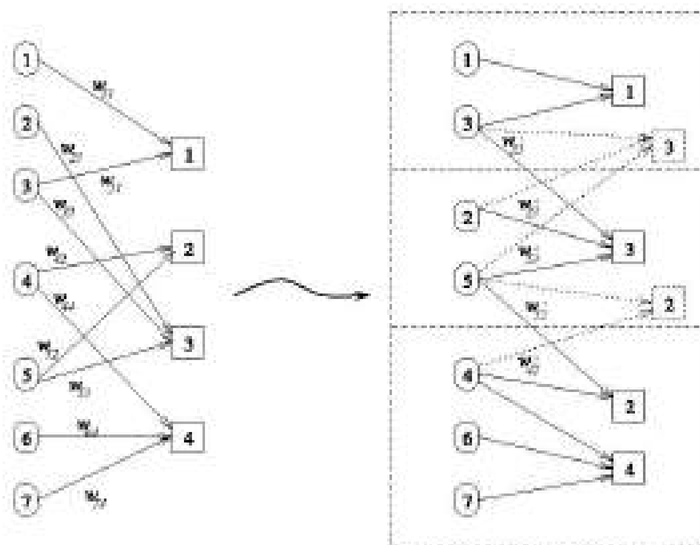

FIG. 3.2 – Graphe biparti de modélisation des communications dans une grappe, et un des placements possibles sur une grappe de trois nœuds

englobant un sous-ensemble des processus et un sous-ensemble des pages mémoires. Chaque arc d'un processus p vers un élément de mémoire partagé m est valué par la fréquence d'accès des processus du nœud vers l'unité de mémoire partagée w_{pm}. La minimisation des communications dans la grappe revient à trouver une disposition des processus et des données sur les différents nœuds de manière à minimiser la somme des relations w_{pm}, pour m et p situés sur deux nœuds différents. Ce problème est appelé « *multi-way cut* »[78]. Ce problème est NP-difficile[27] et sa taille est très grande dans le cas d'une mémoire virtuelle partagée, où le nombre de pages mémoire et de processus est élevé. Des approximations de cet algorithme existent [22] qui permettent de trouver une disposition des processus sur les noeuds de manière approchant le placement optimal (le ratio de conformité par rapport au placement optimal est $1.5 - 1/k$, où k est le nombre de sommets du graphe).

3.4 Ordonnancement adaptable

On a vu dans les paragraphes 3.2 et 3.1 que les ordonnancements statiques et dynamiques permettent de résoudre des problèmes différents, chacun apportant une solution à une charge de travail. Il est donc naturel de réfléchir à la possibilité d'adapter la politique d'ordonnancement à une charge de travail qui est cours d'exécution. Par exemple,

les grappes de calculateurs peuvent être utilisées comme des serveurs départementaux de calcul, et dans ce cas, la charge de travail en cours d'exécution peut varier sur de longues périodes en fonction des projets menés par le département.

Cette adaptabilité de l'ordonnancement est un sujet peu abordé. Pour les machines mono-processeur, le système BOSSA [52] permet de définir une nouvelle politique d'ordonnancement sans modification lourde du système. Cette personnalisation de l'ordonnancement permet également de développer beaucoup plus simplement de nouvelles politiques. En effet, de nombreux travaux ont été menés pour définir de nouvelles politiques mais peu de systèmes offrent un cadre de développement facilitant leur mise en œuvre. Aussi, un important et coûteux travail de mise en œuvre au sein du système doit être effectué pour chaque nouvelle politique. Le système BOSSA offre un tel cadre de développement, qui permet de mettre en œuvre rapidement de nouvelles politiques grâce à un langage spécialisé faisant abstraction de la programmation système, le DSL [14], et à un mécanisme de vérification de propriétés critiques de la politique.

Malheureusement, le système BOSSA ne permet pas d'administrer dynamiquement le système. Pour changer de politique d'ordonnancement, il est nécessaire de recompiler le noyau BOSSA et d'installer cette nouvelle version. Il est donc impossible de changer la politique d'ordonnancement sans arrêter totalement le système. Une autre limitation importante du système BOSSA est que ce système est dédié aux machines mono-processeurs, et ne permet donc pas de définir des ordonnanceurs pour grappe.

Dans le domaine des grappes de calculateurs, des travaux d'ordonnancement adaptable ont été menés sur le micro-noyau MACH [1]. Ces travaux ont permis de mettre en œuvre DSF [55] (« *Distributed Scheduling Framework* ») qui permet à partir de primitives de base de créer de nouvelles politiques d'ordonnancement au sein de la grappe. Le système DSF est entièrement mis en œuvre dans l'espace utilisateur. La mise en œuvre et la maintenance sont donc simplifiées et cela permet d'utiliser des informations fournies par un service système du micro-noyau. En revanche, DSF ne présente qu'un nombre limité de primitives d'ordonnancement, réduisant la nature de la politique pouvant être mise en œuvre (pas de primitives de création de point de reprise permettant par exemple de suspendre l'exécution d'un processus). De même, le système ne dispose pas de mécanisme permettant de dynamiquement modifier la politique d'ordonnancement, sans arrêt des services système, ni des applications, rendant le déploiement de nouvelles politiques complexe.

En contrepartie, les services systèmes sont mis en œuvre dans l'espace utilisateur et le micro-noyau a des difficultés pour gérer la priorité des services systèmes par rapport aux applications utilisateurs en cours d'exécution. Si un nœud accueille des tâches de calcul intensif, le mécanisme d'ordonnancement peut se trouver pénalisé et donc peu réactif. Pour résoudre ce manque de réactivité du système, le système AMOEBA [82, 99, 81] met en œuvre une partie de l'ordonnanceur global (en fait le mécanisme de migration) dans le noyau. Cette approche, associée à une politique locale d'ordonnancement adaptée, permet de rendre l'ordonnancement prioritaire au sein du système, quelque soit la charge des nœuds de la grappe. En revanche, cette approche va à l'encontre du concept initial des micro-noyaux où les services systèmes doivent être mis en œuvre dans l'espace utilisateur.

D'autres travaux ont été menés pour permettre de modifier le comportement de l'ordonnanceur mise en œuvre dans le système. Dans CHARLOTTE, il est possible de fournir des consignes à l'ordonnanceur global. Cela permet non seulement de modifier le comportement de l'ordonnanceur global au sein de la grappe mais également d'activer/désactiver l'ordonnancement automatique.

3.5 Résumé

La littérature abonde de travaux effectués sur l'ordonnancement de tâches. Ces travaux peuvent être classés suivant deux axes : (i) en fonction de leur type de partage (partage temporel ou partage spatial) et (ii) en fonction de leur aspect préemptif (ordonnanceurs statiques et dynamiques). Ces deux axes sont orthogonaux mais permettent de caractériser une politique d'ordonnancement. L'orthogonalité de ces deux axes rend la mise en œuvre d'une politique d'ordonnancement particulièrement difficile si le domaine d'application (*i.e.* le type d'application à exécuter) n'est pas limité.

Le type d'application considéré induit des contraintes sur la politique d'ordonnancement pour obtenir une exécution correcte. Ainsi, si une politique de « *co-scheduling* » est une solution intéressante pour l'ordonnancement de tâches parallèles communiquant par échanges de message, ce type de politique n'est pas du tout adapté à l'ordonnancement d'applications parallèles partageant de la mémoire car ne minimise pas les communications entre threads. On voit donc que le type des applications crée de fortes contraintes sur la politique d'ordonnancement à utiliser et que ces contraintes sont parfois contradictoires.

Il n'existe donc pas de politique d'ordonnancement adaptée à tout type d'utilisation d'une grappe. La notion d'ordonnanceur adaptable est particulièrement intéressante. On peut ainsi mettre en œuvre un simple système de batch, un ordonnanceur dynamique ou encore une politique de « *co-scheduling* », en fonction des applications à exécuter.

4 MÉCANISMES DE GESTION GLOBALE DES PROCESSUS

4.1 Introduction

Quelque soit la politique d'ordonnancement, l'ordonnanceur global doit pouvoir gérer l'exécution d'une tâche au sein de la grappe. La gestion de l'exécution d'une application nécessite dans un premier temps de la déployer au sein de la grappe. Pour cela, l'ordonnanceur doit disposer de mécanismes de création distante et de duplication de processus. L'ordonnanceur global peut mettre en œuvre une politique d'ordonnancement dynamique. Dans ce cas, l'ordonnanceur global doit également être capable de déplacer les processus en cours d'exécution et doit donc disposer d'un mécanisme de migration de processus.. Enfin, si l'ordonnanceur est amené à suspendre puis reprendre l'exécution d'une tâche. Il doit disposer d'un mécanisme de création et de restauration de point de reprise.

De nombreux travaux ont été menés sur ces différents mécanismes et notamment sur la migration de processus. L'un des premiers systèmes à offrir un mécanisme de migration de processus fût DEMOSMP [72] alors que les premiers travaux importants sur la création/restauration de point de reprise d'applications ont été menés dès 1975 [73]. De nombreuses études ont également été menées dans le domaine des systèmes distribués. Par exemple, le système CONDOR [53, 15] pour les architectures de type NOW propose des mécanismes de migration et de création/restauration de processus. De nombreux travaux sur la migration ont également été menés dans le domaine des systèmes distribués intégrés (*e.g.* le système SPRITE qui offre des mécanismes de migration) ou encore dans le domaine des systèmes de mémoire virtuelle partagée pour threads (*e.g.* le système DSM-PM^2 [4] permettant la migration de threads). Néanmoins, à ce jour, il n'existe pas de solution complétement satisfaisante pour les systèmes LINUX alors que l'ensemble des mécanismes utilise un concept commun, la manipulation d'image d'un processus permettant de créer un processus clone.

Nous nous intéressons particulièrement aux travaux menés autour du système LINUX et plus précisément autour des grappes de calculateurs LINUX constituées de nœuds homogènes (une application est représentée par un seul code objet). Nos travaux s'inscrivent dans un contexte LINUX et motivent donc notre étude approndie des solutions disponibles pour ce système d'exploitation.

Le chapitre 4.2 présente un mécanisme de base pour l'exécution d'application sur grappe : le mécanisme de déploiement d'applications. Nous montrons que les mécanismes

de déploiement regroupent un mécanisme de création distante de processus et un mécanisme de duplication de processus. Le chapitre 4.3 présente les travaux effectués sur la migration de processus, mécanisme indispensable pour la mise en œuvre d'un ordonnanceur dynamique. Nous verrons que la migration de processus peut se décomposer en plusieurs phases : la gestion de l'identifiant de processus au sein de la grappe, l'extraction d'un processus d'un système où le processus s'exécute, le transfert du processus vers un nœud distant et enfin la reprise d'exécution du processus sur le nœud distant. Enfin, nous montrons dans le paragraphe 4.4 les différentes approches pour mettre en œuvre un mécanisme de création et de restauration de points de reprise de processus. Nous verrons aussi que ces mécanismes de création et de restauration de points de reprise de processus reposent sur le même principe que la migration ou la duplication de processus. En effet, tous ces mécanismes nécessitent de manipuler des images de processus.

4.2 Déploiement d'applications sur grappe

Les ordonnanceurs statiques les plus simples comme les systèmes de batch nécessitent de pouvoir créer des processus à distance pour déployer les applications dans une grappe.

Deux types de mécanismes peuvent être utilisés pour placer les processus/threads au sein de la grappe : (i) la création distante de processus, (ii) la duplication de processus permettant d'effectuer une création de type « *fork* ».

4.2.1 Création distante de processus

Définition 18 (Création distante de processus) *Nous appelons création distante de processus la création d'un processus sur un nœud distant sans clonage d'un processus existant.*

La création distante permet d'éviter le coût de la migration, les informations nécessaires à la création d'un processus sans duplication d'un processus existant étant beaucoup moins importantes que l'image d'un processus (le lancement d'un processus ne nécessite qu'une référence sur le fichier objet du programme). De nombreux outils dans l'espace utilisateur ont d'ailleurs été mis en œuvre.

C'est notamment le cas des outils pour grappe KA-TOOLS [58] (faisant partie de la distribution CLIC [29] - Cluster LInux pour le Calcul - pour grappe) et des outils C3 [56] (« *Cluster Command and Control* ») faisant partie de l'ensemble d'outils pour grappe OSCAR [30] (*Open Source Cluster Application Resources*). Ces outils permettent de lancer des commandes sur les nœuds de la grappe de façon transparente pour l'utilisateur et sont tous les deux fondés sur les outils *rsh* ou *ssh*. Il est alors simple de lancer de nouvelles applications au sein de la grappe mais également d'administrer les grappes (ces outils permettant de lancer une commande au sein de la grappe ont permis de mettre en œuvre un ensemble d'outils d'administration des grappes, et notamment pour l'installation des systèmes d'exploitation sur l'ensemble des nœuds). Mais le placement ne permet pas de faire face à tous les problèmes d'ordonnancement pouvant apparaître au sein de la grappe. Par

exemple, le placement ne permet pas de garantir un service de haute disponibilité au sein du système (*e.g.* il est impossible de déplacer les processus d'une application s'exécutant sur un nœud lorsque l'on souhaite arrêter celui-ci).

Les environnements de programmation parallèle tels que MPI et PVM permettent de déployer une application parallèle de manière transparente au programmeur. Ces outils placent généralement les processus des applications parallèles de manière simple, sans se soucier d'un éventuel partage temporel des ressources. Les processus sont placés uniformément sur les différents nœuds généralement sans tenir compte de l'état des nœuds. Pour cela, le programmeur fournit uniquement la liste des nœuds qu'il souhaite utiliser et les processus sont déployés automatiquement au lancement de l'application (généralement en utilisant des outils tels que *rsh* ou *ssh*).

4.2.2 Duplication de processus

La duplication qui est une méthode pour déployer une application multi-processus a le désavantage d'être aussi coûteuse que la migration d'un processus : toute l'image du processus père doit être transférée sur le nœud distant où l'on souhaite créer le nouveau processus. Le système GENESIS offre un mécanisme de duplication de processus qui peut également s'appliquer à des groupes de processus afin de simplifier le déploiement des applications parallèles.

4.2.3 Gestion de groupes de processus

Afin de gagner en souplesse d'utilisation et en efficacité, les mécanismes de déploiement de processus peuvent être étendus à la manipulation de groupes de processus. Cela permet par exemple de manipuler simplement des applications parallèles. C'est notamment le cas dans le système GENESIS [40]. GENESIS est un système SSI fondé sur un micro-noyau spécialement développé pour les grappes de calculateurs dont l'ordonnanceur global est fondé sur des mécanismes de manipulation de processus efficaces (*i.e.* la création distante, la migration et la création de points de reprise de processus), mais également un mécanisme de gestion de groupes de processus avancé. Il est ainsi possible de définir des groupes de processus (*e.g.* les processus d'une même application parallèle) et d'appliquer un mécanisme de gestion globale de processus sur l'ensemble de ce groupe.

4.3 Migration de processus

La migration de processus peut être utilisée par un ordonnanceur dynamique pour :
- la répartition dynamique de charge,
- l'administration système. Pendant l'exécution des applications, l'administrateur système peut retirer des nœuds de la grappe. Dans ce cas, le système doit être capable de déplacer les processus en cours d'exécution sur des nœuds distants.
- l'accès aux données. Si les données nécessaires à un processus se trouvent sur un autre nœud, le rapprochement du processus de ses données peut permettre un gain

significatif de performance.

Un processus est composé de deux types de données : les données dites privées et les données dites publiques. Les données privées d'un processus sont constituées de données qui sont propres à son exécution, et rassemblent donc principalement :
- l'état des registres processeur propre à l'exécution du processus,
- la liste des fichiers ouverts par le processus,
- l'identifiant du processus.

Les données publiques sont les données qui peuvent être partagées avec d'autres processus au sein du système. Ces données publiques regroupent principalement l'espace d'adressage du processus. Pour déplacer un processus, il faut donc : (i) extraire les informations privées et publiques du processus, (ii) transférer ces informations, (iii) traiter sur le nœud distant ces informations pour redémarrer le processus. La mise en place d'un mécanisme de migration de processus nécessite de tenir compte de la problématique de l'identification unique de processus au sein de la grappe.

Le chapitre 4.3.1 présente la problématique de la gestion d'identificateur de processus au sein de la grappe lié à la migration de processus. La chapitre 4.3.2 expose les différentes approches pour extraire un processus du système en vue de le déplacer vers un nœud distant, le chapitre 4.3.3 détaille les méthodes de transfert de processus et le chapitre 4.3.4 présente l'exécution d'un processus sur le nœud distant.

4.3.1 Gestion des identificateurs de processus

Le déplacement de processus pendant leur exécution nécessite de pouvoir identifier un processus quelque soit sa location au sein de la grappe. Sans mécanisme d'identification unique d'un processus au sein d'une grappe, il est impossible à un processus de communiquer avec celui-ci via un mécanisme de communication entres processus locaux (« *Inter Process Communications* » - IPC).

Pour résoudre une partie du problème de localisation des processus au sein d'une grappe, le système ZAP [67] (fondé sur CRAK [98]) propose la notion de POD (« *PrOcess Domain* »). Un POD est une abstraction du système d'exploitation pour un groupe de processus (une application par exemple) permettant de regrouper des processus au sein d'un espace de nommage. Cet espace de nommage permet ensuite de résoudre des appels système indépendamment de la localisation des processus. Cette abstraction du système d'exploitation est effectuée par l'interception de tous les appels système se rapportant à un processus ou à groupe de processus. Ainsi, il est possible de rediriger l'ensemble de ces appels système vers les processus ciblés, quelquesoit leur localisation au sein de la grappe. Cette approche permet donc d'offrir une vision du système assez proche d'un système à image unique où les applications s'exécutent de façon transparente à la localisation des processus au sein de la grappe, le but dans les deux cas étant d'offrir une vision unique d'une partie ou de l'ensemble des ressources, de donner l'illusion aux applications en cours d'exécution que le système est un SMP virtuel. En contrepartie, il est nécessaire de modifier l'ensemble des appels systèmes manipulant les identifiants de processus afin de les rediriger vers des fonctions propres à la gestion des POD.

Une autre approche pour résoudre le problème de localisation des processus est de limiter les actions des utilisateurs à un seul nœud de la grappe, appelé frontal. Le frontal sert donc à lancer et à suivre l'exécution les applications. Lorsqu'un processus est créé, un nouveau processus virtuel est créé sur le frontal. Un processus virtuel consiste à créer une entrée dans la table des processus s'exécutant sur le nœud et de mettre en place des mécanismes d'interception des services systèmes se rapportant à la localisation des processus (*e.g.* gestion des signaux). Ainsi, si un service système se rapportant à la localisation des processus est appelé, le système BPROC [44] récupère les informations demandées sur le nœud distant.

4.3.2 Extraction du processus

Les données d'un processus au sein d'un système sont constituées principalement des valeurs de registres associés à l'exécution du processus, de la liste des fichiers ouverts et de son espace d'adressage. Toutes ces informations doivent être extraites du système local pour pouvoir créer un processus clone sur un nœud distant dans le cadre d'une migration de processus.

4.3.2.1 Extraction des valeurs des registres

L'extraction d'un processus est un problème dépendant du système sous-jacent. Par exemple, dans un système LINUX pour architecture x86, les valeurs des registres ne sont accessibles que lorsque le processus est non actif à la suite d'un appel système, d'une exception ou d'une interruption, où le noyau place les valeurs de registres en tête de pile du processus. L'extraction des valeurs des registres est donc fortement liée à l'état du processus.

Deux approches sont donc possibles : (i) modifier le noyau pour créer un nouvel état spécialisé pour l'extraction d'un processus où les valeurs des registres sont disponibles ou (ii) utiliser un état du système existant où les valeurs des registres sont disponibles (*e.g.* après avoir envoyé un signal d'arrêt au processus). L'utilisation d'un état existant pour récupérer les valeurs des registres condamne cet état pour la seule mise en œuvre du mécanisme de récupération des valeurs des registres. Si un autre système tente d'utiliser cet état, une incompatibilité est inévitable et les deux systèmes risquent de ne plus fonctionner correctement. Par exemple, les systèmes EPCKPT [71] et MIYUNGA [66] utilisent le traitement du signal *UNUSED* pour récupérer les valeurs des registres. Le noyau n'est pas modifié mais si un application utilise ce signal pour fonctionner, son exécution devient impossible.

Une autre approche consiste à modifier le noyau pour créer un nouvel état dans lequel les valeurs des registres sont disponibles. Dans ce cas, le nouvel état est spécifique au mécanisme d'extraction des registres et ne génère donc pas d'incompatibilité avec d'autres applications. En revanche, la modification du noyau crée une forte dépendance au noyau sous-jacent. Pour chaque nouvelle version du noyau, un travail de portage est nécessaire. Les administrateurs sont également obligés de déployer le noyau modifié sur l'ensemble des

nœuds de la grappe pour que le mécanisme soit exploitable.

4.3.2.2 Extraction de l'espace d'adressage

L'espace d'adressage d'un processus est composé de segments mémoire, chaque segment mémoire regroupant des pages mémoire. L'extraction de l'espace d'adressage se décompose en deux phases : l'extraction des informations propres aux segments mémoire et l'extraction des pages mémoire. Suivant les techniques de transfert présentées dans le paragraphe 4.3.3, toutes les pages mémoires ne sont pas nécessairement extraite pendant la migration du processus.

Pour les applications parallèles communiquant par partage de données, la mémoire répartie partagée (MPR) peut servir de mécanisme de migration des pages mémoire. Dans ce cas, seules les informations sur les segments mémoire sont à extraire afin de pouvoir créer sur le nœud distant un espace d'adressage identique à celui sur le nœud initial.

4.3.2.3 Extraction des fichiers manipulés par le processus

Lorsqu'un processus accède à un fichier, un descripteur est alloué au processus et le système local alloue des données système permettant de réaliser l'accès physique au fichier. L'accès à ces fichiers doit être garanti dans le cas d'une migration.

Les fichiers doivent donc être accessibles par l'ensemble des nœuds de la grappe. Un système de fichier distribué tel que NFS [76] permet à tous les nœuds d'accéder à un fichier. Dans le système SPRITE, un serveur central contient l'ensemble des fichiers partagés, et garantit une cohérence des fichiers en cas d'accès concurrents.

L'accessibilité des fichiers ne suffit pas car les descripteurs de fichiers alloués au processus par un système ne sont *a priori* plus valides sur un autre nœud. De même, lorsqu'un processus est déplacé sur un nœud, ce nœud ne dispose pas *a priori* des informations système permettant d'accéder physiquement au fichier. Une solution est de procéder à une *réouverture* des fichiers une fois le processus transféré sur un nœud distant. Pour cela, il est nécessaire d'extraire les informations permettant d'identifier les fichiers ouverts par le processus.

4.3.3 Transfert du processus

Quand une image du processus est extraite du système local, cette image doit être transférée vers le nœud distant. Si l'espace d'adressage du processus est de grande taille, le transfert peut prendre beaucoup de temps, pénalisant ainsi l'exécution du processus mais également les systèmes des nœuds émetteurs et récepteurs si ceux-ci sont bloqués pendant le transfert. De plus, le coût du transfert du processus pendant une migration n'est pas le seul coût à prendre en compte. Deux coûts sont donc à prendre en compte lors de la migration : (i) le coût du transfert du processus, (ii) le surcoût d'exécution après la migration. Pour réduire ces coûts, différentes techniques de transfert sont possibles.

4.3.3.1 Migration de l'espace d'adressage complet

Toutes les données privées et publiques du processus sont envoyées lors de la migration. Cette stratégie peut également être utilisée pour la création de points de reprise d'un processus.

Toutes les données du processus étant transférées en une seule fois, la taille des données envoyées pendant la migration dépend donc de la taille de l'espace d'adressage : plus l'espace d'adressage d'un processus est important et plus la migration est coûteuse.

En revanche, cette approche permet de garantir un surcoût d'exécution faible : les pages mémoire utilisées par le processus lors de son fonctionnement sont présentes localement.

Cette approche est, par exemple, mise en œuvre dans CONDOR [53].

4.3.3.2 Transfert des pages modifiées

La technique du transfert des pages modifiées n'est possible que si un mécanisme de pagination à distance est disponible. C'est une simple variante de la technique précédente : seules les pages accédées sont transmises durant la migration. Les pages non accédées sont déplacées seulement lorsque le processus tente d'y accéder depuis un nœud distant.

Cette technique permet de ne pas transférer tout l'espace d'adressage du processus durant la migration. Cette technique est donc intéressante si peu de pages non accédées sont paginées après la migration.

En contre-partie de la diminution du coût de la migration, la migration induit un surcoût d'exécution : chaque page non accédée avant la migration doit être migrée vers le nœud d'exécution courant du processus lorsque le processus tente d'y accéder. Aussi, pour chacune de ces pages, un surcoût est introduit.

Cette stratégie a été mise en place dans LOCUS [91], dans MOSIX [13], ou encore dans MIYUNGA [66] (projet de migration de processus au sein du projet I-CLUSTER [74]).

4.3.3.3 Transfert sur accès à la demande

La technique de transfert sur accès à la demande est proche de la technique précédente : les pages ne sont déplacées que lors de leur accès ; lors de l'exécution du processus après migration.

Cette technique est celle qui permet d'avoir le coût de transfert le plus faible : une très faible partie de l'espace d'adressage est migré, les pages mémoire n'étant migrées que lors de leur accès.

Toutes les pages mémoires référencées par le processus migré devant être déplacées à la demande, un important surcoût d'exécution peut apparaître.

Le système ACCENT[97] met en œuvre cette approche.

4.3.3.4 Transfert par précopie

La technique de transfert par précopie vise à réduire le temps d'inactivité d'un processus introduit par une migration (temps durant lequel un processus ne s'exécute ni sur le nœud

source, ni sur le nœud destination) : les pages référencées par un processus sont précopiées sur un nœud distant (si ces pages ne sont pas trop nombreuses). Ainsi, si le processus est déplacé, les pages mémoire seront déjà présentes localement sur le nœud cible.

Le coût de migration est dans ce cas très réduit : les pages mémoire déjà copiées lors de leur accès et non modifiées depuis ne doivent pas être transférées.

Le surcoût d'exécution est faible : les pages mémoire précopiées ne nécessitent pas de transfert après la migration. En revanche, le coût de la précopie n'est pas négligeable et provoque une charge au sein des systèmes de deux nœuds concernés par une migration.

Cette approche est notamment mise en œuvre dans le système V[86]

4.3.3.5 Enregistrement des pages modifiées sur disque

La technique d'enregistrement des pages modifiées sur disque est une variante de la technique de transfert des pages modifiées : les pages modifiées sont sauvegardées sur disque, via un système de fichier distribué. Une fois la tâche déplacée les pages sont chargées depuis le disque lors de leur accès, à la demande. Cette technique est donc similaire au transfert des pages modifiées pour le nœud source, tandis qu'elle est similaire à la technique de transfert sur accès pour le nœud distant. Néanmoins, la dépendance est dans ce cas vis-à-vis d'un serveur de fichier et non plus du nœud source.

Le coût de transfert est ici modéré : l'espace d'adressage n'est pas transféré dans sa totalité. En revanche, le stockage des pages mémoire modifiées étant effectué sur un serveur de fichier, le transfert de ces pages peut être plus coûteux que la migration des pages vers un nœud distant car lors de leur accès, les pages mémoire doivent être transmises à la demande du serveur de fichier vers le nœud d'exécution du processus. Un surcoût d'exécution est donc introduit pour chaque page accédée.

4.3.3.6 Autres variantes

D'autres techniques, variantes des techniques présentées précédemment, ont été mises en œuvre. Ces techniques ont principalement pour but de réduire le temps d'inactivité d'un processus lors d'une migration. C'est notamment le cas pour les travaux effectués au sein du système d'exploitation Choices [23] qui permet de redémarrer le processus sur le nœud cible parallèlement au transfert du processus. Seules les données indispensables à l'exécution du processus sont initialement transmises, les autres étant transmises en parallèle par la suite.

4.3.4 Exécution du processus sur le nœud de destination

L'exécution d'un processus sur un nœud après migration peut provoquer des accès à des ressources distante générant à chaque fois un coût. En effet, un certain nombre d'appels système ne peuvent pas être exécutés localement car les ressources nécessaires ne sont pas disponibles localement. Les principaux services système concernés par ce phénomène sont les accès à l'espace d'adressage, les accès aux fichiers, les communications IPC et les accès réseau.

Bien entendu, les problèmes d'accès à une ressource distante dépendent du type de migration mis en œuvre. Si la migration génère des dépendances résiduelles, les coûts d'accès à une ressource distante sont importants. Par exemple, nous avons vu avec les différentes techniques de transfert de l'espace d'adressage que si l'espace d'adressage est déplacé complètement lors de la migration, aucun déplacement de pages n'est nécessaire par la suite lors de l'exécution du processus sur le nœud distant. Inversement, si l'espace d'adressage n'est pas déplacé lors de la migration mais à la demande, le système doit aller chercher chaque page sur un nœud distant lors de leur accès.

Un certain nombre de services ne peuvent pas être résolu localement après une migration, même si toutes les ressources de la grappe sont gérés globalement. Lorsqu'un processus migre, les liens entre le processus père, ses frères et ses fils ne peuvent pas être maintenus. Aussi, la communication entre deux processus par IPC (comme les signaux) dont l'un des deux a migré est impossible sans gestion globale des IPC. Par exemple, dans la plupart des systèmes, un processus envoie un signal à son père à sa terminaison. Après une migration, le processus se terminant n'est plus sur le nœud de son processus père et ne peut donc pas lui envoyer directement le signal de terminaison. Le processus père risque donc de rester dans l'attente de la terminaison de son processus fils alors que celui-ci est terminé. Il est donc indispensable de disposer de mécanismes permettant d'accéder à une ressource système distante (*e.g.* un processus).

Comme un accès à une ressource système distante est plus coûteux que l'accès à une ressource système local de même type, le coût du transfert du processus n'est donc pas l'unique coût impliqué par la migration de processus, un coût peut apparaître lors de l'exécution du processus après la migration. L'exécution d'applications parallèles communiquant par messages peut donc générer un très grand nombre de requêtes vers un nœud distant si l'envoi de message nécessite de passer par le nœud qui accueillait le processus avant sa migration. Pour les applications parallèles communiquant par partage de données, les accès à l'espace d'adressage peuvent être très nombreux et s'il est nécessaire de faire une requête vers un nœud distant pour tous ces accès, un important surcoût est généré. De même, si les accès fichier non résolvables localement sont trop nombreux, l'application se trouve fortement pénalisée.

Trois approches sont possibles pour résoudre une requête d'accès à une ressource système :

1. envoyer la requête vers le nœud pouvant résoudre l'accès à la ressource système pour que celui-ci réponde à la requête,
2. soumettre la requête à un système distribué qui va gérer la résolution de la requête,
3. demander une copie de la ressource système pour ensuite résoudre localement la requête.

L'envoi des requêtes vers le nœud possédant la ressource système à laquelle on souhaite accéder est systématique dans les systèmes où une dépendance est maintenue avec le nœud source dans une migration. Cette dépendance est créée lorsque des mécanismes de gestion globale des ressources ne sont pas disponibles. Cette approche est très pénalisante pour les processus effectuant plusieurs migrations lors de leur exécution (pour chaque migration,

une nouvelle dépendance apparaît), ainsi que pour les processus communicants (tous les accès réseau sont redirigés vers le nœud initial, les flux de donnée ne pouvant pas être gérés indépendamment de la localisation des processus). Par exemple, le mécanisme de migration du système MOSIX crée des dépendances entre les nœuds : lorsqu'un processus est déplacé, un lien est conservé avec son nœud initial afin de pouvoir résoudre les appels système lors de l'exécution du processus après migration. Pour réduire les cas de dépendance, MOSIX peut utiliser un système de fichier distribué appelé « *Mosix Direct File System Access* »[2]. MOSIX s'avère donc être un système efficace pour gérer les applications séquentielles de calcul, mais peu efficace pour les autres types d'application comme par exemple les applications parallèles communiquant par messages. Le système MOSIX n'offrant pas de MPR, il ne peut pas exécuter des applications parallèles communiquant par partage de données.

L'utilisation d'un système distribué est une approche intéressante pour limiter les dépendances récursives induites par des migrations successives. Lorsqu'un processus tente d'accéder à une ressource système distante, la requête est envoyée au service distribué qui se charge de trouver la localisation de la ressource système et de résoudre la requête.

La demande d'une copie de la ressource système est mise en œuvre pour la migration de l'espace d'adressage à la demande (voir paragraphe 4.3.3.3). Nous avons vu que cette approche, principalement applicable à l'espace d'adressage, permet de limiter le temps d'inactivité du processus et du système.

4.4 Création et restauration de points de reprise

La création et la restauration de points de reprise peuvent avoir deux applications : (i) la migration de processus, (ii) la mise en œuvre de mécanismes de tolérance aux fautes.

Les premiers mécanismes de migration de processus ont été mis en œuvre en utilisant un mécanisme sous-jacent de création/restauration de points de reprise (notamment dans de nombreux systèmes pour NOW). C'est notamment le cas dans les premières versions de SPRITE et dans NOMAD [70], fondées sur un système de fichier distribué, où lors d'une migration, l'image du processus est copiée sur le système de fichier pour être restaurée sur un nœud distant. Le système GATOSTAR [36] propose également un mécanisme de migration de processus fondé sur la création/restauration d'un point de reprise. La particularité de GATOSTAR est d'unifier les mécanismes de placement de tâche avec le mécanisme de création de points de reprise, réduisant ainsi les redondances de mécanismes par rapport à l'utilisation de deux systèmes indépendants.

Mais la création de points de reprise est surtout un mécanisme important pour tolérer la défaillance d'un nœud au cours de l'exécution d'une application. Les mécanismes de migration et de création/restauration de point de reprise sont proches. Les principales différences entre ces deux mécanismes sont que pour la migration, le processus initial est stoppé ou détruit, alors que pour la création de points de reprise pour la tolérance aux défaillances, le processus initial continue à s'exécuter. En revanche, pour offrir un mécanisme de tolérance aux défaillances, le point de reprise doit être stocké sur un support

de stockage fiable.

La création d'un point de reprise d'une application séquentielle est très similaire à la migration. Comme pour la migration, il est nécessaire d'extraire l'image du processus. La principale différence entre la migration d'une application séquentielle et la création de point de reprise est que, dans le cas de la création de point de reprise, l'image du processus (et donc de l'application) est stockée en mémoire ou sur disque alors que pour la migration, elle est envoyée sur le réseau.

La création de point de reprise pour les applications parallèles (par mémoire partagée ou par échange de messages) diffère de la migration. En effet, lors de la création de points de reprise du processus ou d'un thread d'une application parallèle, il est nécessaire de prendre en compte la cohérence de l'application dans son ensemble, contrairement à la migration. Par exemple, la migration d'un thread ne nécessite pas de prendre garde à l'état global de la mémoire partagée, les pages mémoire pouvant être migrées à la demande d'un nœud vers un autre lors des accès. En revanche, dans le cas de la création d'un point de reprise d'un thread, il est nécessaire de veiller à la cohérence globale de la mémoire partagée afin de garantir que celle-ci soit dans un état cohérent à travers l'ensemble des points de reprise des threads[25]. Cette image globale pouvant servir de réferenciel pour reprendre l'exécution d'une application est appelée ligne de recouvrement.

Définition 19 (Ligne de recouvrement) *Une ligne de recouvrement est le dernier état global cohérent connu à partir duquel le système pourra redémarrer en cas de défaillance.*

La littérature décrit de nombreuses stratégies de création de point de reprise pour applications parallèles afin de garantir une cohérence globale, aussi bien pour les applications parallèles par passage de message[33] que les applications parallèles par mémoire partagée[60]. On peut isoler principalement trois types de stratégies de création de point de reprise : (i) création de point de reprise coordonnée, (ii) création de point de reprise non-coordonnée et (iii) création de point de reprise induite par les communications.

Dans le cas de la création de point de reprise coordonnée, les processus d'une tâche se coordonnent pour créer un point de reprise afin de garantir l'existence d'une ligne de recouvrement. Cette approche permet d'avoir une phase de reprise simple fondée sur le dernier point de reprise de chaque processus et de conserver un seul point de reprise. En revanche, la coordination lors de la création du point de reprise induit un coût important dû à la synchronisation des processus.

Dans le cas de création de point de reprise non-coordonnée, la création des points de reprise est simple et ne nécessite aucune coordination, garantissant un coût faible sur le fonctionnement normal. En revanche, la reprise en cas de défaillance nécessite le calcul d'une ligne de recouvrement en fonction des différents points de reprise disponibles. Le calcul de la ligne de recouvrement est complexe et peut être coûteuse et nécessite une numérotation des points de reprise ainsi que la maintenance d'un historique des interactions entre processus. Lors du calcul de la ligne de recouvrement, un *effet domino* peut apparaître et conduire l'application à reprendre son exécution de son état initial. De plus, cette stratégie nécessite de conserver plusieurs points de reprise par processus.

Enfin, l'approche de création de point de reprise induite par les communications est une approche intermédiaire à la création de point de reprise coordonnée et non-coordonnée. Ici les points de reprise des processus sont créés indépendamment les uns des autres mais peuvent provoquer la création forcée de points de reprise si le processus dépend d'autres processus suite à des interactions. Le système assure la progression de la ligne de recouvrement évitant ainsi l'effet domino et une gestion complexe des points de recouvrement.

4.5 Niveau de mise en œuvre

Deux différentes approches sont possibles mettre en œuvre des mécanismes de gestion globale de processus : (i) approche utilisateur ou (ii) approche noyau.

La mise en œuvre de mécanismes de gestion globale de processus grâce à des bibliothèques en mode utilisateur permet de disposer d'un mécanisme simple à mettre en œuvre, évitant la complexité de la programmation noyau. Le mécanisme mis en œuvre est donc plus facilement maintenable et portable sur différents environnements. La mise en œuvre grâce à des bibliothèques en mode utilisateur permet également de supporter simplement les changements de version du système sous-jacent. En revanche, les bibliothèques en mode utilisateur n'ayant pas accès à toutes les informations noyau du système, ces bibliothèques ne peuvent pas gérer tous les types de processus (il est par exemple complexe de déplacer les informations concernant les communications réseaux depuis l'espace utilisateur). De plus, ce type d'approche nécessite généralement de lier l'application à la bibliothèque utilisée pour la mise en œuvre des mécanismes de manipulation des processus lors de la compilation (*e.g.* le système CONDOR pour son mécanisme de migration, même si le système PROCESSHIJACKING [96] propose les mêmes services que CONDOR sans la contrainte de compilation). En revanche, le principal intérêt de la mise en œuvre au niveau utilisateur concerne la migration de threads qui offre de très bonnes performances (quelques microsecondes).

L'approche noyau permet de lever les contraintes sur le type de processus pouvant être manipulé (toutes les informations noyau propres à un processus sont accessibles), de garantir une transparence aux applications et aux utilisateurs. En revanche, cette approche est difficile à mettre en œuvre, la programmation noyau étant particulièrement complexe et un travail de portage est nécessaire pour chaque nouvelle version du noyau. En fonction des buts à atteindre, le noyau peut être modifié ou non. Par exemple, le projet I-CLUSTER, dans lequel s'inscrit le projet MIYUNGA, vise à utiliser les cycles processeurs sur machines reliées à un réseau et non entièrement utilisée par son propriétaire. On comprend alors que la non-modification du noyau est un avantage certain puisqu'il n'est pas nécessaire d'installer de nouveaux noyaux sur les machines visées, simplifiant l'administration des machines. En revanche, le système MOSIX vise à offrir un système spécialisé pour l'exécution d'applications scientifiques séquentielles. Cette spécialisation n'est possible qu'en modifiant le noyau puisque la recherche de performance nécessite d'agir à des endroits du noyau qui ne sont pas initialement prévus pour accueillir l'exécution d'un code externe.

4.6 Résumé

Aujourd'hui plusieurs solutions pour grappes sous système LINUX existent pour offrir des mécanismes de création distante, de migration ou de création/restauration de points de reprise de processus. Par exemple, le système MIYUNGA permet de créer des points de reprise d'applications séquentielles et parallèles par échanges de messages ; le système MOSIX permet de déplacer des processus séquentiels ou parallèles communicant par échange de messages. La mise en œuvre de ces systèmes est effectuée par modification du système d'exploitation. L'intérêt de l'approche noyau est qu'elle impose moins de limitations sur les applications qui peuvent bénéficier des mécanismes mis en œuvre, notamment pour ce qui concerne la migration. Le système CONDOR, mis en œuvre au niveau utilisateur, permet la création distante et la migration de processus. Néanmoins, à ce jour, aucun système n'offre un ensemble complet de mécanisme de création distante, de migration et de création/restauration de points de reprise pour les systèmes LINUX et pour tout type d'application. De plus, alors que tous ces mécanismes ont en commun la manipulation d'images de processus. Cette approche est d'autant plus intéressante que cela permet de simplifier la maintenance des différents mécanismes (fédération des travaux), les mécanismes de gestion globale de processus étant connus pour être difficiles à mettre en œuvre.

Deuxième partie

Gestion globale des processus dans le système Kerrighed

L'objectif des travaux de cette thèse est la conception et la mise en œuvre d'un système de gestion globale de processus dans une grappe. Mon travail s'inscrit dans le contexte de la conception et de la réalisation du système d'exploitation à image unique KERRIGHED pour l'exécution d'applications à haute performance sur grappe.

Le chapitre 5 décrit le système d'exploitation à image unique KERRIGHED qui constitue le contexte de mes travaux. Dans le chapitre 6, nous présentons la démarche que nous avons adoptée dans la conception de l'ordonnanceur global et des mécanismes sous-jacents dans le système KERRIGHED. Le chapitre 7 a trait à l'architecture de l'ordonnanceur global et le chapitre 8 est consacré aux mécanismes de gestion de processus. Enfin, le chapitre 9 présente des exemples de politiques d'ordonnancement global mises en œuvre dans le système KERRIGHED.

5 LE SYSTÈME D'EXPLOITATION KERRIGHED

5.1 Principes de conception

KERRIGHED est un système d'exploitation pour grappe qui vise à concilier haute performance, haute disponibilité et simplicité d'utilisation pour l'exécution d'applications scientifiques sur grappe[62, 63, 64]. L'objectif est que les différents types d'applications à haute performance, que ce soit des applications séquentielles ou parallèles, fondées sur le paradigme de mémoire partagée ou d'échange de messages, puissent être exécutées sur une grappe dans un contexte de multi-programmation en exploitant au mieux les ressources matérielles disponibles (mémoire, disque, processeur, réseau).

Le système d'exploitation KERRIGHED est constitué d'un ensemble de services distribués qui permettent d'assurer un accès transparent aux ressources et leur partage entre les applications. Sous l'angle du système d'exploitation, il s'agit de gérer globalement à l'échelle de la grappe les blocs de données, les flux de données et les processus. Notre travail a porté sur la gestion globale des processus, la gestion globale de la mémoire ayant fait l'objet des travaux de thèse de Renaud Lottiaux[54] et celle des flux de données étant étudiée dans le cadre de la thèse de Pascal Gallard[39]. Pour fédérer l'ensemble des ressources de la grappe et offrir de bonnes performances aux applications, KERRIGHED est mis en œuvre au niveau noyau. Le prototype de KERRIGHED étend le noyau Linux qui, comme tout système d'exploitation moderne, peut être découpé grossièrement en deux parties : (i) les services systèmes de haut niveau et (ii) les gestionnaires de périphériques (voir figure 5.1). Les services système offrent une virtualisation des ressources physiques (mémoire, disque,

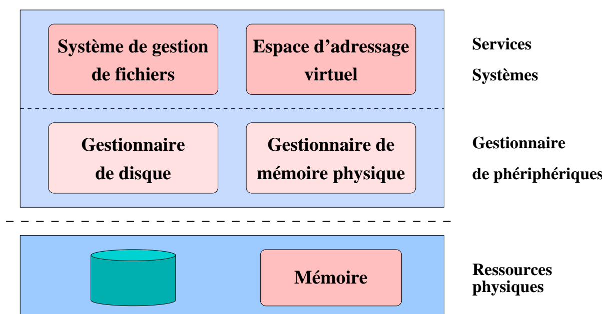

FIG. 5.1 – Architecture d'un système d'exploitation

processeur) en fournissant un ensemble de ressources logiques (mémoire virtuelle, fichiers,

processus) plus faciles à appréhender et à manipuler pour l'utilisateur. Les gestionnaires de périphériques gèrent les ressources à très bas niveau (mot mémoire, secteur de disque dur) en fournissant des abstractions logicielles (page mémoire, bloc disque logique) aux services systèmes.

Dans une grappe, chaque nœud dispose de son propre système d'exploitation composé de ces deux niveaux d'abstraction. Si l'on désire garder la même vision d'une machine unique au dessus d'une grappe, nous ne devons pas modifier les services systèmes de haut niveau. Cependant, ces services doivent pouvoir utiliser et partager l'ensemble des ressources de la grappe. Pour atteindre cet objectif, les services distribués de KERRIGHED gérant les blocs et flux de données sont intercalés entre les services système et les gestionnaires de périphériques. De la sorte, les services système de haut niveau peuvent tirer profit de l'ensemble des ressources matérielles de la grappe gérées localement par les services de bas niveau.

KERRIGHED vise à fédérer quatre types de ressource de la grappe : les pages mémoire, les disques, les processus et les flux de données. Pour chacun d'eux, le système KERRIGHED intercepte les appels des services système classiques du système d'exploitation d'un nœud aux services de bas niveau pour les étendre à l'échelle de la grappe (*e.g.* les appels du système de mémoire virtuelle au gestionnaire mémoire sont interceptés pour mettre en œuvre une mémoire répartie partagée). Contrairement aux travaux antérieurs, le système KERRIGHED gère les différents types de ressources de manière intégrée. Plutôt que de multiplier les mécanismes, KERRIGHED factorise les composants logiciels. Cette approche permet d'obtenir un ensemble de sous-systèmes cohérents, plus faciles à mettre en œuvre et à maintenir.

Les mécanismes de gestion globale d'une ressource peuvent être utilisés par les mécanismes de gestion globale d'autres ressources. Il est ainsi possible de panacher les différents mécanismes de fédération de ressources pour obtenir des services et des mécanismes évolués et efficaces. Par exemple, il est possible d'associer le mécanisme de gestion globale de la ressource mémoire à celui de la gestion globale de la ressource processeur pour développer et mettre en œuvre des mécanismes de manipulation de processus/threads partageant de la mémoire au sein de la grappe.

5.2 Architecture générale du système Kerrighed

KERRIGHED comporte sept principaux sous-systèmes mettant en œuvre des mécanismes distribués pour offrir les propriétés de transparence à la distribution et de partage des ressources d'un système à image unique (voir figure 5.2).

Le système KERMEM met en œuvre le concept de conteneur pour la gestion globale des blocs de données en mémoire. Le système KERSYNC met en œuvre les mécanismes de synchronisation entre processus (*e.g.* barrières, verrous). Les systèmes KerProc et KerGhost résultent de nos travaux de thèse. Le premier met en œuvre la gestion globale des processus. Le second est relatif à la gestion de l'image représentant l'état d'un processus. Le système KERNET met en œuvre la gestion globale des flux de données. Ces services distribués

s'appuient sur le système KERCOM pour la communication par messages entre les nœuds de la grappe. Enfin, le système KERTOOLS est un ensemble d'outils de base utilisés par les autres sous-systèmes (*e.g.* gestion de table de hachage).

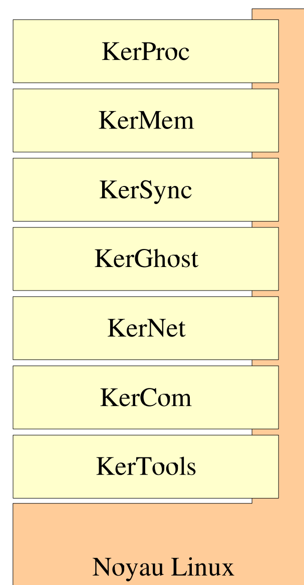

FIG. 5.2 – *Architecture générale du système* KERRIGHED

5.3 Gestion globale de la mémoire

Le sous-système KERMEM met en œuvre le système de gestion globale des blocs de données au sein de la grappe [54] fondé sur le concept de conteneur.

Un conteneur est un objet global permettant la gestion et le partage de pages entre les nœuds d'une grappe. Les conteneurs sont utilisés par les services de haut niveau, par l'intermédiaire de **lieurs** permettant de réaliser la connexion entre les conteneurs et le système d'exploitation des nœuds. À chaque conteneur est associé une paire de lieurs : un lieur de haut niveau appelé **lieur d'interface** permettant de détourner les fonctions d'accès aux périphériques vers le conteneur et un lieur de bas niveau appelé **lieur d'entrée/sortie** permettant au conteneur d'accéder à un gestionnaire de périphérique. Un conteneur donné ne peut être lié qu'à un seul gestionnaire de périphérique. En revanche, il peut être lié à plusieurs services système de haut niveau. Les lieurs permettent ainsi de réaliser le partage vertical des données à travers les conteneurs (voir figure 5.3).

Les lieurs sont utilisés afin de réaliser très simplement un grand nombre de services système distribués. Pour chaque service système distribué de haut niveau, une paire de lieurs différente est utilisée. Par exemple, une paire de lieurs permettant de connecter un conteneur entre le gestionnaire de mémoire virtuelle et le gestionnaire de mémoire physique permet de réaliser une mémoire virtuelle partagée. Une paire de lieurs permettant

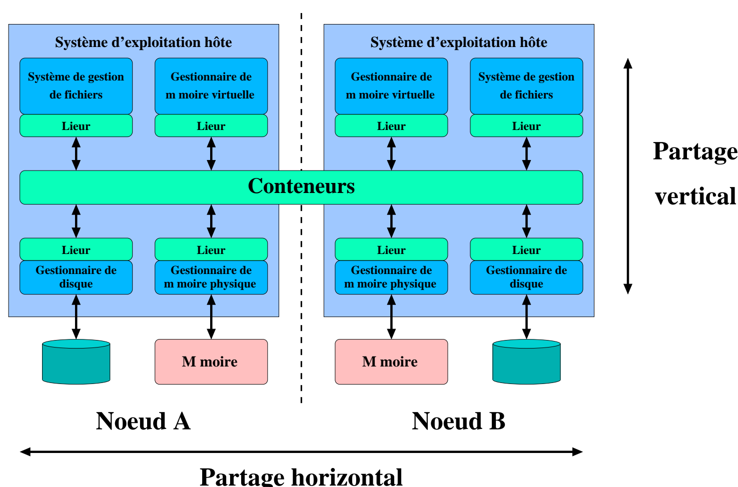

FIG. 5.3 – Les conteneurs Kerrighed

de connecter un conteneur entre le système de gestion de fichiers virtuel et le gestionnaire de disque permet de réaliser un système de gestion de fichiers distribué et un cache coopératif.

Un conteneur est un objet unique au sein de la grappe utilisable sur l'ensemble des nœuds. Pour cela, un conteneur est identifié par un identifiant unique, l'*identifiant de conteneur*. Un conteneur permet l'accès par des processus à des pages mémoire quelque soit la localisation des processus. Ces pages mémoire sont liées à un périphérique. Les conteneurs disposent donc d'un « *offset* » permettant de gérer le déplacement du conteneur dans le périphérique lié. L'identifiant unique et l'« *offset* » des conteneurs permettent à un processus d'*attacher* un segment mémoire à un conteneur. Dans ce cas, toutes les opérations de lecture/écriture au sein du segment mémoire sont interceptées par le mécanisme des conteneurs via le lieur.

Les mécanismes de lieurs et des conteneurs permettent de gérer globalement l'ensemble des données de type bloc. Ils ont permis de mettre en œuvre des mécanismes de partage de segments de mémoire virtuelle, de pagination en mémoire distante et un cache de fichiers coopératif.

5.4 Synchronisation distribuée de processus

L'exécution d'applications parallèles nécessite de disposer de mécanismes de barrière, de verrous, de « *wait condition* » et de sémaphores. Le système Kerrighed, à travers le sous-système KERSYNC propose tous ces mécanismes. Ils ont été mis en œuvre pour naturellement supporter la gestion globale de processus telle que la migration, la duplication ou encore la création de points reprise. Pour cela, le sous-système KERSYNC se fonde sur les mécanismes du module KERGHOST.

5.5 Gestion globale des flux de données

Le système KerNet offre un système de communication conçu pour permettre la migration efficace de processus communicants au sein d'une grappe de calculateurs. L'architecture de KerNet est suffisamment générique pour permettre la migration de tous les types de flux : socket, tube, périphérique de type caractère. Les processus communiquant par sockets IP ou Unix sont déplacés de manière transparente à l'aide du système KerNet, et les communications restent efficaces après la migration. Pour cela, KerNet propose le concept de flux dynamique sur lequel les interfaces standard de communication peuvent s'appuyer (voir figure 5.4). Les extrémités de ces flux sont appelés *sockets* KerNet et peu-

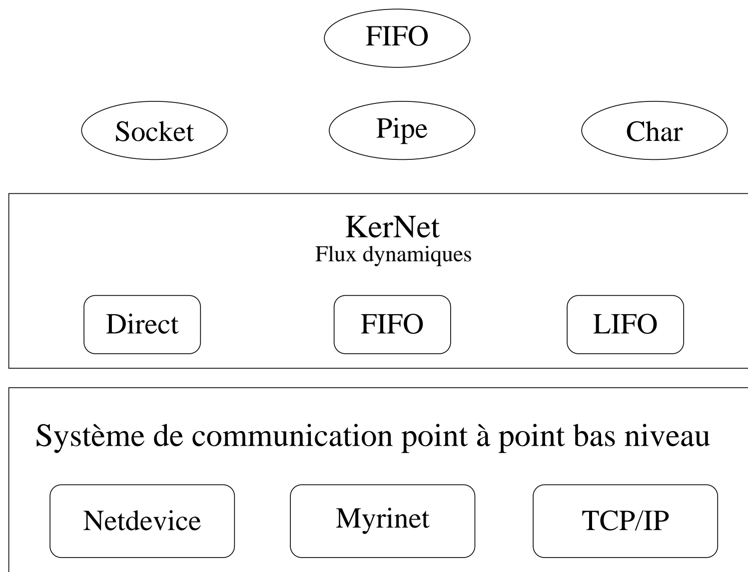

Fig. 5.4 – Architecture de KerNet

vent être déplacées au sein de la grappe. Les flux dynamiques et les sockets KerNet sont fondés sur le système KerCom. KerCom est fondé soit sur des pilotes de périphériques (*e.g.* Myrinet), soit sur l'interface réseau générique du noyau Linux (*i.e. netdevice*), soit sur un protocole de communication haut niveau (*e.g.* TCP/IP). Fondé sur cette couche de communication bas niveau, le système KerNet offre trois types de flux dynamiques : flux direct, flux FIFO et flux LIFO. Les flux dynamiques sont spécialisés par interface. Le système KerNet utilise ces flux dynamiques pour offrir des versions dynamiques des interfaces standard de flux Unix (*e.g.* inet/unix socket, pipe).

5.6 Gestion globale de données noyau

La fédération des ressources d'une grappe nécessite de pouvoir gérer globalement des données noyau afin de les partager au sein de la grappe. Cette gestion de données noyau,

indépendamment de leur localisation, est effectuée par le sous-système KerGhost qui fournit des primitives de base permettant de virtualiser les données noyau. Une fois virtualisées, les données noyau peuvent être envoyées à travers le réseau, ou encore stockées en mémoire ou sur disque. Ce mécanisme, fourni par le sous-système KERGHOST, est le fondement de mécanismes comme la migration ou la création de point de reprise de processus. Ce mécanisme a été développé dans le cadre des travaux de cette thèse et est présenté dans le paragraphe 8.1.

5.7 Gestion globale des processus

La gestion globale de processus au sein du système Kerrighed, mis en œuvre dans KERPROC, (principal travail de cette thèse) est effectuée grâce à la mise en œuvre d'un ordonnanceur global et de mécanismes de gestion globale de processus. Ces travaux sont présentés de manière détaillée dans les chapitres suivants.

5.8 Communications haute performance

La mise en œuvre des services distribués qui constituent le système KERRIGHED nécessite de pouvoir échanger des données entre les noyaux s'exécutant sur les différents nœuds de la grappe. Ceci implique l'utilisation d'un système de communication dans le noyau. Linux offre deux types de mécanismes de communication au sein du noyau : les « *sockets* » et les RPC. Or, ces deux mécanismes offrent des performances médiocres et une interface de programmation mal adaptée à la programmation de services système. Le système KERRIGHED a donc été doté d'un système de communication à haute performance offrant une interface de programmation simple, appelé KERCOM et qui répond aux objectifs suivants :

- **Haute performance** : le système de communication est conçu pour offrir des latences faibles et un débit élevé ;
- **Portabilité** : le système de communication fonctionne de la même manière et avec la même interface quelque soit la technologie réseau sous-jacente ;
- **fiabilité** : tout message envoyé est transmis sans aucune perte ou altération de données ;
- **Interface simple** : l'interface d'utilisation est simple et adaptée à la programmation distribuée.

5.9 Outils pour la création de services système distribués

Pour mettre en œuvre des mécanismes système distribués, des outils de base communs sont utilisés, comme la manipulation de tables de hachage, la manipulation de vecteurs de bit, ou encore des fonctions d'allocation de mémoire. Tous ces outils sont fournis par le sous-système KERTOOLS et mis à disposition des autres sous-systèmes du système d'exploitation KERRIGHED

5.10 Résumé

Kerrighed est une extension du noyau Linux visant à offrir un système à image unique pour grappe avec pour but d'offrir de hautes performances et une haute disponibilité pour l'exécution d'applications scientifiques (séquentielles et parallèles communiquant par message et par partage de mémoire). Pour cela, le système Kerrighed étend les services système afin de gérer globalement les pages mémoire, les fichiers, les processus et les flux de données. Les services distribués, permettant cette gestion globale, sont mis en œuvre par sept modules noyau et des modifications mineures du noyau.

Les travaux présentés dans ce document portent sur le conception et la réalisation du module KerProc, module mettant en œuvre la gestion globale des processus.

6 ORDONNANCEMENT GLOBAL ADAPTABLE

6.1 Objectifs

Historiquement, les laboratoires de recherche et développement tels que ceux d'EDF R&D utilisent des machines parallèles pour le développement et l'exécution d'applications scientifiques. Les applications scientifiques peuvent être de nature différente. Elles se différencient par le paradigme de programmation utilisé : les applications peuvent être séquentielles (*e.g.* études paramétriques), ou parallèles communiquant par mémoire partagée ou par échange de messages (*e.g.* simulation numérique). Les applications peuvent avoir des caractéristiques différentes en terme d'exécution. Certaines sont gourmandes en puissance de calcul, alors que d'autres applications requièrent une grande quantité de mémoire ou réalisent des entrées/sorties intensives. Or, quelque soit les applications exécutées et la charge de la grappe, le système doit être capable de garantir qu'un maximum d'applications s'exécute en un minimum de temps.

Définition 20 (Débit applicatif) *Le débit applicatif est le nombre d'applications pouvant effectuer leur exécution complète pendant une unité de temps donnée.*

Aujourd'hui, les grappes présentant un rapport performance/prix intéressant par rapport aux machines parallèles qu'elles tendent à remplacer dans le domaine du calcul scientifique pour certaines applications. Malheureusement, l'utilisation et la programmation des grappes est complexe, les ressources étant physiquement réparties sur les différents nœuds de la grappe. Pour simplifier la programmation et l'utilisation des grappes, une approche possible est de fournir un système à image unique ou « *Single System Image* » (SSI), donnant l'illusion que la grappe est une machine à mémoire partagée. Un SSI gère globalement les ressources disponibles au sein d'une grappe.

Cette approche est celle du système d'exploitation pour grappe KERRIGHED dans lequel s'inscrit notre travail de thèse. Le système KERRIGHED vise à gérer globalement les pages mémoire, les flux de données et des processus en supportant les interfaces système et de programmation standard. Notre travail de thèse est donc une contribution à la fédération des ressources au sein d'un système à image unique, à travers l'étude de la gestion globale des processus.

L'objectif de cette thèse est de concevoir des mécanismes de gestion globale des tâches dans les grappes de calculateurs, afin d'augmenter le *débit applicatif.* La gestion globale des tâches se décompose en deux parties :

1. un ordonnanceur global dont l'objectif est de répartir au mieux les processus des

tâches à exécuter dans la grappe,

2. un ensemble de mécanismes de gestion globale des processus qui constituent le fondement de l'ordonnanceur global en permettant de créer, dupliquer, déplacer et de sauvegarder et restaurer les processus.

Or, chaque classe d'application demande une politique d'ordonnancement propre pour garantir un *débit applicatif* optimal. Par exemple, une application OpenMP ne pourra s'exécuter de façon optimale que si l'ordonnanceur global garantit une utilisation efficace de la mémoire partagée répartie, alors que pour une application MPI, l'ordonnanceur global doit garantir une utilisation efficace de la ressource réseau. Mais la politique d'ordonnancement adéquate pour l'exécution d'une charge applicative donnée dépend également de l'état de la grappe. Par exemple, une politique d'ordonnancement *d'initiative à l'expéditeur* (*c.f.* paragraphe 3.2) est adaptée à une grappe dont la charge moyenne est faible, alors qu'une politique d'ordonnancement *d'initiative au récepteur* (*c.f.* paragraphe 3.2) est adaptée à une grappe dont la charge moyenne est haute.

Il n'existe donc pas une seule politique d'ordonnancement satisfaisante si la charge de travail de la grappe ou le type d'applications exécutées varient dans le temps. Il est donc intéressant de fournir des mécanismes de personnalisation et de reconfiguration de l'ordonnanceur global. La personnalisation et la reconfiguration doivent se faire sans interruption du système, ni des applications en cours d'exécution. Or, la politique d'ordonnancement des systèmes traditionnels est incorporée directement dans le système d'exploitation sans qu'il soit possible de la modifier. Quelques systèmes permettent d'adapter la politique d'ordonnancement mais ne garantissent pas la continuité de fonctionnement du système et des applications.

Notre système doit donc permettre :

- de mettre en place les politiques d'ordonnancement dynamique traditionnelles, que ce soit des politiques de partage d'espace ou de temps, préemptive ou non,
- d'adapter simplement la politique d'ordonnancement aux applications que l'on souhaite exécuter grâce à une définition et une configuration simple des politiques,
- de changer dynamiquement la politique d'ordonnancement sans interrompre les services systèmes, ni l'exécution des applications,
- de permettre l'auto-adaptation de la politique d'ordonnancement en fonction de l'état de la grappe,
- l'utilisation de mécanismes sous-jacents efficaces de gestion globale des processus.

Définition 21 (Ordonnanceur adaptable) *Un ordonnanceur adaptable est un ordonnanceur dont le programmeur système peut personnaliser la politique d'ordonnancement.*

Définition 22 (Ordonnanceur dynamiquement configurable) *Un ordonnanceur dynamiquement configurable est un ordonnanceur dont la politique d'ordonnancement peut être modifiée, configurée pendant l'utilisation de la grappe, sans interruption des applications ni du système d'exploitation.*

6.2 Approche proposée

La mise en œuvre d'un système à image unique peut être effectué au niveau utilisateur, ou dans le noyau. L'approche utilisateur permet de réduire la complexité du développement et de la maintenance du système, alors qu'une approche noyau permet d'accéder à toutes les informations système et donc *a priori* de subir moins de contraintes qu'une approche utilisateur (notamment moins de contraintes sur les applications pouvant être gérées).

Le système KERRIGHED adopte une approche noyau, pour offrir un système à image unique en étendant les fonctionnalités du noyau Linux. L'une des principales difficultés d'une approche noyau est le portage des fonctionnalités mises en œuvre sur des versions différentes du noyau sous-jacent. Pour limiter cette contrainte, les extensions du noyau Linux, effectuées dans le cadre du système d'exploitation KERRIGHED, sont mises en œuvre par des modules noyau. Le module est le mécanisme standard d'extension des fonctionnalités du noyau Linux (notamment utilisé pour les pilotes de périphérique). Les mécanismes du système d'exploitation KERRIGHED sont donc mis en œuvre grâce à des modules noyau et ne demandant que des modifications mineures du noyau permettant d'insérer les points d'entrée nécessaires aux nouveaux mécanismes.

Cette démarche est appliquée pour la conception des sous-systèmes de gestion globale des processus. Afin de limiter au maximum les modifications du noyau, l'ordonnanceur global est indépendant de l'ordonnanceur fourni par le noyau Linux (*ordonnanceur local*). Cette approche permet également de garantir un fonctionnement totalement indépendant de l'ordonnanceur global vis-à-vis de l'ordonnanceur local. L'ordonnanceur global n'effectue que des ajouts ou des retraits de processus au sein de l'ordonnanceur local, tout comme lors de la création ou la terminaison d'un processus au sein d'un système d'exploitation traditionnel. Cela permet non seulement d'activer ou non l'ordonnanceur global, mais également de pouvoir modifier l'ordonnanceur global sans interrompre les ordonnanceurs locaux des nœuds de la grappe.

En outre, le code du noyau est réutilisé au maximum. Cela permet d'éviter toute redondance avec le code du noyau LINUX et simplifie donc la maintenance. Cette réutilisation du code du noyau LINUX simplifie notamment la mise en œuvre et la maintenance des mécanismes de gestion globale des processus bien que cette réutilisation du code nécessite quelques modifications du noyau.

6.3 Ordonnanceur global modulaire

Pour offrir un ordonnanceur adaptable et dynamiquement configurable, nous proposons une approche modulaire [90] (voir figure 6.1). La modularité permet non seulement de créer facilement de nouvelles politiques d'ordonnancement (par de simples compositions de modules), mais également de pouvoir ajouter et retirer à loisir chacun des composants constituant l'ordonnanceur global.

Un ordonnanceur global est, dans le système KERRIGHED, constitué de trois types de composants instanciés sur chaque nœuds de la grappe :

– les *gestionnaires d'ordonnancement global* qui mettent en œuvre les politiques d'ordonnancement globales. Ces gestionnaires sont les seuls à avoir une vue de l'état de la grappe. Leur vue est construite grâce à l'échange d'informations entre les différents nœuds. Cette vue permet de détecter des problèmes globaux d'ordonnancement et d'appliquer des actions d'ordonnancement telles que déplacer un processus ou arrêter un processus. Pour cela, les instances des gestionnaires d'ordonnancement global des nœuds de la grappe communiquent entre eux.
– les *analyseurs locaux* qui détectent les états locaux qui ont un impact sur l'ordonnancement global et qui filtrent les informations système. Pour cela, les analyseurs locaux d'un noeud i filtrent les informations remontées par les sondes du nœud i avant de les transmettre aux gestionnaires d'ordonnancement du nœud i.
– les *sondes* système qui extraient des informations du système d'exploitation local pour ensuite les envoyer aux analyseurs locaux du même nœud. Une sonde peut être associée à un ou plusieurs analyseurs locaux.

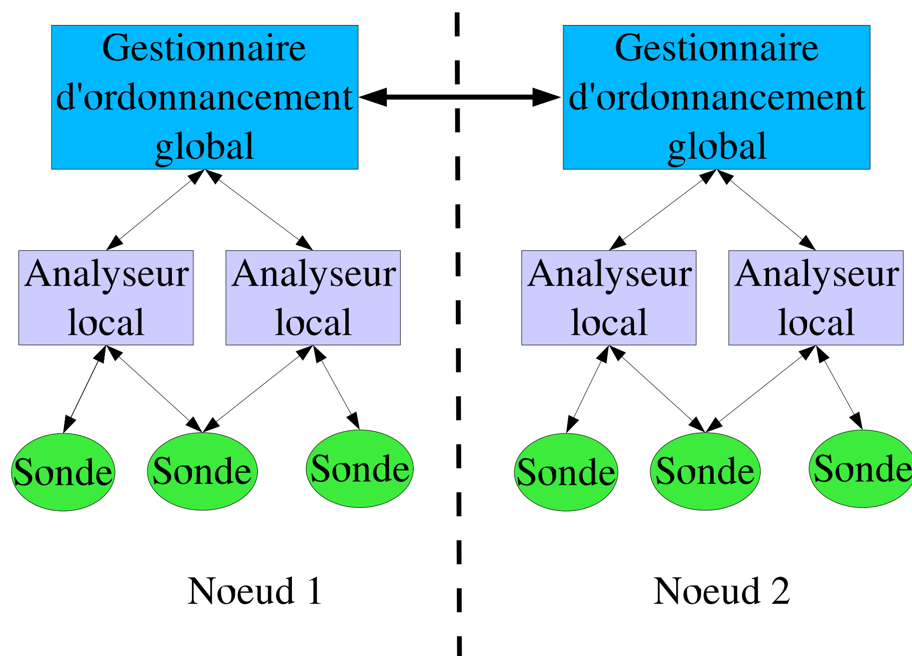

Fig. 6.1 – Architecture de l'ordonnanceur global

Cette architecture modulaire sert de support à la configuration dynamique de l'ordonnanceur global. Il est également intéressant que les politiques puissent s'auto-adapter à l'état courant de la grappe. Cette auto-adaptation permet de définir simplement des politiques d'ordonnancement complexes qui soient adaptées à différents états de la grappe. Il doit donc être possible de définir simplement une politique d'ordonnancement en utilisant les différents composants de l'ordonnanceur global, mais également de pouvoir activer et désactiver chacun des composants. Chaque composant possède une interface générique lui

permettant de se connecter à d'autres composants, ainsi qu'une interface permettant à un autre composant de l'activer et de le désactiver.

Il est ainsi possible, par de simples compositions de créer des politiques d'ordonnancement aussi bien statiques que dynamiques, mais également dynamiquement configurables et adaptatives.

6.4 Mécanismes de gestion globale des processus

Les applications visées étant de nature différente (*e.g.* applications OpenMP, MPI, séquentielles), l'ordonnanceur global doit être fondé sur des mécanismes de gestion globale des processus génériques et performants qui doivent permettre de déployer les applications et de les manipuler pendant leur exécution.

6.4.1 Définitions

En plus des définitions données dans le chapitre 2, nous précisons la définition de retour arrière d'un processus.

> Définition 23 (Retour arrière de processus) *Le retour arrière de processus est similaire à la restauration d'un point de reprise mais concerne un processus en cours d'exécution dans le système alors que la restauration d'un point de reprise concerne un processus qui n'existe plus dans le système. L'exécution d'un processus est reprise depuis l'un de ces points de reprise, sans création d'un nouveau processus. C'est le processus pour lequel le système a créé des points de reprise qui est remis dans l'état de l'un d'entre eux.*

6.4.2 Les processus fantômes

De façon similaire à un système d'exploitation traditionnel, l'ordonnanceur global doit pouvoir gérer l'ensemble du cycle de vie d'un processus, de la création jusqu'à la terminaison en passant par le changement de processeur. L'ordonnanceur global doit donc pouvoir s'appuyer sur plusieurs mécanismes traditionnels : la migration de processus, la création distante de processus, la duplication de processus, la création de point de reprise de processus, et la reprise d'exécution à partir d'un point de reprise. Chacun de ces mécanismes nécessite de pouvoir extraire du système les informations nécessaires et suffisantes pour créer un processus à l'identique, excepté pour la création distante comme nous le verrons ultérieurement. Nous avons donc proposé de factoriser ces différents mécanismes autour d'un mécanisme unique, le mécanisme de gestion des *processus fantôme*. Un processus fantôme regroupe l'ensemble des informations d'un processus nécessaires pour recréer un autre processus à l'identique sur un nœud quelconque de la grappe.

Pour la migration et la duplication de processus, le processus fantôme est géré de la même manière : le processus fantôme est créé (*i.e.* l'état du processus est extrait du système), transféré sur un nœud distant, sur lequel un clone actif est créé. La principale différence entre ces deux mécanismes est que lorsque le processus migre, le processus initial

(sur le nœud d'origine) est stoppé, alors que dans le cas de la duplication, le processus initial reste actif.

Pour la création de point de reprise, le processus fantôme doit être stocké soit en mémoire, soit sur disque, sur le nœud local et/ou sur des nœuds distants.

On constate donc que le processus fantôme peut être manipulé via plusieurs media : le réseau pour la migration et la duplication de processus, la mémoire et les disques avec la création de points de reprise. Il est donc intéressant de pouvoir découpler le mécanisme de création de processus fantôme et l'accès au medium utilisé. De fait, la création du processus fantôme se limite à la définition de l'ensemble des données système nécessaires et suffisantes d'un processus afin de pouvoir en créer un clone indépendamment de sa localisation.

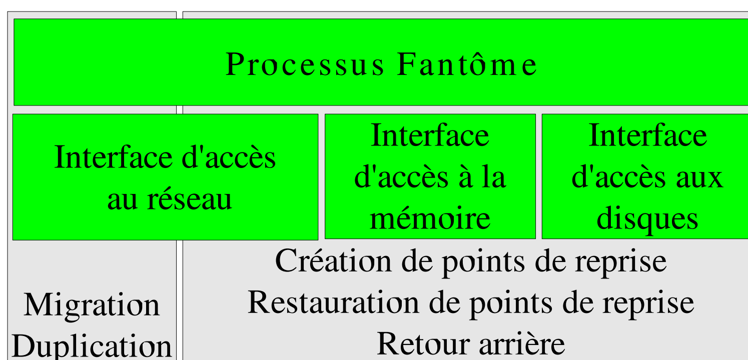

FIG. 6.2 – Manipulation du processus fantôme

Afin de faciliter la création du processus fantôme, deux services système ont été mis en place : un service d'exportation et un service d'importation de processus. Le service d'exportation permet d'extraire le processus du système.

Aussi, ce système d'exportation associé à une méthode d'accès à une ressource de la grappe permet de mettre en œuvre la migration, la duplication et la création de points de reprise. Avec cette approche, il est également possible de mettre en œuvre simplement de nouveaux mécanismes de gestion globale de processus. Par exemple, un système de *migration par ensemble* a été mis en œuvre en créant simplement un processus fantôme dans un tampon en mémoire, puis en envoyant le contenu de ce tampon à travers le réseau sur un nœud distant (évitant ainsi de très nombreux et coûteux accès réseau comme lorsque le processus fantôme est directement envoyé à travers le réseau, chaque structure étant envoyée tour à tour).

Le service d'importation permet de créer un processus actif à partir d'un processus fantôme sur un nœud. Tout comme l'exportation, ce système d'importation associé à une méthode d'accès à une ressource de la grappe permet de créer un processus sur un nœud dans le cadre d'une migration, d'une duplication ou de la restauration d'un point de reprise. La création de processus lors de la migration fait appel au mécanisme de création de processus du noyau du nœud. Le service d'importation du mécanisme de migration, de duplication et de restauration de point de reprise diffère dans la phase de traitement qui

suit la création d'un nouveau processus. Cette phase de traitement permet de donner une sémantique différente au processus au sein du système local.

6.4.3 Interface Pthread

Les mécanismes de gestion globale des processus natifs du système KERRIGHED permettent de mettre en œuvre les interfaces standard utilisées pour la programmation d'applications, telle que l'interface *pthread* [45]. La mise en œuvre d'une interface *pthread* est indispensable pour offrir d'une grappe la vision d'une machine SMP et pour en faciliter l'utilisation et la programmation.

Par exemple, les mécanismes de gestion globale des processus à travers le mécanisme de processus fantôme associés aux mécanismes de KERRIGHED relatifs à la synchronisation et au partage de mémoire ont permis de mettre en œuvre une interface *pthread* efficace.

Le système d'exploitation KERRIGHED supporte l'exécution d'applications *pthread* sans modification des applications.

L'interface *pthread* de KERRIGHED a permis l'exécution d'applications OPENMP sur grappe sans modification des applications et en les compilant avec un compilateur existant non modifié ciblant les *pthreads*. Bien entendu, fonctionnalité n'est pas synonyme de performance. L'obtention de performances raisonnables lors de l'exécution d'applications OPENMP sur grappe est conditionnée par l'application elle-même dont la granularité des données manipulées ne doit pas être trop fine, et par l'utilisation d'un compilateur prenant en compte les spécificités d'une grappe[19] (la page est le grain de partage dans une grappe alors que au sein d'un SMP c'est la ligne de cache).

6.5 Environnement de développement de politiques d'ordonnancement global

Afin de faciliter la définition de nouvelles politiques d'ordonnancement, un environnement de développement a été mis en œuvre. Cet environnement de développement regroupe les primitives de base permettant de créer une politique d'ordonnancement, ainsi que les primitives permettant d'utiliser des composants d'ordonnanceur global. Un ensemble de composants est également fourni avec le système KERRIGHED et peuvent être utilisés pour le développement de nouveaux ordonnanceurs globaux.

Pour simplifier la manipulation de ces primitives, une interface graphique (voir figure 6.3) a été développée, fournissant automatiquement le prototype des composants et la liste des primitives mises à disposition des développeurs système.

6.6 Résumé

Le système KERPROC permet de gérer globalement les processus au sein du système KERRIGHED. Pour cela, KERPROC propose un mécanisme d'ordonnancement global adaptable fondé des mécanismes de gestion globale des processus.

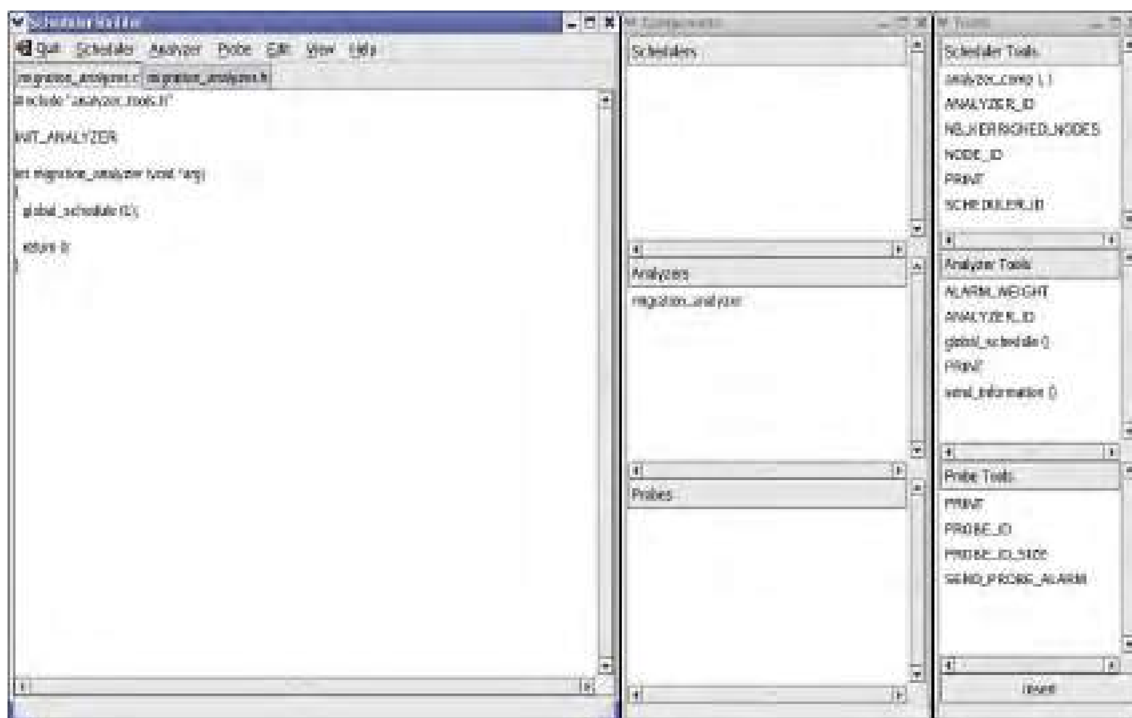

FIG. 6.3 – Interface de développement de composants d'ordonnanceurs globaux

Le mécanisme d'ordonnancement du système KERRIGHED est divisé en trois partie : les gestionnaires d'ordonnancement global, les analyseurs locaux et les sondes. Les gestionnaires d'ordonnancement global sont chargés de mettre en œuvre la politique d'ordonnancement global. Les sondes sont chargées d'extraire les informations système des nœuds de la grappe nécessaire à l'ordonnancement global. Ces informations sont filtrées par les analyseurs locaux avant d'être relayées vers un gestionnaire d'ordonnancement global. Les analyseurs locaux sont également chargés de détecter les état locaux qui influencent l'ordonnancement global. Cette modularité permet d'offrir simplement un mécanisme de configuration de la politique d'ordonnancement. De plus, un environnement de programmation a été mis en œuvre pour cacher la complexité de programmation noyau lors de la programmation de nouveaux ordonnanceurs globaux, qui peuvent être ensuite dynamiquement configurés et implantés.

L'ordonnanceur global de KERRIGHED s'appuie sur des mécanismes de gestion globale de processus efficaces fondé sur un mécanisme unique appelé *processus fantôme*. Ce mécanisme permet d'extraire du système une image d'un processus. Cette image peut être ensuite manipulée dans la grappe et permet de créer des processus clones indépendamment de la localisation initiale du processus. Dans KERRIGHED, les mécanismes de duplication, de migration et de création/restauration de points de reprise sont fondés sur le mécanisme de *processus fantôme*.

7 CONCEPTION D'UN ORDONNANCEUR GLOBAL ADAPTABLE

Nous proposons pour le système à image unique KERRIGHED une architecture d'ordonnanceur global modulaire pour permettre de choisir, par configuration, la politique d'ordonnancement globale pour une grappe donnée. En outre, l'architecture que nous proposons permet la configuration dynamique, *i.e.* le changement à chaud de la politique d'ordonnancement, sans arrêt du système ni des applications à l'initiative de l'administrateur ou de la politique d'ordonnancement globale courante. Compte tenu de cette dernière caractéristique nous qualifions l'ordonnanceur global de KERRIGHED d'adaptable [88].

Nous détaillons tout d'abord les différents composants constituant l'ordonnanceur global adaptable de KERRIGHED que sont les gestionnaires d'ordonnancement global, les analyseurs locaux et les sondes. Nous montrons ensuite comment ces composants communiquent entre eux et enfin comment il est possible de dynamiquement configurer les différents composants constituant un ordonnanceur global.

7.1 Définition des composants de l'ordonnanceur global

L'ordonnanceur global est constitué de trois types de composants : les gestionnaires d'ordonnancement global mettant en œuvre la politique d'ordonnancement à appliquer au sein de la grappe, les sondes système permettant d'extraire des informations système de chaque nœud des analyseurs locaux chargés de filtrer les informations système, et qui servent d'intermédiaires entre les sondes et les gestionnaires d'ordonnancement global.

La modularité de l'architecture permet de faciliter la création d'une politique d'ordonnancement. La complexité système se limite aux sondes, les autres composants n'étant que la mise en œuvre d'algorithmes. Les analyseurs locaux permettent d'isoler la problématique de la pertinence des informations système récupérées par les sondes. Ils ne relaient que les informations exploitables par les gestionnaires d'ordonnancement global. Les analyseurs locaux sont également capables de détecter les états du nœud pouvant influencer l'ordonnancement global.

La modularité permet également une reconfiguration simple et efficace : il est possible de changer la configuration d'un composant sans interruption des autres composants.

7.1.1 Les gestionnaires d'ordonnancement global

7.1.1.1 Définition

Définition 24 (Gestionnaire d'ordonnancement global) *Un Gestionnaire d'Ordonnancement Global (GOrG) est chargé de mettre en œuvre une politique d'ordonnancement global. Une instance de GOrG s'exécute sur chaque nœud de la grappe.*

Pour prendre des décisions d'ordonnancement, un GOrG s'appuie sur sa vision de l'état de la grappe. Cette vision peut être complète (le GOrG global connaît l'état de tous les nœuds) ou partielle (le GOrG ne connaît l'état que d'un sous-ensemble des nœuds comme dans le système pour grappe MOSIX où la vision partielle de la grappe est associée à une politique d'ordonnancement probabiliste) en fonction de la politique d'ordonnancement global mise en œuvre. Un GOrG construit sa vision de l'état de la grappe à partir des informations qui lui sont transmises par les analyseurs locaux de son nœud ou par les GOrG des autres nœuds de la grappe. Une partie des mécanismes associés aux GOrG est donc la gestion de ces informations.

Lorsqu'un déséquilibre de charge au sein de la grappe est détecté, la politique d'ordonnancement globale permet de décider d'un changement dans l'allocation des processeurs aux processus. Il s'agit de décider (i) quels sont les processus à déplacer ou à suspendre et (ii) sur quels nœuds les placer, les déplacer ou stocker leur image. Deux fonctions sont donc communes à la majorité des politiques d'ordonnancement : l'élection d'un processus et d'un nœud pour appliquer un nouvel ordonnancement.

Une politique globale d'ordonnancement peut donc être divisée en trois parties : l'algorithme d'ordonnancement global qui définit les cas où un nouvel ordonnancement des processus est nécessaire ; un algorithme pour désigner le ou les processus qui subiront ce nouvel ordonnancement (*e.g.* une migration, une suspension) ; un algorithme pour désigner les nœuds cibles pour ce nouvel ordonnancement. Ces trois parties constituent le cœur de la définition d'un GOrG et doivent être définies par le programmeur de nouvelles politiques.

7.1.1.2 Les interfaces

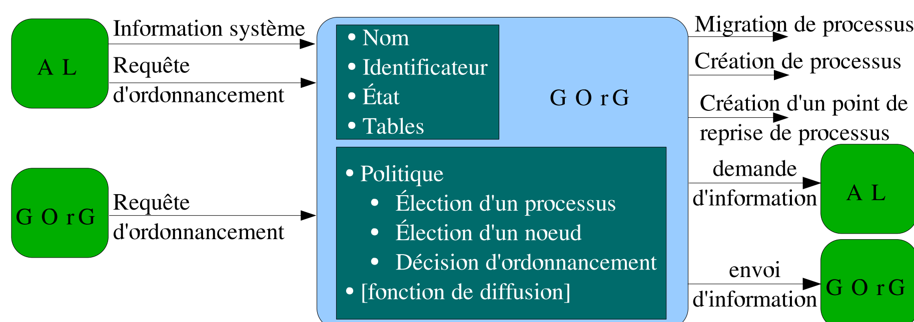

FIG. 7.1 – Structure d'un gestionnaire d'ordonnancement global

Chaque GOrG possède des données privées qui le définissent (voir figure 7.1) :
- un nom permettant au programmeur d'identifier simplement un GOrG,
- un identifiant unique utilisé pour le fonctionnement interne des GOrG,
- un état (activé ou désactivé),
- une ou plusieurs tables pour stocker les informations système des différents nœuds de la grappe.

Le nom du GOrG est défini par le programmeur dans un fichier de configuration XML associé (voir paragraphe 7.3), tandis que l'identifiant unique est défini par le système, lors du lancement de l'ordonnanceur. L'identifiant unique du GOrG, et son état sont des informations opaques pour le programmeur système, elles ne sont accessibles que par des fonctions prédéfinies (*c.f.* paragraphe 7.3.4) et ne peuvent donc pas être manipulées directement par le programmeur.

Chaque instance de GOrG communique avec les autres pour la diffusion des informations système. Lorsqu'une instance de GOrG reçoit une information du système local, cette information peut être diffusée à d'autres nœuds vers d'autres instances de GOrG. Ainsi chaque instance de GOrG possède une vision de la grappe cohérente pour la politique d'ordonnancement qu'elle met en œuvre. Deux cas de figure sont possibles : le programmeur de la politique d'ordonnancement a spécifié ou non une fonction de diffusion. Si aucune fonction de diffusion n'est spécifiée, l'information système est automatiquement envoyée de façon transparente à l'ensemble des nœuds dont lui-même (voir figure 7.2). Si une fonction de diffusion est spécifiée, elle est appelée afin de diffuser les informations système à un sous-ensemble des nœuds de la grappe. Les informations système représentant l'état du système ne sont ensuite stockées dans des tables qu'à la réception d'un message d'information venant d'une instance de GOrG (lui-même pour les informations locales).

Lorsque les GOrG sont fondés sur une vision partielle de la grappe, il peut être nécessaire de procéder à une négociation pour garantir que l'ordonnancement décidé est acceptable. L'instance de GOrG voulant appliquer un nouvel ordonnancement envoie une requête d'ordonnancement au(x) nœud (s) cible(s). Si toutes les requêtes sont acceptées, le nouvel ordonnancement est appliqué, sinon il est abandonné. Pour décider si une requête d'ordonnancement est acceptable, une instance de GOrG peut interroger les analyseurs locaux qui lui sont associés. A son tour, un analyseur local interrogé par une instance de GOrG peut interroger une sonde. Il est ainsi facile de créer des politiques d'ordonnancement global fondées sur une vision partielle de la grappe, des primitives de négociation (*e.g.* envoi de requête, interrogation des analyseurs locaux et des sondes) permettant au programmeur de mettre en œuvre des politiques de négociation. Lors de la création d'un GOrG, le programmeur doit définir plusieurs fonctions internes :
- une fonction de diffusion,
- une fonction d'ordonnancement,
- une fonction pour la politique d'élection d'un nœud distant,
- une fonction de traitement des informations système des nœuds de la grappe, afin de permettre au programmeur de nouvelles politiques d'ordonnancement de fonder sa politique sur une vision globale ou partielle de l'état de la grappe,
- une fonction d'élection d'un processus pouvant subir un nouvel ordonnancement (dans

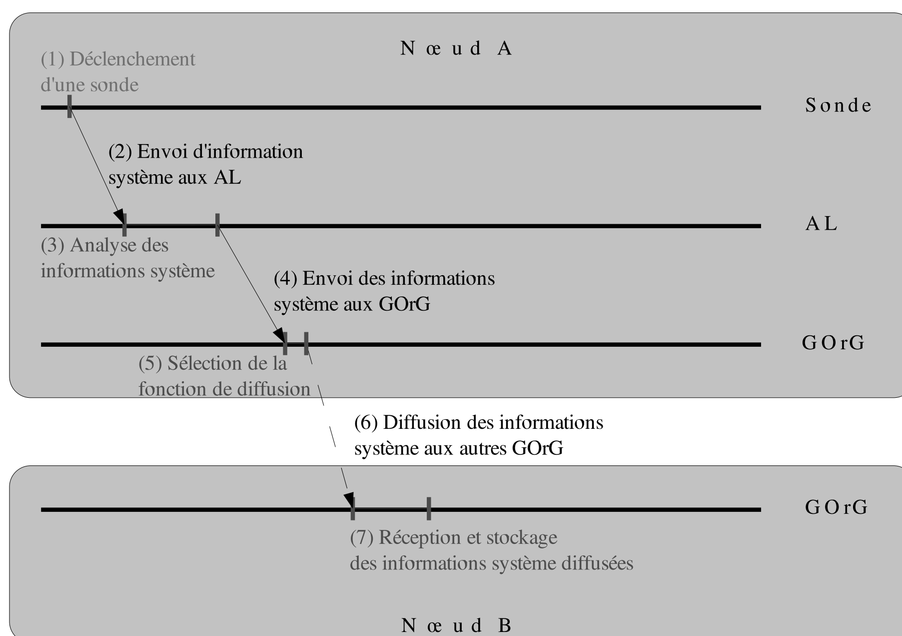

FIG. 7.2 – Traitement des informations système au sein de l'ordonnanceur global

le cas d'un équilibrage de charge).

Chaque instance de GOrG peut être associée avec un ou plusieurs analyseurs locaux (qui sont définis dans le paragraphe 7.1.2) qui lui fournissent les informations sur le système local et qui lèvent des alarmes en cas de surcharge locale. Une liste des analyseurs locaux associés à chaque instance de GOrG est tenue à jour, grâce aux informations des fichiers de configuration. Dans le code généré grâce aux fichiers de configuration (voir paragraphe 7.3) se trouvent les enregistrements des différents analyseurs locaux. Une fonction *add_analyzer* permet de mettre à jour à la fois la liste des instances de GOrG associés aux analyseurs locaux et la liste des analyseurs associés aux instances de GOrG. Cette fonction est utilisée lors de la création des instances de GOrG, et notamment lors de la configuration de l'ordonnanceur global comme nous le verrons dans le paragraphe 7.3.

7.1.2 Les analyseurs locaux

7.1.2.1 Définition

> Définition 25 (Analyseur Local) *Un Analyseur Local (AL) est chargé d'analyser et de filtrer les informations système extraites par les sondes avant de les remonter, le cas échéant, au GOrG s'exécutant sur le même nœud.*

Le filtrage des informations système permet de ne garder que les informations système nécessaires au fonctionnement de l'ordonnanceur global, optimisant ainsi son fonctionnement général.

L'analyse des informations système permet de détecter localement les situations malsaines du point de vue de l'ordonnancement global (*e.g.* la saturation de la mémoire, le ping-pong d'une page mémoire entre deux nœuds). La détection d'une situation pouvant conduire à rectifier l'allocation courante des processus aux différents nœuds de la grappe est donc effectuée par deux types de composants : un GOrG à partir de sa vision globale de la grappe ou un analyseur local à partir à partir de sa vision du nœud sur lequel il s'exécute. Cette décomposition facilite la conception des politiques d'ordonnancement global.

Chaque AL est associé à une ou plusieurs instances locales de GOrG et de sondes. Comme pour les GOrG, les AL possèdent une liste permettant de stocker les associations avec les GOrG et les sondes. De plus, des fonctions sont définies pour insérer chaque AL dans le système en tenant à jour les informations des différentes listes : *add_probe* pour associer un AL à une sonde et comme nous l'avons vu *add_analyzer* pour associer un GOrG et un AL, la liste des GOrG associés étant mise à jour lors du lancement des GOrG.

7.1.2.2 Les interfaces

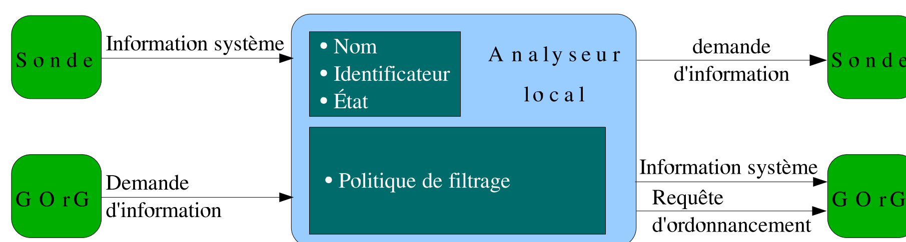

FIG. 7.3 – Structure d'un analyseur local

Les AL possèdent des données privées (voir figure 7.3) :
- un nom permettant au programmeur d'identifier simplement un AL,
- un identifiant unique, utilisé pour le fonctionnement interne des AL,
- un état (activé ou désactivé),

Seule la politique de filtrage est à définir pour chaque AL par le programmeur.

7.1.3 Les sondes système

7.1.3.1 Définition

> Définition 26 (Sonde système) *Une sonde système est chargée d'extraire du système d'exploitation local les informations système nécessaires au fonctionnement d'un ordonnanceur global.*

> Définition 27 (Information système) *Une information système est une information du noyau donnant l'état d'une ressource ou d'un service système.*

Il existe deux types de sondes système : les *sondes passives* et les *sondes actives*. Les sondes passives (voir figure 7.4) se déclenchent suite à un évènement système. Les évènements systèmes sont de deux types : un évènement noyau ou un évènement de l'ordonnanceur global.

> Définition 28 (Événement de l'ordonnanceur global) *Un événement de l'ordonnanceur global est un événement se déroulant dans le contexte d'exécution noyau du système d'un nœud mais indépendant du fonctionnement de celui-ci et qui est propre au fonctionnement de l'ordonnanceur global.*

Un exemple d'événement de l'ordonnanceur global est une migration de processus. Il est donc possible de créer une sonde passive se déclenchant lorsqu'un processus migre.

On peut aussi mettre en œuvre une sonde de détection de ping-pong de pages mémoire[75] au sein de la Mémoire Partagée Répartie (MPR) de KERRIGHED, 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 (voir figure 7.5) surveillent une entité système périodiquement. Dans ce cas, la sonde est associée à un *timer* qui, à chaque échéance, déclenche la sonde. Par exemple, on peut imaginer une sonde chargée de fournir la charge moyenne d'un processeur lors de la dernière minute.

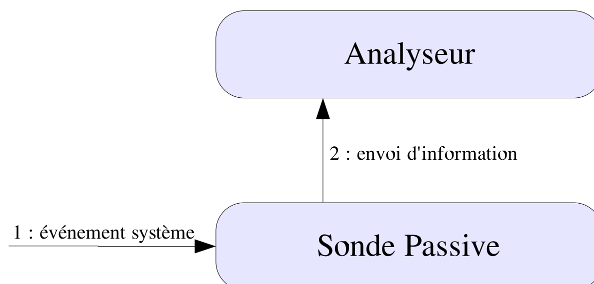

FIG. 7.4 – Les sondes passives

Les sondes devant extraire des informations système du nœud, elles sont enfouies dans le système d'exploitation 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 de la dernière minute, des 5 et 15 dernières minutes) qui peuvent être réutilisées pour la mise en œuvre de nouvelles politiques d'ordonnancement.

L'intégration de nouvelles sondes dans le système KERRIGHED est du ressort des programmeurs système.

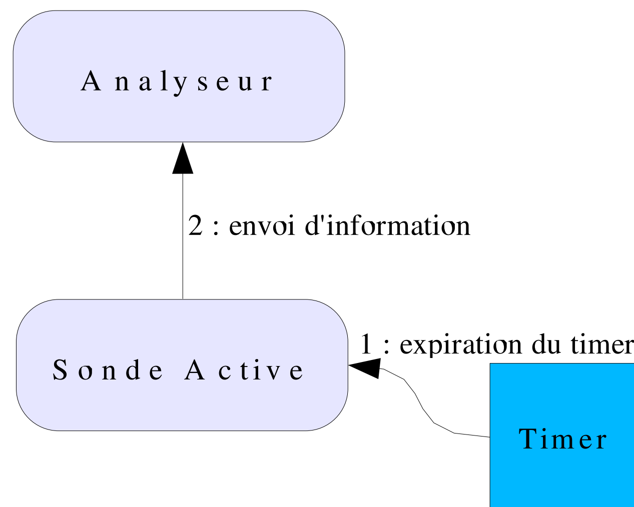

FIG. 7.5 – Les sondes actives

7.1.3.2 Les interfaces

Une sonde possède des informations privées :
- son nom permettant au programmeur d'identifier simplement une sonde,
- son identifiant unique utilisé pour le fonctionnement interne des sondes,
- son état (activé ou désactivé),

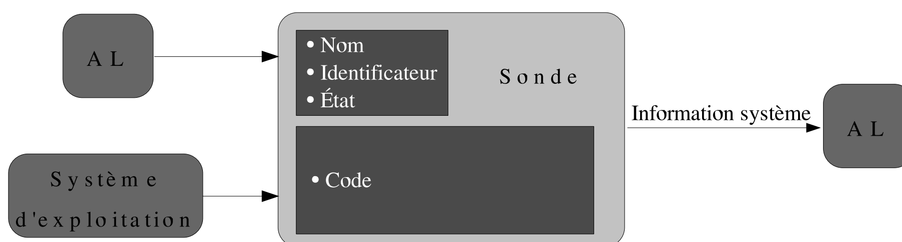

FIG. 7.6 – Structure d'une sonde système

7.2 Communication entre sondes, analyseurs locaux et gestionnaires d'ordonnancement global

Sur chaque nœud, chaque composant (*i.e.* sonde, AL ou GOrG) peut être associé à un ou plusieurs autres composants. Or, ces associations peuvent être modifiées pendant le fonctionnement du système en cas de changement de configuration. Pour prendre en compte ces modifications à chaud, un modèle de communication inter-composant fondé sur des échanges de requêtes a été mis en place (les différents types de requête sont donnés

Nom de la requête	Type	Description
SEND_ALARM	1	Envoi d'information d'une sonde vers des AL
PROBE_ALARM	2	Diffuse l'information obtenue d'une sonde vers les AL cible (après réception d'une requête SEND_ALARM
CALL_PROBE	3	Envoie une requête de déclenchement vers une sonde
SCHEDULING_REQUEST	4	Demande d'ordonnancement global d'un AL vers des GOrG
CHECK_SCHEDULE	5	Négociation d'un ordonnancement entre deux GOrG
SCHEDULE_ACK	6	Acceptation d'un ordonnancement lors d'une négociation
SCHEDULE_REJ	7	Refus d'un ordonnancement lors d'une négociation
LOCAL_INFO_REQUEST	8	Notification de la réception d'une information envoyée par un GOrG
MIGRATION_REQUEST	9	Demande de migration

TAB. 7.1 – Liste des requêtes du système de communication inter-composants

dans le tableau 7.1). Tous les composants envoient leurs requêtes à un gestionnaire qui est chargé de trouver les destinataires de ces requêtes. Ce fonctionnement asynchrone permet de garantir qu'aucun service ne sera interrompu si l'un des composants devient inactif ou s'il est évincé de l'ordonnanceur global. Cela permet donc de supporter les changements de configuration mais également les défaillances de nœuds : chaque nœud a un fonctionnement indépendant, hormis les communications entre gestionnaires d'ordonnancement global qui peuvent être spécifiées par le programmeur.

Chaque composant est associé à une file d'attente de type *FIFO*, et toute requête à destination d'un composant est insérée dans la file d'attente correspondante. Lorsqu'une requête est insérée dans une file d'attente, le gestionnaire de communication inter-composant est appelé (voir algorithme 1) et traite toutes les requêtes présentes dans les diverses files d'attente. Le gestionnaire est endormi lorsque toutes les files sont vides. Il est réveillé lorsqu'une requête est insérée dans une file d'attente et il ne s'endort à nouveau que lorsque les files sont toutes vides (voir algorithme 2, 3, 4 et 5).

Chaque requête est analysée et en fonction des résultats de cette analyse, le gestionnaire de communication entre composants envoie une ou plusieurs requêtes vers d'autres composants ou bien invoque directement un composant (voir figure 7.7).

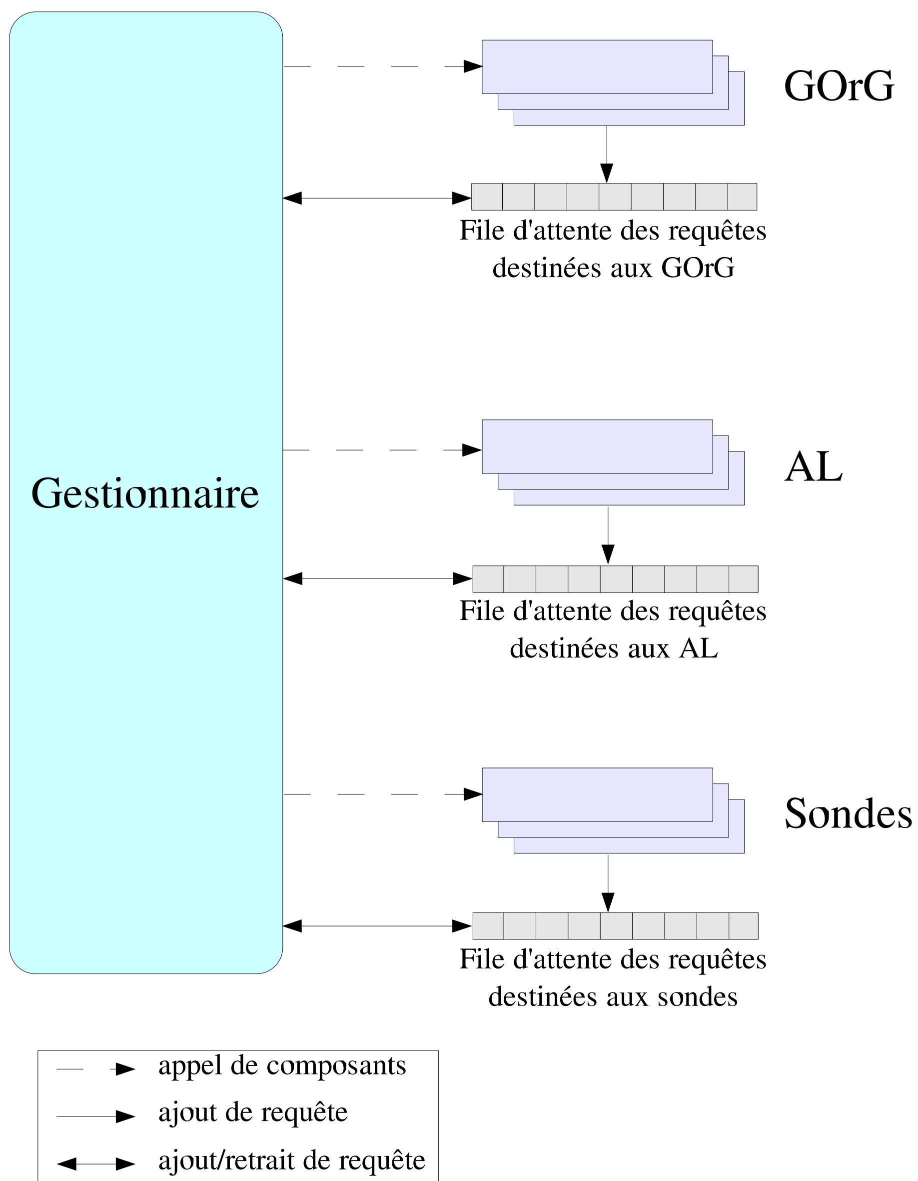

FIG. 7.7 – Communications inter-composants

Algorithme 1 :Algorithme d'insertion de requête

```
Insertion de requête :
{
    Insérer requête dans file d'attente ;
    si gestionnaire de communication endormi alors
       réveiller gestionnaire de communication ;
    fin si
}
```

Algorithme 2 :Algorithme du gestionnaire de communication entre composants (voir en complément les algorithmes 3, 4, 5)

```
Gestionnaire de Communication :
{
    tant que gestionnaire de communication actif faire
          si files d'attente vides alors
             Devenir inactif
          fin si
          exécuter le traitement des requêtes des sondes ;
          exécuter le traitement des requêtes des AL ;
          exécuter le traitement des requêtes des GOrG ;
    fin tant que
}
```

Toute requête est composée d'un type, de données et de la taille de ces données, les requêtes pouvant être de taille quelconque (voir figure 7.8).

Type	Taille des données	Données

FIG. 7.8 – Format des requêtes inter-composants

7.3 Configuration et gestion dynamique

Les sondes, analyseurs locaux et gestionnaires d'ordonnancement global peuvent être vus comme des composants que l'on peut assembler pour créer de nouvelles politiques d'ordonnancement. Afin de faciliter cette composition, elle est décrite dans des fichiers de configuration en XML [94] (« *eXtensible Markup Language* »). L'utilisation de XML, qui

Algorithme 3 :Algorithme de traitement des requêtes des sondes

```
Traitement des requêtes des sondes :
{
    tant que file d'attente non vide faire
        selon type de la requête
          cas Envoi information système :
              Trouver les AL associés actifs ;
              Envoyer une requête d'information à chaque AL actif ;
          cas Interrogation d'un AL :
              si sonde cible active alors
                Déclencher la sonde ;
              fin si
        fin selon
        Effacer la requête ;
    fin tant que
}
```

est une forme restreinte de SGML [47] normalisée par le W3C, permet de créer simplement des documents structurés indépendamment de la mise en forme de ceux-ci, tout en restant lisibles par les utilisateurs. De nombreux outils sont disponibles pour la manipulation des documents XML et sont largement utilisés pour la diffusion de documents et comme format pour les fichiers de configuration des applications. Parmi ces outils, nous utilisons XSL [95] (« *Extensible Style Sheet Language* ») et plus précisément XSL-T [93] (« *Extensible Style Sheet Language Transformations* »). XSL permet de définir des feuille de style pour document XML. Un fichier XSL est lui-même un document XML et doit donc répondre à l'ensemble des spécifications du W3C. XSL-T a été conçu pour être utilisé comme une partie de XSL et définit un langage de transformation d'un document XML. Ce langage permet de transformer un document XML vers un format quelconque (*e.g.* un autre document XML, un fichier C) en parcourant le fichier XML et en appliquant les règles décrites par le fichier XSL-T. Cette transformation est effectuée par un moteur. Le système KERRIGHED fonctionne de base avec le moteur nommé Xalan[6] fourni par *apache.org* mais peut également fonctionner avec tout autre moteur respectant les normes du W3C.

Le système KERRIGHED fournit des fichiers XSL/XSL-T qui permettent, à partir des fichiers de configuration en XML, de générer automatiquement le code d'un module noyau, appelé *chargeur* qui permet de charger/décharger à volonté les différents composants de l'ordonnanceur global (figure 7.9). Lors du chargement, le *chargeur* initialise le système KERRIGHED et les composants de l'ordonnanceur global, pour ensuite les activer. Lors du déchargement, le *chargeur* met à jour les différentes informations système propre à chaque composant afin de les désactiver sans interruption des services système, ni des applications.

Les GOrG, les AL et les sondes sont décrits dans des fichiers différents.

Algorithme 4 :Algorithme de traitement des requêtes des AL

Traitement des requêtes des AL :
{
  tant que file d'attente non vide **faire**
    selon type de la requête
      cas Réception d'informations système d'une sonde :
        Trouver les AL associés actifs ;
        Exécuter les AL actifs avec les informations système comme paramètres ;
      cas Interrogation provenant d'un GOrG :
        si sonde cible active **alors**
          Appeler la sonde cible
        fin si
    fin selon
    Effacer la requête ;
  fin tant que
}

7.3.1 Configuration des gestionnaires d'ordonnancement global

La configuration des GOrG s'effectue dans le fichier *scheduler_ conf.xml* (voir figure 7.10). Dans ce fichier, le programmeur définit l'ensemble des nœuds sur lesquels le GOrG sera déployé, facilitant ainsi le partitionnement de la grappe. Il définit également le nom de l'ordonnanceur, le nom du fichier contenant le code du GOrG, et les fonctions propres à chaque GOrG et définissant la politique d'ordonnancement mise en œuvre. Le nom de l'ordonnanceur est utilisé dans la programmation des autres composants comme identificateur. Le nom du fichier contenant le code du GOrG est également le nom de la fonction principale de la politique d'ordonnancement, appelée lorsqu'un nouvel ordonnancement est appliqué. Enfin, les fonctions propres à chaque GOrG et définissant la politique d'ordonnancement mise en œuvre permettent de définir la fonction d'élection d'un nœud, la fonction d'élection de processus pour l'application d'un nouvel ordonnancement. De plus, afin que l'ordonnanceur global soit entièrement configurable, il est possible de définir des fonctionnalités appliquées à chaque évènement propre au fonctionnement des GOrG, notamment lors de la réception et du stockage d'une information système (fonction *on_ node_ info_ update*). En plus de ces éléments, il est nécessaire de définir les AL qui sont associés à chaque GOrG. Pour cela, il suffit de définir le nom de l'AL et optionnellement, par l'utilisation d'une balise XML *global_ info*, stipuler que les informations système relayées par l'AL doivent être stockées dans une table (ce stockage est effectué de façon transparente et automatique par le gestionnaire de communication).

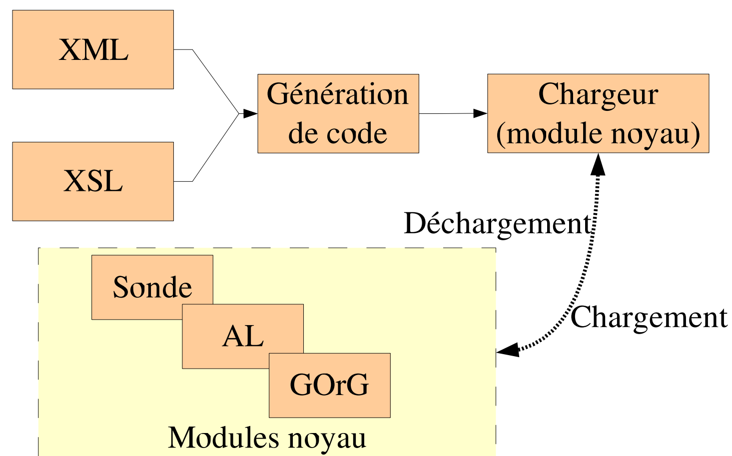

FIG. 7.9 – Configuration de l'ordonnanceur global grâce à des fichiers XML

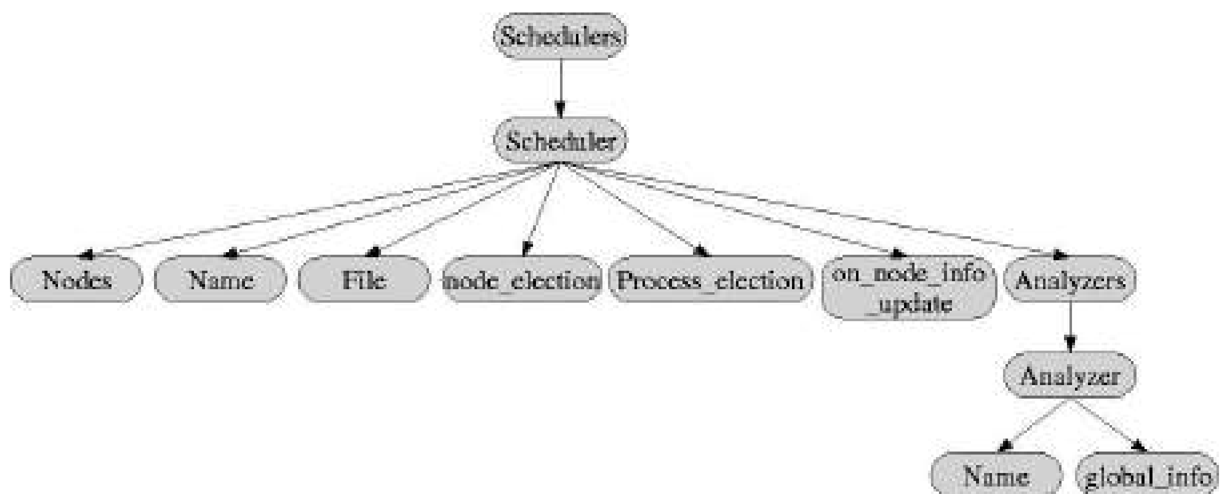

FIG. 7.10 – Représentation de l'arbre DOM (« *Document Object Model* ») du fichier de configuration XML des GOrG

Algorithme 5 :Algorithme de traitement des requêtes des GOrG

```
Traitement des requêtes des GOrG :
{
    tant que file d'attente non vide faire
        selon type de la requête
          cas Réception d'informations système d'un AL :
              Trouver les GOrG associés actifs
              Diffuser les informations
          cas Requête d'ordonnancement :
              Trouver les GOrG associés actifs
              Appeler la politique d'ordonnancement global
          cas Requête de négociation d'ordonnancement :
              Appeler la fonction de négociation d'ordonnancement
          cas Acceptation d'ordonnancement :
              Effectuer l'ordonnancement en cours
          cas Refus d'ordonnancement :
              Abandonner l'ordonnancement en cours
        fin selon
        Effacer la requête ;
    fin tant que
}
```

7.3.2 Configuration des analyseurs locaux

La configuration d'un AL s'effectue en deux parties : la définition des données internes à l'analyseur (son nom et le nom du fichier contenant le code de l'AL), et la liste des sondes associées (voir figure 7.11). Le nom du fichier permet de définir la fonction principale des AL, appelée lors de l'invocation de l'AL suite à la requête d'un GOrG ou d'une sonde. Les sondes associées à l'AL sont définies en insérant le nom de chaque sonde dans une balise XML *probe*. Enfin, le programmeur définit pour chaque AL la liste des nœuds sur lesquels les AL sont déployés.

7.3.3 Configuration des sondes

La configuration des sondes diffère pour les sondes actives et les sondes passives. Pour les sondes passives, le programmeur doit définir le nom de la sonde, le nom du fichier (qui est également le nom de la fonction mettant en œuvre le mécanisme de la sonde), ainsi que la liste des nœuds sur lesquels la sonde sera déployée. Pour les sondes actives, le programmeur doit en outre définir le temps de fonctionnement de l'alarme associée à la sonde.

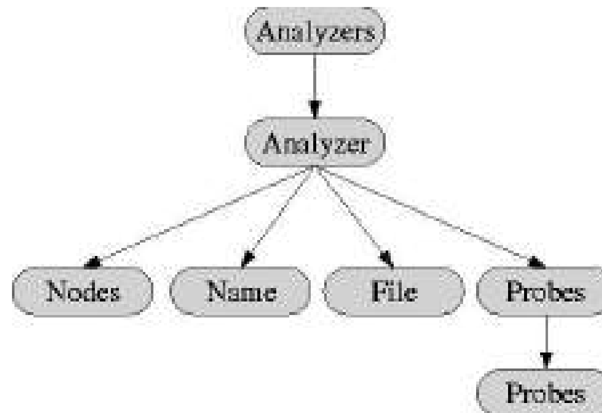

FIG. 7.11 – Représentation de l'arbre DOM (« *Document Object Model* ») du fichier de configuration XML des AL

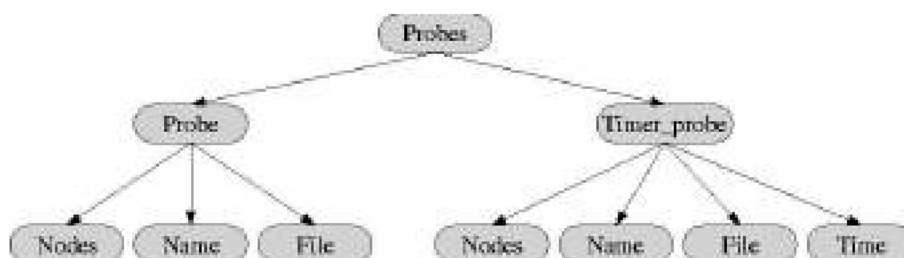

FIG. 7.12 – Représentation de l'arbre DOM (« *Document Object Model* ») du fichier de configuration XML des sondes

7.3.4 Politiques auto-adaptables

Tous les composants de l'ordonnanceur global peuvent être activés et désactivés par l'utilisation d'une interface simple : *active_GOrG*, *active_al* et *active_probe* pour activer les différents composants d'un ordonnanceur global ; *unactive_GOrG*, *unactive_al* et *unactive_probe* pour les désactiver. Cette interface modifie l'état des composants, le gestionnaire de communication entre composants pouvant ainsi déterminer la liste exacte des composants destinataires lors du traitement d'une requête. Cela permet d'écrire des politiques d'ordonnancement qui s'auto-adaptent à l'état de la grappe en désactivant un certain nombre de composants de l'ordonnanceur pour les remplacer par d'autres. On peut donc imaginer par exemple de basculer sur une politique d'ordonnancement effectuant moins de communications via le réseau si celui-ci est surchargé, alors qu'en fonctionnement normal, une politique plus efficace mais nécessitant plus de communication réseau est active.

7.4 Résumé

Nous proposons un ordonnanceur global composé de trois modules : les gestionnaires d'ordonnancement (GOrG), les analyseurs locaux (AL) et les sondes. Les GOrG permettent de mettre en œuvre la politique d'ordonnancement. Cette politique est fondée sur une vision de l'état de la grappe, vision constituée grâce aux informations système extraites par les sondes. L'ensemble de ces informations n'étant pas utilisable par les GOrG, les AL peuvent les filtrer. Les AL ont également pour rôle de détecter les états locaux pouvant influencer l'ordonnancement global des tâches au sein de la grappe. Un outil de programmation de composants permet de facilement mettre en œuvre de nouveaux composants et donc de programmer de nouveaux ordonnanceurs, qu'ils soient dynamiques ou statiques. Pour le moment, l'outil de programmation fournit des primitives pour le partage spatial des ressources mais il peut être étendu pour offrir des primitives de partage temporel des ressources.

Un ordonnanceur global est composé sur chaque nœud d'instance des différents composants qui le constitue. Les communications entre composants, que ces communications soient locales ou globales entre GOrG, sont assurées par des gestionnaires qui assurent une continuité de fonctionnement même en cas de changement d'état ou de configuration de l'un des composants.

Il est ainsi possible de configurer dynamiquement la politique d'ordonnancement à mettre en œuvre dans la grappe. Cette configuration est facilité par un mécanisme automatique de génération de modules noyau à partir de fichiers de configuration XML. L'ordonnanceur peut donc être adapté au type d'application en cours d'exécution.

8 MÉCANISMES DE GESTION DES PROCESSUS DANS UNE GRAPPE

L'ordonnancement global de processus est en charge du déploiement des tâches sur la grappe ainsi que de l'équilibrage de la charge. Le SSI KERRIGHED doit donc offrir à l'ordonnanceur global des mécanismes de gestion de processus permettant la création à distance et la duplication de processus pour le déploiement de tâches sur les nœuds de la grappe ainsi que la migration et la sauvegarde et la restauration de points de reprise à des fins d'équilibrage de charge et de reconfiguration dynamique. Nous avons conçu l'ensemble de ces mécanismes avec le soucis de l'efficacité. En outre, nous nous sommes attachés d'une part à factoriser le code mettant en œuvre les fonctionnalités communes à plusieurs mécanismes et d'autre part à réutiliser chaque fois que cela est possible les fonctions offertes par le noyau LINUX. Cette démarche nous a conduit à introduire le concept de processus fantôme autour duquel sont bâtis les mécanismes de duplication, migration et sauvegarde/restauration de points de reprise et à minimiser les modifications du noyau LINUX.

Un autre aspect important de notre travail de conception de mécanismes de gestion globale des processus est l'intégration de ces derniers au sein du SSI KERRIGHED. Nous avons cherché à exploiter au mieux le concept de conteneur pour la gestion de l'espace d'adressage des processus.

Nous présentons le concept de processus fantôme dans le paragraphe 8.1 et montrons dans le paragraphe 8.2 son utilisation pour la mise en œuvre des différents mécanismes. Nous terminons par un résumé dans le paragraphe 8.3.

8.1 Le concept de processus fantôme

8.1.1 Introduction

La majorité des mécanismes pour manipuler les processus nécessitent de manipuler des *clones* de processus indépendamment de leur localisation.

> Définition 29 (Processus clone) *Un processus clone est un processus actif identique à un processus donné s'exécutant au sein de la grappe.*

Par exemple, la migration consiste en la création d'un processus clone sur une machine distante et en l'élimination du processus sur le nœud d'origine. Pour suspendre un processus, son image est stockée pour permettre de créer un clone s'il redevient actif ultérieure-

ment, puis le processus est éliminé du système. Certaines créations de processus, comme lors d'un *fork*, consistent à dupliquer un processus. L'ensemble des données nécessaires à la création d'un clone constitue un *processus fantôme*.

Définition 30 (Processus fantôme) *Un processus fantôme est l'ensemble des informations relatives à un processus et qui permettent d'en créer un clone sur un nœud quelconque. Une propriété fondamentale d'un processus fantôme est qu'il est indépendant de tout nœud de la grappe et qu'il peut être rangé sur un support quelconque (disque, mémoire, réseau). Un processus fantôme est une entité inactive.*

Le concept de processus fantôme permet de mettre en œuvre tous les mécanismes de manipulation de processus nécessitant de construire une image de processus, comme la migration, la duplication ou la création et la restauration de points de reprise. Les mécanismes de manipulation de processus comportent deux phases : la construction du processus fantôme (qui est stocké sur une ressource de la grappe), puis son utilisation pour créer un ou plusieurs processus clones (à partir de la lecture des informations du processus fantôme). La phase de construction du processus fantôme est appelée **exportation** du processus fantôme : l'ensemble des informations du processus fantôme sont envoyées vers une ressource. La phase de création d'un clone à partir du processus fantôme est appelée **importation** du processus fantôme : les informations du processus fantôme sont chargées en mémoire puis utilisées pour créer un processus clone. Par exemple, pour une migration de processus, l'exportation correspond à l'envoi de l'image du processus fantôme à travers le réseau, alors que l'importation correspond à la réception de l'image du processus sur le nœud distant. Pour la suspension d'un processus avec stockage en mémoire, l'exportation est le stockage dans une zone mémoire de la grappe, alors que l'importation est la lecture du processus fantôme depuis cette zone mémoire.

8.1.2 Architecture

Le processus fantôme permet de fournir une interface unique pour les phases d'exportation et d'importation : il définit l'ensemble des données nécessaires pour la création d'un processus clone et le type d'accès aux ressources. Le concept de processus fantôme définit la liste des informations noyau permettant de créer un processus clone et d'autre part fournit l'accès à diverses ressources (actuellement le réseau, la mémoire, le système de fichiers). En associant un accès à une ressource avec la liste des informations système définissant le processus fantôme, on crée de nouveaux mécanismes de manipulation de processus.

La liste des informations noyau constituant le processus fantôme est construite en faisant appel à une fonction (*make_ghost*) permettant de créer un fantôme des différentes informations système (voir figure 8.1). Cette fonction permet de définir les données à traiter en faisant abstraction de la ressource utilisée pour les stocker. Cette approche permet de simplifier la maintenance du mécanisme de processus fantôme : toute modification de la liste des informations constituant le processus fantôme est utilisée de façon transparente par tous les mécanismes de manipulation de processus. On peut donc voir un processus fantôme comme un composant opaque, dont le but est de stocker les informations du processus

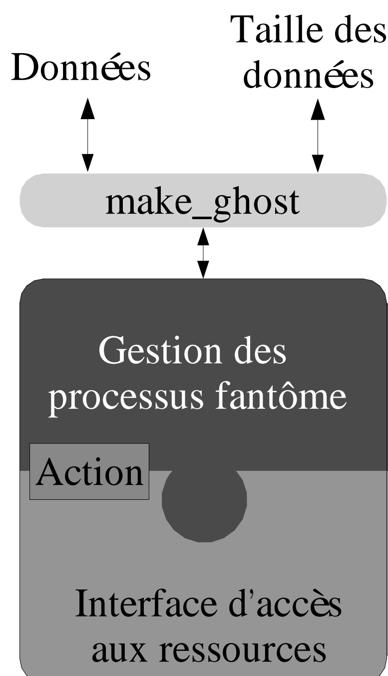

FIG. 8.1 – Architecture d'un processus fantôme

et de cacher l'accès à la ressource associée à ces données.

Le type d'accès aux ressources est défini de manière simple par l'**action** associée au processus fantôme (modifiable par une interface de type *set_ ghost_ action*). Cette interface nécessite bien sûr que dans le mécanisme de processus fantôme, une action soit associée à un accès à une ressource donnée. Cette association est effectuée via l'interface d'accès aux différentes ressources. Par exemple, pour la migration de processus, le mécanisme de processus fantôme définit l'association entre migration et écriture/lecture réseau.

En plus du mécanisme de processus fantôme, il est nécessaire de disposer de gestionnaires propres à chaque mécanisme de manipulation de processus permettant d'effectuer toutes les actions au sein du système qui leur sont propres. Par exemple, pour le mécanisme de migration de processus, les actions effectuées au sein du système ne sont pas les mêmes que lors de la duplication de processus (le processus initial est détruit dans le premier cas et pas dans le second). Trois gestionnaires sont donc disponibles : un gestionnaire de migration (qui gère également les duplications de processus), un gestionnaire de points de reprise et un gestionnaire de restauration de points de reprise. Chacun de ces gestionnaires est chargé d'effectuer des tâches spécifiques au sein des systèmes des nœuds impliqués par la manipulation d'un processus (*e.g.* initialisation ou mise à jour de données des processus clones créés à partir d'un processus fantôme, modification d'informations du noyau). Les traitements pré-exportation et post-importation effectués par ces gestionnaires permettent donc de mettre en œuvre les différences entre les divers mécanismes de manipulation de processus.

8.1.2.1 Interfaces d'accès aux ressources

La définition des données constituant le processus fantôme et l'importation/exportation de ces données sont deux mécanismes indépendants. L'importation/exportation des données de processus fantômes est effectuée à travers une interface d'accès aux différentes ressources de la grappe. Pour le moment, dans le système KERRIGHED, il est possible d'accéder aux disques locaux, au réseau et à la mémoire locale. Le processus fantôme peut donc indifféremment utiliser ces trois ressources. L'utilisation d'autres ressources peut être définie au sein du mécanisme de processus fantôme. Pour cela, il suffit de spécifier l'interface d'accès à la ressource que l'on souhaite utiliser pour l'importation et l'exportation du processus fantôme. Il est par exemple envisagé d'ajouter un accès aux disques distants et à la mémoire distante en utilisant les mécanismes de fédération des ressources disponibles dans KERRIGHED.

Le type d'accès aux ressources associé au processus fantôme est défini par l'*action* de celui-ci (voir algorithme 6), cette action étant propre à la sémantique du service que l'on souhaite mettre en œuvre à travers l'utilisation du processus fantôme. Plusieurs types d'action sont disponibles dans KERRIGHED :

- pour l'accès au réseau, en utilisant le système de communication noyau mis en place en sein du système KERRIGHED, ces informations regroupent le numéro de nœud du destinataire, le numéro de port utilisé, ainsi que que le type de canal utilisé.
- pour l'accès au système de fichier local, la définition de l'accès physique est constituée uniquement de la structure noyau représentant le fichier utilisé. Cette structure noyau regroupe l'ensemble des informations nécessaires pour lire et écrire sur le fichier.
- pour l'accès à la mémoire, en utilisant le gestionnaire de zones tampons du système KERRIGHED, l'accès est réalisé par l'intermédiaire de la structure noyau représentant la zone de mémoire tampon qui regroupe toutes les informations nécessaires pour lire et écrire en mémoire.

Pour définir l'action associée à un processus fantôme, celui-ci possède un champ générique permettant de définir l'accès physique à la ressource utilisée. Cette approche extrêmement souple permet d'utiliser tout type de ressource sans contrainte d'interface (une interface lecture/écriture n'est pas indispensable) puisque les accès sont définis pour chaque cas.

8.1.2.1.1 Utilisation du système de fichier local L'accès à un fichier s'effectue en quatre étapes : ouverture du fichier, association du fichier à l'interface d'accès aux ressources du processus fantôme, définition de l'action du processus fantôme, traitement (lecture ou écriture) des données (voir les algorithmes 7 et 8). Les trois premières étapes sont effectuées durant l'initialisation, mais il est également possible de modifier le type d'accès pendant la manipulation d'un processus fantôme. La possibilité de modifier le type d'accès pendant la manipulation d'un processus fantôme permet par exemple de stocker les informations du processus fantôme dans plusieurs fichiers. Il est ainsi possible pour des threads de stocker les informations propres à chaque thread dans un fichier et l'espace d'adressage (partagé entre les threads) dans un second fichier, facilitant la mise en œuvre de la restauration de la MPR utilisée par les threads.

Algorithme 6 :Algorithme d'accès aux ressources lors de la création d'un processus fantôme

Accès au processus fantôme (données, taille des données) :
{
  action := *get_ghost_action* ;
  selon action
    cas SEND_GHOST_TASK :
      Envoi_réseau (données, taille des données) ;
    cas RECV_GHOST_TASK :
      Réception_réseau (données, taille des données) ;
    cas SAVE_GHOST_TASK :
      Ecriture_disque (données, taille des données) ;
    cas LOAD_GHOST_TASK :
      Lecture_disque (données, taille des données) ;
    cas BUFF_SAVE :
      Ecriture_mémoire (données, taille des données) ;
    cas BUFF_LOAD :
      Lecture_mémoire (données, taille des données) ;
    }

8.1.2.1.2 Utilisation de la mémoire locale Le processus fantôme peut être construit en mémoire, en utilisant le mécanisme de manipulation de zones mémoires de KERRIGHED. Ce mécanisme permet d'allouer en mémoire une zone de taille donnée, puis d'y écrire et d'y lire des données. Pour utiliser ce mécanisme, il est nécessaire de connaître à priori la taille du processus fantôme. Pour cela, une interface d'accès aux ressources a spécialement été créée : aucun accès à une ressource n'est effectué, mais un compteur représentant la taille du processus fantôme est incrémenté de la taille de la donnée couramment traitée. Il est ainsi possible par une construction *virtuelle* du processus fantôme de connaître sa taille. Il est ensuite aisé d'allouer une zone mémoire équivalente pour y construire ensuite effectivement le processus fantôme (voir algorithme 9).

8.1.2.1.3 Utilisation des interfaces réseau locales L'accès au réseau s'effectue via le système de communication haute performance de KERRIGHED. Ce système de communication permet d'établir des communications entre nœuds de la grappe en utilisant divers types de réseaux tels que Ethernet ou Myrinet de façon transparente. Toutes les communications s'effectuent via un port et un canal.

L'accès direct au réseau s'effectue en trois étapes : association du port, du canal et de l'identifiant du destinataire au processus fantôme, définition de l'action du processus fantôme, traitement (lecture ou écriture) des données. Les deux premières étapes sont effectuées durant l'initialisation (voir les algorithmes 10 et 11).

Algorithme 7 :Algorithme de création d'un processus fantôme en utilisant le système de fichier local

Création d'un processus fantôme en utilisant le système de fichier local :
{
fichier = *ouverture_fichier* (nom_fichier) ;
initialise_process_fantôme () ;
set_ghost_file_infos (fichier) ;
set_ghost_process_action (SAVE_GHOST_TASK) ;
export_process () ;
}

Algorithme 8 :Algorithme de chargement d'un processus fantôme en utilisant le système de fichier local

Chargement d'un processus fantôme en utilisant le système de fichier local :
{
fichier = *ouverture_fichier* (nom_fichier) ;
initialisation_process_fantôme () ;
set_ghost_file_infos (fichier) ;
set_ghost_process_action (LOAD_GHOST_TASK) ;
import_process () ;
}

8.1.2.1.4 Résumé Le même mécanisme de manipulation de processus fantôme permet donc de mettre en œuvre des mécanismes de manipulation globale de processus très différents tels que la migration de processus, la création de point de reprise de processus en mémoire et sur fichier.

Le mécanisme de manipulation de processus fantôme permet également de mettre en œuvre simplement l'accès à d'autres ressources, par extension de l'interface d'accès aux ressources, comme par exemple un système de fichier ou la mémoire locale d'un autre nœud.

8.1.2.2 Manipulation des processus fantômes

Afin de faciliter la manipulation des processus fantômes, une interface d'importation/-exportation est disponible.

La fonction *export_process* permet de créer un processus fantôme à partir d'un processus donné. Bien sûr, la ressource utilisée pour stocker ce processus fantôme varie en fonction de l'action qui lui est associée.

La fonction *import_process* permet de charger le processus fantôme et d'en créer un processus clone.

Algorithme 9 :Algorithme de création du processus fantôme en mémoire

Création du processus fantôme en mémoire :
{
initialisation_process_fantôme () ;
set_ghost_process_action (COMPUTE_GHOST_SIZE) ;
export_process () ;
taille = *get_ghost_size* () ;
buffer = *allocation_buffer_kerrighed* (taille) ;
set_ghost_buff_infos (buffer) ;
set_ghost_process_action (BUFF_WRITE) ;
export_process () ;
}

Algorithme 10 :Algorithme de création du processus fantôme en utilisant l'interface d'accès au réseau

Création du processus fantôme via le réseau(port, canal, destinataire) :
{
initialisation_process_fantôme () ;
set_ghost_net_comm_info (port, canal, destinataire) ;
set_ghost_process_action (SEND_GHOST_TASK) ;
export_process () ;
}

Ces deux fonctions servent donc de fonctionnalité de base pour la mise en œuvre de mécanismes de manipulation globale de processus comme la migration ou la duplication.

8.1.3 Création du processus fantôme

La création d'un processus fantôme est constituée de plusieurs étapes :

1. l'extraction du système des informations concernant l'espace d'adressage du processus,
2. l'extraction du système des informations concernant les données privées du processus (fichiers ouverts, données propres au processus comme son identifiant),
3. la gestion d'un identifiant unique à l'échelle de la grappe afin de garantir la transparence de localisation,
4. la gestion des signaux à l'échelle de la grappe pour garantir la transparence à la localisation.

Les deux derniers points sont intéressants non pas pour la mise en œuvre des mécanismes de manipulation de processus eux-mêmes, mais pour garantir une continuité des mécanismes

Algorithme 11 :Algorithme de chargement du processus fantôme en utilisant l'interface d'accès au réseau

Chargement du processus fantôme via le réseau (port, canal, destinataire) :
{
initialisation_process_fantôme () ;
set_ghost_net_comm_info (port, canal, destinataire) ;
set_ghost_process_action (RECV_GHOST_TASK) ;
import_process () ;
}

système des nœuds comme l'identification des processus et la gestion des signaux même après déplacement de processus vers d'autres nœuds.

8.1.3.1 Extraction de l'espace d'adressage

Un processus est constitué d'un ensemble de segments mémoire. Lors de la création d'un processus fantôme associé, ces données doivent être extraites. Deux cas sont à considérer : le processus partage des segments mémoire avec d'autre processus s'exécutant au sein de la grappe grâce aux mécanismes des conteneurs (voir paragraphe 5.3) du système KERRIGHED, soit le processus est un processus Linux standard.

Un processus partageant des segments mémoires avec d'autres processus au sein de la grappe grâce au mécanisme des conteneurs peut également avoir des segments mémoire non partagés. Dans ce cas, l'extraction du processus est effectuée par extraction des informations sur les conteneurs pour les segments mémoire partagés, et par extraction complète des segments mémoire non partagés.

8.1.3.1.1 Extraction sans utilisation des conteneurs Pour extraire l'espace d'adressage d'un processus n'utilisant pas les conteneurs, il faut extraire les segments mémoire et les pages mémoires associées.

Coût de l'extraction d'un segment mémoire non lié à un conteneur Soit C_1 le coût d'extraction des données propres au segment mémoire. Soit p_i le nombre de pages du segment i. Soit c_p le coût d'extraction d'une page mémoire. Le coût d'extraction E_i d'un segment mémoire i non lié à un conteneur est donné par :

$$E_i = C_1 + p_i \times c_p$$

Coût de l'extraction des segments mémoire d'un processus n'utilisant pas de conteneurs Soit v le nombre de segments de mémoire n'utilisant pas de conteneur. Soit E'_0 le coût de l'extraction des segments mémoire d'un processus n'utilisant pas de conteneurs.

$$E'_0 = \sum_{i=0}^{i=v} E_i = \sum_{i=0}^{i=v} (C_1 + p_i \times c_p) = v \times C_1 + \sum_{i=0}^{i=v} (p_i \times c_p)$$

8.1.3.1.2 Extraction avec utilisation des conteneurs En dehors du fait que les conteneurs KERRIGHED permettent de partager de la mémoire entre processus au sein de la grappe, les conteneurs présentent deux principaux avantages pour le mécanisme des processus fantômes : il est possible d'étendre les mécanismes de gestion globale des processus aux threads[1], et il est possible de transférer les pages mémoire à la demande (gain de performance).

Pour illustrer cela, prenons le cas de la migration de processus. Comme nous l'avons vu dans le paragraphe 8.1.3.1.1, si un segment mémoire n'est pas lié à un conteneur, l'ensemble des informations de ce segment mémoire (pages de mémoire physique, structures de donnée regroupant les informations du segment) doivent être extraites lors de la création du processus fantôme. Avec les conteneurs, l'extraction des segments mémoire est beaucoup plus simple : il suffit d'extraire les informations nécessaires pour pouvoir lier le processus au conteneur après sa migration (voir figure 8.2). On peut alors voir la migration comme un glissement le long des conteneurs : toutes les informations des segments mémoire liés à un conteneur sont gérées par le système des conteneurs de KERRIGHED, elles ne sont donc pas à traiter lors de la migration et les pages mémoire nécessaires à l'exécution du processus après la migration sont déplacées à la demande.

Le mécanisme des conteneurs simplifie donc grandement la création des processus fantôme tout en permettant la migration des informations mémoire du processus à la demande.

Coût de l'extraction d'un segment mémoire lié à un conteneur Soit C_1 le coût d'extraction des données propres au segment mémoire. Soit C_c le coût d'extraction des données du conteneur. Soit E_1 le coût de l'extraction d'un segment mémoire lié à un conteneur.

$$E_1 = C_1 + C_c$$

C_1 est constant : ces données ne dépendent pas de la taille du segment mémoire. C_c est constant. Les données du conteneur regroupent en effet le numéro du conteneur et sa position courante associée.

Le coût lors de l'extraction est donc très faible, mais une partie de ce coût est reporté lors de l'accès aux pages mémoire (migration des pages à la demande).

Coût de l'extraction des segments mémoire d'un processus utilisant les conteneurs Soit V_c le nombre de segments mémoire utilisant de conteneurs. Soit E'_1 le coût de l'extraction des segments mémoire d'un processus utilisant les conteneurs.

[1]Dans KERRIGHED, comme dans Linux, un thread est mis en œuvre par un processus partageant sa mémoire.

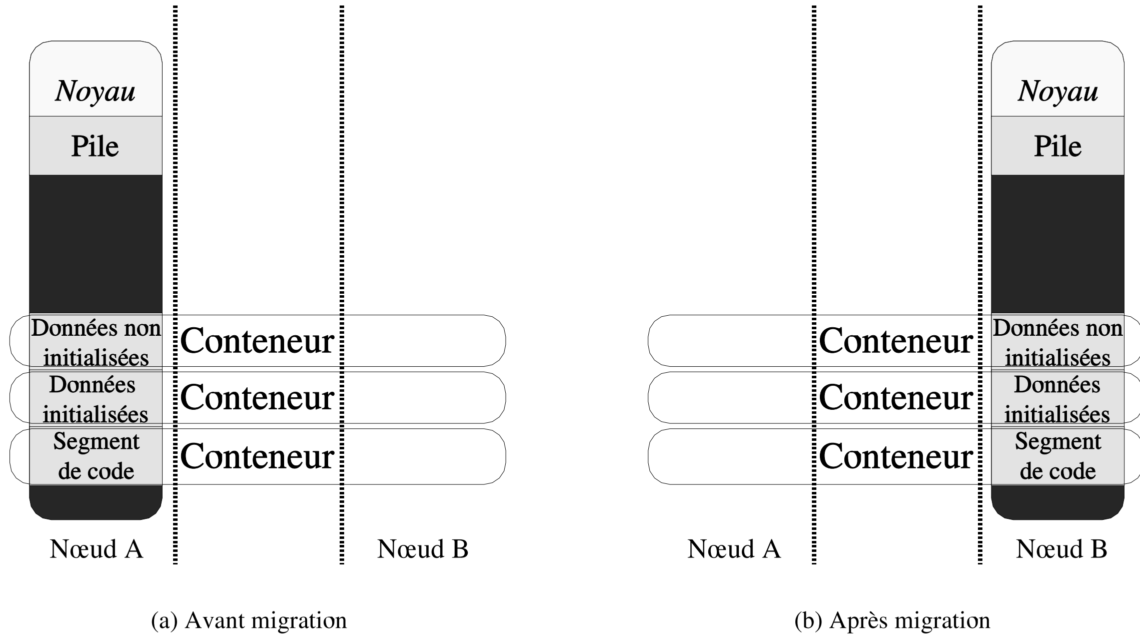

FIG. 8.2 – La migration de processus avec utilisation des conteneurs

$$E'_1 = \sum_{i=0}^{V_c} E_1 = \sum_{i=0}^{V_c}(C_1 + C_c) = V_c(C_1 + C_c)$$

8.1.3.1.3 Coût total Soit C_m le coût d'extraction de l'espace d'adressage d'un processus.

$$C_m = E'_0 + E'_1 = (v \times C_1 + \sum_{i=0}^{i=v}(p_i \times c_p)) + (V_c(C_1 + C_c)) = (v + V_c) \times C_1 + V_c \times C_c + \sum_{i=0}^{i=v}(p_i \times c_p))$$

Il est intéressant de noter que si l'ensemble des segments mémoire sont liés à des conteneurs ($v = 0$), le coût de l'extraction de l'espace d'adressage d'un processus devient alors :

$$(v + V_c) \times C_1 + V_c \times C_c$$

Le coût d'extraction est donc constant, quelque soit le nombre de pages mémoire utilisées.

8.1.3.2 Extraction des données privées du processus

Les données privées d'un processus sont constituées d'une part par l'ensemble de ses données noyau (comme son identifiant (PID), l'identifiant du propriétaire du processus), et d'autre part par la liste de ses fichiers ouverts, des signaux et des registres processeur

qui lui sont associés. La gestion des signaux ne pose pas de problème : la manipulation d'un processus ne peut être effectuée que dans un état garantissant qu'aucun signal n'est en attente de traitement (cela est garanti par la méthode d'insertion des mécanismes de manipulation de processus au sein du système hôte).

La principale tâche pour l'extraction des données privées est donc l'extraction des informations sur les fichiers manipulés par le processus.

8.1.3.2.1 Liste des fichiers ouverts Comme pour l'extraction de l'espace d'adressage du processus, deux cas sont possibles : l'accès aux fichiers se fait par l'utilisation des conteneurs ou pas.

Extraction des informations fichier sans utilisation des conteneurs Une possibilité pour accéder aux fichiers de manière transparente quelque soit la localisation est l'utilisation d'un système de fichier distribué tel que NFS[76]. Avec un tel système de fichiers, il est possible d'accéder depuis tous les nœuds au même système de fichier. Une approche pour la gestion des informations sur les fichiers manipulés par un processus est donc d'extraire du noyau l'ensemble des informations sur ces fichiers distribués et de les inclure dans le processus fantôme. Ainsi, lors de la création d'un processus clone à partir du processus fantôme, il est possible de récupérer l'ensemble des données sur les fichiers permettant ainsi de réouvrir les fichiers dans le contexte du processus clone.

Extraction des informations fichier avec utilisation des conteneurs Comme pour l'extraction des segments mémoire, la gestion des informations sur les fichiers manipulés par le processus est rendue plus simple par l'utilisation des conteneurs. En effet, il n'est alors pas nécessaire de gérer l'ensemble des données noyau sur les fichiers afin de procéder à une réouverture de ceux-ci lors de la création d'un processus clone. Il suffit de stocker au sein du processus fantôme les informations sur les conteneurs utilisés (*i.e.* le numéro du conteneur et l'offset utilisé pour y accéder). Ces informations, lors de la création de processus clones permettront de lier le processus aux conteneurs liés à des fichiers, garantissant ainsi un accès transparent aux fichiers.

8.1.3.2.2 Extraction de la valeur des registres processeur L'une des principales difficultés pour obtenir les informations nécessaires à la création d'un processus clone est l'obtention de la valeur des registres processeur associés au processus. En effet, ces valeurs ne sont disponibles que dans des états précis du noyau, comme par exemple, lors du traitement des signaux, et ne sont sauvegardées et restaurées qu'à un niveau très bas dans le noyau.

De plus, les mécanismes de manipulation des processus comme la migration doivent pouvoir être déclenchés à tout moment comme pour les signaux.

Le comportement des mécanismes de gestion globale des processus est donc très proche de celui des signaux. Un nouvel état a été créé dans le système, similaire à celui pour le traitement des signaux, afin de déclencher l'ensemble des mécanismes de manipulation des

processus. Cette modification est enfouie au cœur du système, mais elle est très légère et commune à l'ensemble des mécanismes de gestion des processus, ce qui semble être un bon compromis entre modification noyau, maintenabilité et complexité.

8.1.3.2.3 Extraction des signaux propres au processus Il n'est pas nécessaire d'effectuer de traitement particulier lors de l'extraction des signaux puisque la création d'un processus fantôme ne peut être effectuée que lorsque tous les signaux en attente du processus ciblé ont été traités. Aucune information particulière n'est donc extraite, les informations nécessaires à la gestion des signaux sont initialisées lors de la création d'un processus clone depuis le processus fantôme.

8.1.3.2.4 Extraction du PID du processus Lors de l'extraction des données privées d'un processus, son PID est inclus dans le processus fantôme. Ce PID permettra lors de la création d'un processus clone de disposer de l'identifiant unique du processus au sein du système source. Cet identifiant est stocké avec le numéro du nœud source du processus. Ces deux informations constituent les seules informations nécessaires pour que le nœud source puisse effectuer des traitements à la place du processus initial. Cela permet par exemple de prévenir le processus père d'un processus lorsqu'un processus migré se termine.

Le paragraphe suivant montre comment le système KERRIGHED permet de gérer un identifiant global de processus au sein de la grappe permettant de mettre en place des systèmes distribués gérant les processus indépendamment de leur localisation.

8.1.4 Gestion globale d'un identifiant unique de processus

Pour pouvoir gérer les processus indépendamment de leur localisation, il est nécessaire de disposer d'un mécanisme d'identification unique des processus à l'échelle de la grappe. Nous avons donc introduit dans le système KERRIGHED la notion de *processus Kerrighed*.

Définition 31 (Processus KERRIGHED) *Un processus* KERRIGHED *est un processus standard qui reçoit un identificateur unique au sein de la grappe.*

Définition 32 (KPID) *Un identificateur unique de processus dans la grappe, appelé KPID, est constitué du numéro du nœud de création et de l'identificateur de thread (cet identificateur est égal à 0 pour les processus traditionnels).*

Cet identificateur, associé à des gestionnaires de *KPID*, permet de pouvoir contacter tout processus KERRIGHED quelque soit sa localisation dans la grappe.

Pour appliquer une fonctionnalité sur un processus ou un thread (*e.g.* communications entre threads), le gestionnaire de *KPID* envoie à tous les nœuds le *KPID* du processus concerné et l'identifiant de la fonctionnalité à appliquer. Sur un nœud, lorsqu'une telle requête arrive, le gestionnaire de KPID recherche localement le processus ou le thread. Si ce processus ou ce thread est présent, la fonctionnalité est appliquée grâce à son identifiant.

8.1.5 Gestion globale des signaux associés aux processus

Les signaux, dans le noyau Linux, sont prévus uniquement pour fonctionner avec des processus en cours d'exécution localement. Ce mécanisme doit donc être étendu pour fonctionner avec tout processus, indépendamment de sa localisation. Pour cela, deux approches sont possibles :

1. étendre le mécanisme existant (table des processus migrés avec redirection des signaux),
2. utiliser la gestion globale des flux de données au sein de la grappe.

Le système KERRIGHED a pour but de gérer globalement l'ensemble des ressources disponibles au sein de la grappe, dont les flux de données. Cette gestion globale permet d'étendre de façon transparente la gestion des signaux à l'échelle de la grappe. Ce travail est effectué par Pascal Gallard dans le cadre de sa thèse mais n'est pas suffisamment abouti pour être utilisé dans le cadre de mes travaux. Afin de pouvoir offrir une gestion transparente des signaux, un mécanisme de transfert des signaux a donc été mis en place dans le cadre de la gestion globale des processus. Ce mécanisme est fondé sur des gestionnaires de signaux s'exécutant sur chaque nœud de la grappe. Le fonctionnement du gestionnaire de signaux est fondé sur une table des processus migrés. Cette table comporte deux informations : la liste des processus qui ont migré vers d'autres nœuds, et la liste des processus migrés qui s'exécutent localement. Ce gestionnaire est appelé à chaque envoi de signal, et grâce à sa table, fait suivre les signaux si cela s'avère nécessaire.

8.2 Déploiement, migration et création de points de reprise de processus

Nous montrons comment le concept de processus fantôme est exploité pour la mise en œuvre des mécanismes de duplication, de migration et de gestion de points de reprise.

8.2.1 Déploiement de tâches

La création à distance de processus permet de placer un processus à sa création sur un nœud donné, contrairement à la migration qui est effectuée durant l'exécution du processus.

Deux types de création distante de processus peuvent être considérés :

1. la *duplication* d'un processus à partir d'un processus père (*e.g.* comme lors de l'utilisation d'une interface de programmation de type *fork* ou *pthread*),
2. la *création distante* d'un processus lors du lancement d'une tâche.

La principale différence entre ces deux types de création de processus est leur coût. Alors que pour la création distante de processus, le coût est relativement limité (peu d'informations sont à envoyer), la duplication est aussi coûteuse que la migration puisque l'image du processus père doit être transférée afin de créer le nouveau processus.

8.2.1.1 Duplication de processus

Le mécanisme de *processus fantôme* permet de créer un clone d'un processus indépendamment de la localisation. Il est donc très simple de créer un mécanisme de duplication de processus. Deux mécanismes s'appuyant sur la duplication ont été étudiés : le support d'une interface *pthread* et d'une interface du type *fork* (voir figure 8.3). La duplication de pro-

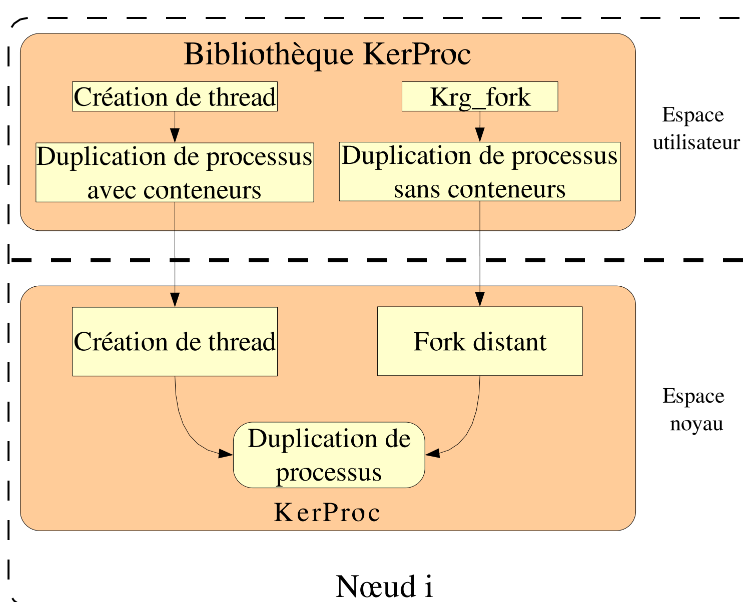

FIG. 8.3 – Architecture logicielle pour la création de processus et de threads dans Kerrighed

cessus est similaire à la migration de processus puisque dans les deux cas, il est nécessaire de transférer l'image du processus (processus fantôme) initial vers le nœud distant.

8.2.1.1.1 Interface fork Une interface standard pour la création de processus à distance est l'utilisation d'une interface de type « *fork* ». Voici un exemple type de l'utilisation du « *fork* » dans un système Unix traditionnel.

```
n = fork ();
if (n == 0)
printf ("Processus fils\n");
else
printf ("Processus père\n");
```

Cette interface permet de créer un nouveau processus en dupliquant le processus père. Après la duplication, le processus fils démarre son exécution juste après l'endroit où le processus père a effectué sa duplication (durant la duplication, la pile et les registres processeur sont dupliqués, la prochaine instruction à exécuter, qui est contenue dans le registre *eip*, est donc la même pour le processus père et fils). La principale différence entre le père

et le fils est que le père a pour valeur de retour le PID du fils alors que la valeur de retour est égale à zéro pour le fils.

Pour offrir un « *fork* » à l'échelle de la grappe, il est nécessaire de modifier légèrement cette interface. En effet, la valeur de retour d'une fonction de type « *fork* » pour la grappe ne peut pas être constituée uniquement du PID, celui-ci ne permettant pas d'identifier de manière unique un processus au sein de la grappe. Le système KERRIGHED dispose du mécanisme de KPID permettant l'identification unique de processus au sein de la grappe. Le KPID et le mécanisme de duplication permettent donc d'offrir simplement une interface de type « *fork* » au sein de la grappe que l'on appelera « *krg_fork* ». Dans ce cas, la valeur de retour de la fonction « *krg_fork* » est le KPID du processus fils pour le processus père, et est égale à zéro pour le processus fils. Pour cela, une interface utilisateur a été mise en œuvre.

8.2.1.1.2 Interface Pthread Les mécanismes de création distante de processus ont permis de mettre en œuvre une interface *pthread* sur grappe. Typiquement, les mécanismes de base qu'une interface pthread doit offrir sont :

1. création de thread,
2. partage de la mémoire,
3. terminaison de thread,
4. communication entre threads (synchronisation, signaux).

Une partie de ces mécanismes sont déjà mis en œuvre dans des services du système KERRIGHED en dehors de KERPROC. Le service KERSYNC met en œuvre les mécanismes de verrous, de barrières et de conditions. Le service KERMEM permet de partager de la mémoire entre processus. Or, les threads étant mis en œuvre au sein du noyau LINUX traditionnel par des processus partageant leurs segments mémoire, il est simple d'offrir un mécanisme de threads au sein de KERRIGHED grâce aux conteneurs mémoire.

Enfin KERRIGHED offrant des mécanismes d'identification unique des processus au sein de la grappe et de gestion globale des processus, il est simple d'étendre ces mécanismes pour répondre aux besoins de la mise en œuvre d'une interface *pthread* en terme d'identification de thread et de communication par signaux.

En résumé, l'approche système à image unique permet d'offrir rapidement une interface *pthread* au sein de la grappe par simple extension des mécanismes KERRIGHED déjà existants.

8.2.1.1.3 Support de OpenMP La mise en œuvre d'une interface *pthread* a permis de supporter facilement l'exécution d'applications OPENMP. Pour cela, l'approche consiste à utiliser un compilateur OPENMP qui cible les *pthreads* : le programme OPENMP est transformé en programme *pthread* qui peut alors s'exécuter sur KERRIGHED [57].

Cette approche a pour principal avantage de ne pas limiter le support d'OPENMP sur KERRIGHED a un seul compilateur, tous les compilateurs générant un code *pthread* pouvant être utilisés. Un autre intérêt de cette approche est que le support d'OPENMP est complet,

contrairement à bon nombre d'autres systèmes où toutes les directives OpenMP ne sont pas supportées. En revanche, l'inconvénient de cette approche est que les performances sont très dépendantes du compilateur utilisé.

8.2.1.2 Création distante de processus

Le mécanisme de création distante de processus est indispensable pour le lancement d'applications séquentielles. Contrairement à la duplication de processus, la création distante de processus ne nécessite pas de transférer l'image du processus père.

Dans un système traditionnel tel que Linux, lors du lancement d'une application séquentielle, le processus courant est dupliqué, le nouveau processus créé étant ensuite remplacé puis initialisé avec le code de l'application lancée (fonctionnement du type « *fork/exec* »). On voit donc ici que lorsque l'on souhaite lancer une application, seuls une référence au fichier source et les éventuels paramètres de lancement suffisent.

Pour la création distante de processus au sein d'une grappe de calculateurs, la démarche est similaire : on peut lancer une application séquentielle en envoyant une référence sur le fichier contenant le programme objet et les éventuels paramètres de lancement sur le nœud distant qui hébergera le nouveau processus. Dans ce cas, il n'est pas nécessaire de transférer l'image complète du processus père, celle-ci étant inutile.

Deux approches sont donc possibles : une approche en mode utilisateur et une approche en mode noyau.

L'approche utilisateur permet d'utiliser les outils existants dans le système Linux tel que « *rsh* » ou « *ssh* » pour lancer des applications sur des nœuds distants avec possibilité de fournir des paramètres de lancement. Avec cette approche, il faut dans un premier temps questionner l'ordonnanceur global pour obtenir l'identifiant d'un nœud distant pouvant accueillir l'application à lancer. Lorsque cet identifiant est obtenu, il est alors possible d'envoyer une requête de création de processus sur le nœud distant en utilisant un outil tel que « *rsh* » ou « *ssh* ». Ces outils fonctionnent tous de manière similaire : lorsqu'une requête est reçue sur un nœud, le processus courant est cloné pour être ensuite remplacé par l'application à lancer. On se rend donc compte que cette approche a pour principal inconvénient de faire de multiples requêtes de l'espace utilisateur vers le noyau (lors du questionnement de l'ordonnanceur global et lors de la création d'un processus sur le nœud distant), introduisant à chaque fois un coût.

L'approche noyau évite ces aller-retours entre espace utilisateur et espace noyau : une requête de création distante est envoyée à l'ordonnanceur global qui fait suivre cette requête au nœud cible. Lorsque la requête arrive sur le nœud distant, un gestionnaire noyau de création distante de processus crée directement un nouveau processus.

Le système Kerrighed offre ces deux approches.

8.2.2 Migration de processus

La migration de processus peut être effectuée suivant deux approches différentes, approches largement utilisées dans de nombreux systèmes. Dans une première approche, la

migration de processus peut être effectuée par l'utilisation explicite du réseau, c'est le cas dans OPENMOSIX et GENESIS. Les données du processus nécessaires à la migration sont réceptionnées sur le nœud cible, puis un processus clone est créé.

Une autre approche est d'effectuer la migration de processus en utilisant le système de fichiers, comme dans les systèmes EPCKPT et CRAK. Dans ce cas, la migration de processus tire profit d'un système de fichier distribué : l'image du processus (le processus fantôme dans KERRIGHED) est stocké dans un fichier accessible depuis tous les nœuds. Il est donc aisé de créer un processus clone en utilisant cette image, opérant ainsi une migration.

Dans le système KERRIGHED, la première approche a initialement été adoptée pour garantir un maximum d'efficacité[87]. Par la suite, la seconde approche a été proposée mais l'approche par accès direct au réseau reste la méthode utilisée par défaut pour les mécanismes de migration de processus.

8.2.2.1 Migration par utilisation explicite du réseau

Deux approches sont possibles pour déplacer un processus par utilisation explicite du réseau :

1. Les éléments du processus fantôme sont transférés au fur et à mesure vers le réseau (plusieurs messages sont envoyés du nœud source au nœud de destination).
2. Les éléments du processus fantôme sont rassemblés en mémoire puis envoyés en un seul message vers le réseau.

Quelque soit l'approche adoptée, la migration suit toujours le même algorithme : lorsque le processus clone est créé sur le nœud distant, le processus initial est arrêté.

Accès direct au réseau Pour cette approche, lors de la construction du processus fantôme, chaque élément est envoyé tour à tour via le réseau (voir algorithme 12 et figure 8.4). Cette technique d'accès au réseau a pour principal défaut que chaque élément du proces-

Algorithme 12 :Algorithme de migration avec envoi direct des données sur le réseau

Migration par envoi direct sur le réseau :
{
initialisation_en_écriture_de_l'interface_d'accès_au_réseau (port, canal, destinataire) ;
export_process () ;
}

sus fantôme génère un accès au réseau, chacun impliquant une latence (variable suivant le type de réseau utilisé). Or, le processus fantôme est constitué d'un nombre important de données noyau, générant donc un nombre important d'accès au réseau. Ces multiples accès peuvent donc créer une latence importante pour le traitement du processus fantôme sur certains types de réseau.

FIG. 8.4 – Migration de processus

Accès au réseau après avoir construit le processus fantôme en mémoire Pour éviter de cumuler les latences réseau en effectuant beaucoup d'accès réseau, une solution est de créer tout d'abord le processus fantôme en mémoire. Pour cela, il suffit de créer le processus fantôme dans la mémoire locale (voir paragraphe 8.1.2.1.2) puis d'envoyer cet élément via le réseau (voir algorithme 13).

Algorithme 13 :Algorithme de migration avec création du processus fantôme en mémoire locale

---

Migration avec processus fantôme en mémoire locale :
{
    buffer = *création_du_processus_fantôme_en_mémoire* () ;
    initialisation_en_écriture_de_l'interface_d'accès_au_réseau (port, canal, destinataire) ;
    make_ghost (buffer) ;
}

La souplesse du mécanisme des processus fantômes permet donc très simplement de mettre en œuvre un mécanisme de migration qui, avec certains types de réseau comme Ethernet, est plus performant.

8.2.2.2 Migration par le système de fichier

La migration en utilisant le système de fichier est très simple : il suffit de stocker le processus fantôme dans un fichier partagé entre les nœuds de la grappe (voir algorithme 7), puis de créer un processus clone sur le nœud cible en utilisant ce processus fantôme (voir algorithme 8).

8.2.3 Création de points de reprise

La création d'un point de reprise peut être effectuée suivant deux approches : avec stockage en mémoire ou avec stockage sur disque. Ces deux approches ne diffèrent que par la ressource utilisée, la création du processus fantôme et sa manipulation sont donc identiques dans les deux cas, seule l'action associée diffère.

8.2.3.1 En mémoire

L'interface d'accès aux ressources du mécanisme des processus fantômes permet de manipuler le processus fantôme dans la mémoire locale. Il est donc très simple de créer un mécanisme de création de point de reprise en mémoire : il suffit de créer un processus fantôme en mémoire (voir figure 8.5).

FIG. 8.5 – Point de reprise de processus en mémoire

8.2.3.2 Sur disque

Pour créer un point de reprise sur disque, la ressource utilisée est le système de fichier local (voir figure 8.6).

8.2.3.3 Exécution d'un processus à partir d'un point de reprise

Les points de reprise peuvent être utilisés dans deux cas : (i) à la suite d'une défaillance du nœud sur lequel un processus d'exécute ou (ii) un thread d'une application parallèle doit être remis dans un état antérieur pour garantir une cohérence globale de l'application parallèle après une défaillance de l'un des threads[9].

Une fois qu'un point de reprise de processus est créé, il est possible de reprendre l'exécution du processus à partir de ce point de reprise. Pour cela, deux approches sont possibles :

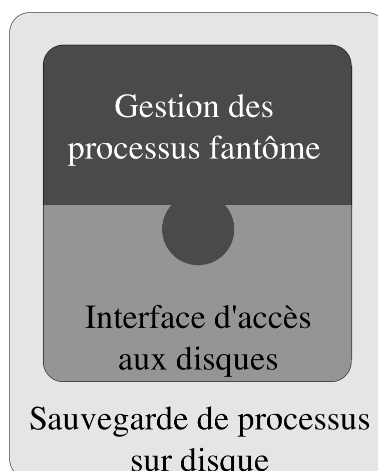

FIG. 8.6 – Point de reprise de processus sur disque

un processus clone est créé (c'est notamment le cas si l'on souhaite reprendre l'exécution sur un nœud différent de celui d'origine), ou bien un retour arrière du processus inital est effectué. Lors d'un retour arrière, le processus que l'on souhaite exécuter à partir d'un point de reprise est recouvert par le processus fantôme.

Dans le cas de la défaillance de l'application ou d'un nœud sur lequel l'application s'exécute, des processus ou threads de l'application peuvent être perdus. Dans ce cas, un nouveau processus ou thread doit être créé en remplacement du processus ou thread perdu par restauration d'un point de reprise.

Dans le cas d'un retour-arrière d'un thread pour garantir la cohérence d'une application parallèle après défaillance d'un thread, le thread existe déjà et doit repartir d'un certain état contenu dans un point de reprise.

8.2.3.3.1 Restauration d'un point de reprise Cette approche est la plus simple à mettre en œuvre car elle utilise directement le mécanisme de processus fantôme pour créer un processus clone. Cette approche présente cependant quelques inconvénients :

- Le nouveau processus est différent du processus initial du point de vue du système (*e.g.* les deux processus n'ont pas le même identifiant) et une mise à jour des informations système est donc nécessaire pour prendre cette différence en compte. Heureusement, les mécanismes de fédération des ressources de KERRIGHED (*e.g.* la gestion globale des signaux) permet de traiter ces modifications de façon transparente.
- Cette méthode est coûteuse car elle procède à la création d'un nouveau processus qui, si le processus initial est localement présent, n'est pas indispensable.

8.2.3.3.2 Retour arrière de processus Dans ce cas, il n'y a pas de création d'un nouveau processus : le processus initial est localement présent et peut donc être recouvert par le point de reprise. Cette approche est donc beaucoup moins coûteuse, aucun nouveau

processus n'est créé. En revanche, cette méthode est plus complexe à mettre en œuvre car la mise à jour transparente des informations système lors de la création d'un processus fantôme doit être effectuée explicitement (*e.g.* réouverture des fichiers utilisés par le processus, création d'une nouvelle pile).

8.2.3.4 Création de points de reprise d'applications parallèles

Un mécanisme de création de points de reprise pour applications parallèles, fondé sur le mécanisme de création de points de reprise de processus, a été mis en œuvre en collaboration avec Ramamurthy Badrinath de l'*Indian Institute of Technology* de Kharagpur (Inde) [10]. De nombreuses méthodes pour la création de points de reprise existent pour les applications parallèles (*e.g.* création coordonnée et non coordonnée de points de reprise)[61].

Ne sachant pas *a priori* quel type d'application est exécuté sur la grappe (*e.g.* applications parallèles par mémoire partagée, applications parallèles par échange de messages), nous avons voulu développer des mécanismes de base permettant ensuite de mettre en œuvre plusieurs stratégies de création de point de reprise pour applications parallèles[8].

Nous avons principalement étudié dans le cadre de nos travaux la problématique de la création de point de reprise pour les applications parallèles à mémoire partagée selon une approche de création coordonnée de points de reprise. Pour les applications parallèles par mémoire partagée, deux points sont importants : (i) être capable de créer un point de reprise de chaque thread de l'application, (ii) être capable de créer un point de reprise pour les pages partagées entre les threads. Le premier point ne pose pas de difficulté, le système de gestion globale des processus permettant, comme nous l'avons vu, de créer un point de reprise d'un thread. La création du point de reprise pour les données partagées est plus complexe, la mémoire partagée de KERRIGHED ne disposant pas *a priori* de mécanismes permettant de créer ce point de reprise.

La création coordonnée de points de reprise s'effectue en trois étapes :

1. coordonner les threads de l'application,
2. créer des points de reprise des informations privées de chaque thread,
3. créer un point de reprise pour les conteneurs utilisés par les threads.

L'architecture du mécanisme de création de points de reprise actuelle est composée d'un coordinateur et d'un serveur de point de reprise (voir figure **??**).

Le coordinateur est chargé de synchroniser les threads lors de la création d'un point de reprise. Une phase de négociation est initiée. Si cette phase de négociation aboutit (*i.e.* tous les threads peuvent se synchroniser pour créer un point de reprise), le coordinateur envoie l'ordre de création de point de reprise aux serveurs de point de reprise de chaque nœud. Ce serveur crée alors un point de reprise de tous les threads en exécution sur le nœud aussi que des pages mémoires physiquement présentes.

8.3 Résumé

Nos travaux ont abouti à proposer dans le système KERRIGHED un système unique permettant de virtualiser des processus, appelé mécanisme de *processus fantôme*, en vue de manipuler les processus indépendamment de leur localisation. Les processus fantôme permettent d'extraire l'image d'un processus pour en créer des clones. La séparation de la définition des informations constituant un processus fantôme et de la définition de la ressource utilisée pour manipuler le processus fantôme permet de créer un ensemble complet de mécanismes de gestion globale de processus.

Le mécanisme de processus fantôme a permis de concevoir et de mettre en œuvre au sein du système KERRIGHED différent mécanismes comme la migration de processus, la duplication ou encore la création/restauration de points de reprise en mémoire ou sur disque. Un mécanisme de création distante de processus, qui n'utilise pas le mécanisme de processus fantôme, est également proposé.

L'ensemble des mécanismes de gestion globale des processus peuvent tirer profit des mécanismes déjà présents dans KERRIGHED. Le mécanisme de conteneur, permettant une gestion globale des blocs au sein de la grappe, est utilisé pour partager des segments mémoire entre processus ou pour bénéficier d'un cache coopératif de fichiers. Avec ce partage de la mémoire entre processus s'exécutant sur différents nœuds de la grappe, les mécanismes de gestion globale des processus ont été étendus à la gestion globale des threads d'applications parallèles.

De même, les systèmes de synchronisation et de gestion globale des flux de données peuvent être utilisés en complément des mécanismes que nous avons conçus.

L'approche retenue dans KERRIGHED reposant sur l'utilisation conjointe de mécanismes de gestion globale des ressources a permis de mettre en œuvre une interface complète *pthread*, interface standard de programmation d'applications parallèles à mémoire partagée.

9 EXEMPLE D'ORDONNANCEUR GLOBAL

Pour illustrer les avantages de notre architecture et le travail à effectuer pour la mise en œuvre d'une nouvelle politique d'ordonnancement, nous détaillons quelques politiques mises en œuvre dans KERRIGHED et destinées à fonctionner sur des nœuds mono-processeur (le module KERPROC ne supporte pas actuellement dans les nœuds multi-processeurs).

Le paragraphe 9.1 présente le prototype de base des fichiers pour la mise en œuvre de composants d'ordonnanceurs globaux. Le paragraphe 9.2 montre la mise en œuvre d'une politique statique fondée sur l'utilisation moyenne des processeurs lors de la dernière minute. Le paragraphe 9.3 montre la mise en œuvre d'une politique dynamique elle aussi fondée sur l'utilisation moyenne des processeurs lors de la dernière minute.

9.1 Prototype des fichiers des composants de l'ordonnanceur global

L'architecture des ordonnanceurs globaux dans KERRIGHED impose des règles pour la mise en œuvre de composants d'ordonnanceur globaux. Ces règles permettent de bénéficier du mécanisme de configuration dynamique et une intégration intégration à l'architecture globale pour la gestion globale des tâches dans KERRIGHED.

La gestion de ces contraintes peut être évitée par l'utilisation de l'interface de programmation que nous avons mis en œuvre et qui fournit le prototype de chaque type de composant. Le programmeur met alors uniquement en œuvre les algorithmes qu'il souhaite implanter.

Afin de mieux comprendre les exemples présentés dans la suite de ce document, nous présentons le prototype des différents composants. Dans le paragraphe 9.1.1, nous détaillons le prototype des fichiers mettant en œuvre une sonde. Le paragraphe 9.1.2 détaille le prototype des fichiers mettant en œuvre un AL. Le paragraphe 9.1.3 détaille le prototype des fichiers mettant en œuvre un GOrG.

9.1.1 Prototype d'une sonde

Le fichier mettant en œuvre une sonde doit être fondé sur le prototype du listing 9.1.

Listing 9.1 – Prototype d'une sonde

```
#include "probe_tools.h"
```

```
MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");

INIT_PROBE

weight_t nom_de_la_sonde (void *arg)
{
}
```

L'inclusion du fichier *probe_tools.h* permet de bénéficier de toutes les macros et primitives facilitant le développement de nouvelles sondes. Parmi ces macros, *INIT_PROBE* permet d'insérer l'ensemble du code propre à un module noyau de façon transparente pour le programmeur.

Disposant de ce squelette de base permettant de créer un module noyau conforme à l'architecture du noyau, le programmeur peut spécifier son nom et une description de la sonde (respectivement les éléments *MODULE_AUTHOR* et *MODULE_DESCRIPTION*). Ces informations optionnelles sont utilisées par le noyau pour la gestion des modules noyau.

La dernière contrainte pour la programmation d'une sonde est le nom de la fonction principale du composant. Ce nom doit être identique au nom du fichier. Par exemple, si le fichier se nomme *ma_sonde.c*, la fonction principale doit s'appeler *ma_sonde*[1].

9.1.2 Prototype d'un analyseur local

Le fichier de mise en œuvre d'un AL doit être fondé sur le prototype du listing 9.2.

Listing 9.2 – Prototype des AL

```
#include "analyzer_tools.h"

MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");
INIT_ANALYZER

int nom_de_l_al (void *arg)
{
  return 0;
}
```

L'inclusion du fichier *analyzer_tools.h* permet de bénéficier de toutes les macros et primitives facilitant la programmation de nouveaux AL. Parmi ces macros, *INIT_ANALYZER* permet d'insérer l'ensemble du code propre à un module noyau de façon transparente pour la programmation.

[1]Une nouvelle version de KerProc est en développement et lève cette contrainte.

Disposant de ce squelette de base permettant de créer un module noyau conforme à l'architecture du noyau, le programmeur peut spécifier son nom et une description de l'AL (respectivement les éléments *MODULE_AUTHOR* et *MODULE_DESCRIPTION*). Ces informations optionnelles sont utilisées par le noyau pour la gestion des modules noyau.

La dernière contrainte pour la programmation d'un AL est le nom de la fonction principale du composant. Ce nom doit être identique au nom du fichier. Par exemple, si le fichier se nomme *mon_AL.c*, la fonction principale doit s'appeler *mon_AL*[2].

9.1.3 Prototype d'un gestionnaire d'ordonnancement global

Le fichier de mise en œuvre d'un GOrG doit être fondé sur le prototype du listing 9.3.

Listing 9.3 – Prototype d'un GOrG

```
#include "scheduler_tools.h"

MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");
INIT_SCHEDULER

int nom_du_gorg (void *arg)
{
  return 0;
}
```

L'inclusion du fichier *scheduler_tools.h* permet de bénéficier de toutes les macros et primitives facilitant le développement de nouveaux GOrG. Parmi ces macros, *INIT_SCHEDULER* permet d'insérer l'ensemble du code propre à un module noyau de façon transparente pour le programmeur.

Une fois ce squelette de base permettant de créer un module noyau conforme à l'architecture du noyau, le programmeur peut spécifier son nom et une description du GOrG (respectivement les éléments *MODULE_AUTHOR* et *MODULE_DESCRIPTION*). Ces informations optionnelles sont utilisées par le noyau pour la gestion des modules noyau.

La dernière contrainte pour le développement d'un GOrG est le nom de la fonction principale du composant. Ce nom doit être identique au nom du fichier. Par exemple, si le fichier se nomme *mon_GOrG.c*, la fonction principale doit s'appeler *mon_GOrG*[3].

9.2 Ordonnanceur statique

Un ordonnanceur simple à mettre en œuvre est un ordonnanceur statique qui répartit les tâches soumises en fonction de la charge processeur des nœuds de la grappe.

[2] Une nouvelle version de KerProc est en développement et lève cette contrainte.
[3] Une nouvelle version de KerProc est en développement et lève cette contrainte.

Le premier composant de cet ordonnanceur est une sonde système pour extraire la charge processeur des nœuds. Dans un souci de simplicité, nous allons utilisé ici les informations qui sont déjà gérées au sein du noyau. Parmi les informations processeur, l'information la plus précise que le noyau gère est la moyenne de la charge processeur durant la dernière minute. Nous allons donc mettre en œuvre une sonde utilisant cette information. Ceci minimise la complexité du développement et les modifications du noyau.

Les informations système extraites doivent être relayées à l'ordonnanceur global. Pour cela, il est nécessaire de mettre en œuvre un AL simple dont le seul rôle est de transmettre les informations qu'il reçoit des sondes vers les GOrG.

Enfin, le dernier composant nécessaire à la mise en œuvre d'un tel composant est un GOrG ayant une politique de placement fondée sur les valeurs de la table contenant les informations processeur des nœuds de la grappe.

Le paragraphe 9.2.1 détaille la mise en œuvre de la sonde extrayant la charge moyenne du processeur lors de la dernière minute. Le paragraphe 9.2.2 détaille la mise en œuvre de l'AL associé à cette sonde et qui est chargé de transmettre les informations système vers le GOrG. Enfin, le paragraphe 9.2.3 détaille la mise en œuvre du GOrG répartissant les tâches à leur création en fonction de la charge processeur des nœuds de la grappe.

9.2.1 Description de la sonde utilisée

La sonde utilisée, par souci de simplicité, est fondée sur la charge moyenne du processeur lors de la dernière minute. Cette information est maintenue par le noyau et contenue dans un tableau nommé *avenrun*. L'accès au tableau *avenrun* ne nécessite qu'une modification mineure du noyau LINUX (exportation des symboles *avenrun*, *LOAD_INT* et *LOAD_FRAC*).

Le code de la sonde est ensuite directement extrait du code contenu dans le noyau LINUX (voir listing 9.4). À partir de la valeur contenue dans le tableau *avenrun*, la charge moyenne est calculée. La valeur de cette charge donne le nombre de processus actifs suivant la formule suivante : $charge = nombre\ de\ processus\ actifs * 100$.

Listing 9.4 – Sonde de la charge moyenne du processeur lors de la dernière minute (probe_loadavg1.c)

```
#include "probe_tools.h"

#include <linux/sched.h>
#include <linux/kernel_stat.h>

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("CPU LoadAverage Probe");
MODULE_LICENSE("GPL");

INIT_PROBE
```

```
#define FSHIFT          11

#define LOAD_INT(x) ((x) >> FSHIFT)
#define LOAD_FRAC(x) LOAD_INT(((x) & (FIXED_1-1)) * 100)

weight_t probe_loadavg1 (void *arg)
{
  int a, load1;

  a = avenrun[0] + (FIXED_1/200);
  load1 = LOAD_INT(a)*100+LOAD_FRAC(a)-500;

  SEND_PROBE_ALARM (PROBE_ID, NULL, 0, load1);
  return load1;
}
```

Cette sonde crée de fortes limitations. Le fait qu'elle extrait la charge moyenne du processeur durant la dernière minute induit une latence importante et risque de ne pas représenter la charge réelle après la création d'un nouveau processus sur un nœud. Des systèmes comme MOSIX offrent une sonde plus représentative de la charge réelle du processeur. Un portage de la méthode de calcul de la charge processeur du système MOSIX est en cours au sein d'une sonde système KERRIGHED.

Une fois la sonde mise en œuvre, il faut configurer celle-ci pour que le système puisse gérer les instances nécessaires à la mise en place de l'ordonnanceur global (voir listing 9.5).

Listing 9.5 – Fichier de configration XML des sondes pour l'ordonnanceur statique

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<probes>
  <timer_probe>
    <file>probe_loadavg1</file>
    <name>probe_loadavg1</name>
    <time>20</time>
  </timer_probe>
</probes>
```

Puisque la sonde est active (elle doit extraire la charge processeur périodiquement), le temps imparti à la sonde doit être spécifié. Dans notre exemple, la sonde se déclenche toutes les 20 secondes.

9.2.2 Description de l'analyseur local utilisé

Pour relayer les informations de la sonde processeur vers les GOrG, un AL basique est nécessaire. Cet AL n'a aucun rôle de filtrage ni d'analyse. Il est donc très simple à mettre en œuvre (voir listing 9.6), il suffit de relayer l'information (fonction *send_information*).

Listing 9.6 – Analyseur associé à la sonde de la charge moyenne du processeur lors de la dernière minute (cpu_analyzer2.c)

```
#include "analyzer_tools.h"

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("CPU Analyzer");
MODULE_LICENSE("GPL");
INIT_ANALYZER

int cpu_analyzer2 (void *arg)
{
  send_information ();
  return 0;
}
```

L'ordonnanceur global ne nécessite qu'une instance de cet AL. Le fichier de configuration comporte la déclaration de l'instance de cet AL et son association à l'instance locale de la sonde configurée précédemment (voir listing 9.7).

Listing 9.7 – Fichier de configration XML de l'AL de l'ordonnanceur statique

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<analyzers>
  <analyzer>
    <file>cpu_analyzer2</file>
    <name>cpu_analyzer2</name>
    <probes>
      <probe>probe_loadavg1</probe>
    </probes>
  </analyzer>
</analyzers>
```

9.2.3 Description du gestionnaire d'ordonnancement utilisé

Lorsqu'une information système est remontée par un AL, cette information est par défaut diffusée à l'ensemble des nœuds de la grappe. Lorsque les informations arrivent sur les nœuds, elles sont automatiquement stockées dans des tables. Cette gestion est totalement transparente au programmeur du GOrG. Le programmeur doit coder la fonction utilisée pour le placement de tâche, appelée dans cet exemple *cpu_scheduler_process_placement*, et la fonction de base du module qui est obligatoire, fonction appelée *cpu_scheduler2* (voir listing 9.8). Cette fonction doit normalement mettre en œuvre la politique appliquée lorsqu'un ordonnancement dynamique est nécessaire, et elle est donc vide dans cet exemple.

Listing 9.8 – GOrG de l'ordonnanceur statique (cpu_scheduler2.c)

```
#include "scheduler_tools.h"
#include "kernel_gthread.h"

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("CPU Scheduler");
MODULE_LICENSE("GPL");
INIT_SCHEDULER

int placement_according_to_cpu_load ()
{
  int i, diff=0, diff_max=0, selected_node=-1;
  int local_info;
  analyzer_list_t *analyzer_list;

  FOR_EACH_ANALYZER (analyzer_list)
    {
      global_table_t *table = GET_GLOBAL_TABLE (analyzer_list->analyzer);
      if (table != NULL)
        {
          local_info = get_node_info (table, NODE_ID);
          for (i=0; i<NB_KERRIGHED_NODES; i++)
            {
              /* on recherche pour chaque table globale */
              int remote_info;
              remote_info = get_node_info (table, i);
              diff = local_info - remote_info;
              if (diff > 100 && diff >= diff_max)
                {
                  diff_max = diff;
                  selected_node = i;
                }
            }
        }
    }
  return selected_node;
}

int cpu_scheduler_process_placement ()
{
  int selected_node=-1;

  selected_node = placement_according_to_cpu_load ();
```

```
  return selected_node;
}

int cpu_scheduler2 (void *arg)
{
  return 0;
}
```

Lorsqu'une tâche est soumise à l'ordonnanceur, celui-ci appelle automatiquement la fonction *cpu_scheduler_process_placement*. Grâce aux informations stockées dans la table (il n'y a ici qu'une table qui stocke l'utilisation processeur des nœuds), l'ordonnanceur compare la charge processeur du nœud local avec les autres nœuds de la grappe. Le but de la politique est de trouver un nœud qui accueille au moins un processus actif de plus que le nœud courant et le nœud de la grappe qui est le plus chargé. Puisque la charge d'un nœud est exprimée sous la forme $charge = nombre\ de\ processus\ actifs * 100$, la recherche d'un nœud candidat pour accueillir un nouveau processus est $(charge_local - charge_du_nud_analys) > 100\ et\ diff >= diff_max$ où $diff_max$ est la différence maximum courante de charge trouvée entre deux nœuds.

La configuration du GOrG s'effectue en deux phases au sein du fichier *scheduler_conf.xml* (voir listing 9.9).

Listing 9.9 – Fichier de configration XML du GOrG de l'ordonnanceur statique

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<schedulers>
  <scheduler>
    <nodes>all</nodes>
    <name>scheduler</name>
    <file>cpu_scheduler2</file>
    <process_placement>cpu_scheduler_process_placement</process_placement>
    <analyzers>
      <analyzer>
        <name>cpu_analyzer2</name>
        <global_info/>
      </analyzer>
    </analyzers>
  </scheduler>
</schedulers>
```

La première phase concerne la création et le déploiement du GOrG. Pour cela, il est nécessaire de spécifier les nœuds sur lesquels l'instance doit être déployée (le déploiement sur un sous-ensemble de nœuds de la grappe n'est pas totalement finalisé), le nom de l'instance, le nom du fichier source du GOrG et enfin le nom de la fonction de placement des processus. La seconde phase consiste à spécifier les AL qui sont associés aux GOrG. Dans cet exemple, seul l'AL remontant les informations processeur du nœud est à associer au GOrG, en

spécifiant que les données remontées sont à stocker dans une table.

9.3 Ordonnanceur dynamique

La politique statique présentée précédemment ne permet pas de régler dynamiquement des déséquilibres de charge (*e.g.* si des tâches sont lancées sans être soumises à l'ordonnanceur). Nous montrons dans ce paragraphe comment introduire un équilibrage de charge dynamique (d'initiative au récepteur) en étendant l'ordonnanceur statique présenté dans le paragraphe 9.2.

Pour cela, nous avons mis œuvre une approche fondée sur l'étude de la charge des différents nœuds. Lorsqu'un GOrG reçoit la charge processeur d'un nœud, il compare la charge de ce nœud avec la charge des autres nœuds. Si une différence de charge est détectée (on utilise les mêmes règles de détection que celles présentées dans le paragraphe 9.2.3), *i.e.* si un nœud distant est trop chargé par rapport au nœud courant, le nœud courant demande la migration d'un processus au nœud distant. Pour éviter des décisions d'ordonnancement inadaptées dûes au manque de précision des sondes de la charge moyenne du processeur lors de la dernière minute, la politique doit désactiver l'envoi ou la réception de processus pendant une minute après une migration. Cette déactivation permet de prendre en ompte la migration effectuée dans la charge moyenne processeur durant la dernière minute sur les deux nœuds concernés par une migration.

De plus, nous ne souhaitons pas que toutes les applications puissent être déplacées pendant leur exécution. Pour cet exemple, seules les applications nommées *up-gs* et *mgs* peuvent être déplacées.

Nous avons donc besoin de nouveaux composants :
- une nouvelle sonde pour détecter les migrations,
- un nouvel AL associé à la sonde de migration.

En plus de ces deux nouveaux composants, il est nécessaire détendre le GOrG présenté au sein de l'ordonnanceur statique.

9.3.1 Description des sondes utilisées

La sonde extrayant la charge moyenne du processeur durant la dernière minute peut être réutilisée (voir listing 9.4).

Une nouvelle sonde détectant les migrations est nécessaire. Cette sonde se déclenche lorsqu'un processus est envoyé vers un nœud distant ou lorsqu'un processus d'un nœud distant arrive sur un nœud. Cette sonde est donc une sonde passive et elle a été intégrée directement dans KERRIGHED.

Le fonctionnement de cette sonde est simple : lorsqu'une migration est détectée (*i.e.* le système KERRIGHED intègre un appel de la sonde à l'envoi et à la réception d'un processus lors d'une migration), la sonde envoie une alarme vers l'AL qui lui est associé (voir listing 9.10).

Listing 9.10 – Sonde détectant une migration (migration_probe.c)

```
#include "probe_tools.h"

#include <linux/sched.h>
#include <linux/kernel_stat.h>

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("Migration Probe");
MODULE_LICENSE("GPL");

INIT_PROBE

weight_t migration_probe (void *arg)
{
  int migration = 1;
  PRINT ("migration detected\n");
  SEND_PROBE_ALARM (PROBE_ID, NULL, 0, migration);
  return migration;
}
```

La nouvelle sonde doit être ensuite configurée au sein du fichier *probe_ conf.xml* comme étant une sonde passive (voir listing 9.11).

Listing 9.11 – Fichier de configration XML des sondes pour l'ordonnanceur dynamique

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<probes>
  <probe>
    <file>migration_probe</file>
    <name>migration_probe</name>
  </probe>
  <timer_probe>
    <file>probe_loadavg1</file>
    <name>probe_loadavg1</name>
    <time>20</time>
  </timer_probe>
</probes>
```

9.3.2 Description des analyseurs locaux utilisés

L'AL associé à la sonde processeur et présenté dans le listing 9.6 est réutilisé.

Un nouvel AL doit être associé à la sonde de migration. Cet AL doit recevoir les alarmes levées par la sonde de migration et les relayer vers le GOrG auquel il est associé (voir listing 9.12).

Listing 9.12 – Analyseur associé à la sonde de migration (migration_analyzer.c)

```
#include "analyzer_tools.h"

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("Migration Analyzer");
MODULE_LICENSE("GPL");
INIT_ANALYZER

int migration_analyzer (void *arg)
{
  global_schedule (1);

  return 0;
}
```

Le nouvel AL doit configuré au sein du fichier *analyzer_ conf.xml* comme étant associé à la sonde de migration (voir listing 9.13).

Listing 9.13 – Fichier de configration XML de l'AL de l'ordonnanceur statique

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<analyzers>
  <analyzer>
    <file>cpu_analyzer2</file>
    <name>cpu_analyzer2</name>
    <probes>
      <probe>probe_loadavg1</probe>
    </probes>
  </analyzer>
  <analyzer>
    <file>migration_analyzer</file>
    <name>migration_analyzer</name>
    <probes>
      <probe>migration_probe</probe>
    </probes>
  </analyzer>
</analyzers>
```

9.3.3 Description du gestionnaire d'ordonnancement global utilisé

Maintenant que nous disposons des sondes et des AL nécessaires, la politique dynamique doit être mise en œuvre. Cette politique nécessite deux principales fonctions : (i) comment trouver un nœud lorsqu'un déséquilibre est détecté et (ii) comment choisir un processus pour effectuer une migration.

La fonction d'élection d'un nœud est simple : la politique d'ordonnancement d'un nœud

recherche le nœud distant dont la charge est supérieure d'au moins un processus actif et qui est la plus haute de la grappe (voir listing 9.14).

Listing 9.14 – Fonction d'éléction d'un nœud distant de l'ordonnanceur dynamique

```
int cpu_scheduler_find_a_remote_node (void *arg)
{
  int i, diff, diff_max=0, selected_node=-1;

  for (i=0; i<NB_KERRIGHED_NODES; i++)
    {
      int local_info, remote_info;
      local_info = get_local_info_on_local_update (arg, NODE_ID);
      remote_info = get_local_info_on_local_update (arg, i);
      diff = local_info - remote_info;
      if (diff < -100 && diff <= diff_max)
        {
          diff_max = diff;
          selected_node = i;
        }
    }
  return selected_node;
}
```

L'élection d'un processus candidat pour une migration est plus complexe. Elle doit tout d'abord ne sélectionner que les processus des applications *up-gs* et *mgs*. Ensuite, la politique ne doit pas accepter de déplacer un processus si une migration a eu lieu il y a moins d'une minute. Cela implique que l'élection d'un processus soit impossible si cette minute n'est pas écoulée. Un timer (*migration_wait*) est utilisé (voir listing 9.15).

Listing 9.15 – Fonction d'éléction d'un processus de l'ordonnanceur dynamique

```
struct task_struct *find_a_process_to_migrate ()
{
  struct task_struct *p;
  if (migration_wait > 0)
    {
      PRINT ("Migration impossible, waiting (%d)...\n", migration_wait);
      return NULL;
    }
  for_each_task (p)
    {
      if (strcmp(p->comm, "up-gs") == 0 || strcmp(p->comm, "mgs") == 0)
          if (p->state == TASK_RUNNING)
              return (p);
    }
```

```
  return NULL;
}
```

Pour compléter cet ordonnanceur, il est nécessaire de gérer l'initialisation du timer et de compléter sa gestion.

Listing 9.16 – Traitement d'une alerte remontée par l'AL de migration

```
void init_wait_time ()
{
  PRINT ("Init waiting timer\n");
  migration_wait = 7;
}

int cpu_scheduler2 (void *arg)
{
  if (analyzer_comp (ANALYZER_ID, "migration_analyzer") == 0)
      init_wait_time ();

  return 0;
}
```

L'approche la plus simple est d'initialiser le timer quand une migration est détectée, *i.e.* une alerte, (voir listing 9.16) et d'utiliser la périodicité de la sonde de charge processeur pour décrémenter le timer (voir listing 9.17).

Listing 9.17 – Réception d'une information locale

```
int cpu_loads_update_local_info (void *arg)
{
  if (migration_wait > 0)
      migration_wait --;
  return 0;
}
```

Finalement le code du GOrG permettant de mettre en œuvre la politique dynamique correspond au listing 9.18.

Listing 9.18 – GOrG de l'ordonnanceur dynamique

```
#include "scheduler_tools.h"
#include "kernel_gthread.h"

MODULE_AUTHOR("Geoffroy Vallee");
MODULE_DESCRIPTION("CPU Scheduler2");
MODULE_LICENSE("GPL");
INIT_SCHEDULER
```

```
/* after a migration, the system wait before made another migration */
/* the wait time is managed by the time variable */
/* if (wait == 0) then the system can migrate a process */
/* else the system wait */
int migration_wait = 7;

int placement_according_to_cpu_load ()
{
  int i, diff=0, diff_max=0, selected_node=-1;
  int local_info;
  analyzer_list_t *analyzer_list;

  FOR_EACH_ANALYZER (analyzer_list)
    {
      global_table_t *table = GET_GLOBAL_TABLE (analyzer_list->analyzer);
      if (table != NULL)
        {
          local_info = get_node_info (table, NODE_ID);
          for (i=0; i<NB_KERRIGHED_NODES; i++)
            {
              /* on recherche pour chaque table globale */
              int remote_info;
              remote_info = get_node_info (table, i);
              diff = local_info - remote_info;
              if (diff > 100 && diff >= diff_max)
                {
                  diff_max = diff;
                  selected_node = i;
                }
            }
        }
    }
  return selected_node;
}

int cpu_scheduler_process_placement ()
{
  int selected_node=-1;
  selected_node = placement_according_to_cpu_load ();
  return selected_node;
}

int cpu_scheduler_find_a_remote_node (void *arg)
```

```
{
  int i, diff, diff_max=0, selected_node=-1;

  for (i=0; i<NB_KERRIGHED_NODES; i++)
    {
      int local_info, remote_info;
      local_info = get_local_info_on_local_update (arg, NODE_ID);
      remote_info = get_local_info_on_local_update (arg, i);
      diff = local_info - remote_info;
      if (diff < -100 && diff <= diff_max)
        {
          diff_max = diff;
          selected_node = i;
        }
    }
  return selected_node;
}

struct task_struct *find_a_process_to_migrate ()
{
  struct task_struct *p;
  if (migration_wait > 0)
      return NULL;
  for_each_task (p)
      if (strcmp(p->comm, "up-gs") == 0 || strcmp(p->comm, "mgs") == 0)
          if (p->state == TASK_RUNNING)
              if (get_aragorn_action (p) == NO_ARAGORN_ACTION)
                  return (p);
  return NULL;
}

/* function to manage node information */
int cpu_loads_update_node_info (void *arg)
{
    int remote_node;
    if (migration_wait > 0)
      return 0;
    remote_node = cpu_scheduler_find_a_remote_node (arg);
    if (remote_node >= 0)
        request_a_migration (remote_node, SCHEDULER_ID);
  return 0;
}
```

```
int cpu_loads_update_local_info (void *arg)
{
  if (migration_wait > 0)
    migration_wait --;
  return 0;
}

void init_wait_time ()
{
  PRINT ("Init waiting timer\n");
  migration_wait = 7;
}


int cpu_scheduler2 (void *arg)
{
  if (analyzer_comp (ANALYZER_ID, "migration_analyzer") == 0)
      init_wait_time ();

  return 0;
}
```

9.4 Résumé

L'architecture modulaire de l'ordonnanceur global permet de programmer facilement de nouveaux ordonnanceurs globaux, qu'ils soient statiques ou dynamiques.

La programmation des sondes restent néanmoins complexe car très proche du système. Des sondes ont été développées et sont fournies dans le système KERRIGHED. Ces sondes peuvent être réutilisées par les programmeurs de nouveaux ordonnanceur globaux.

La programmation de nouveaux composants est facilitée par la mise à disposition de macros et de primitives permettant de cacher la difficulté de programmation noyau. Le programmeur a donc uniquement à se concentrer sur la mise en œuvre des algorithmes de la politique d'ordonnancement.

De nouvelles politiques sont en cours de développement, notamment pour l'exécution d'applications MPI. L'utilisation du standard MPI introduisant souvent des dépendances entre différents processus de l'application (*e.g.* processus de calcul associés à des processus de surveillance des communications), des politiques spécialisées sont nécessaires pour bénéficier d'un équilibrage de charge.

Troisième partie

Mise en œuvre et évaluation

Notre contribution à la conception du système à image unique KERRIGHED a trait à la gestion globale des processus. Nous avons mis en œuvre dans le prototype de KERRIGHED les propositions que nous avons présentées dans la partie 2 de ce document. Ainsi, nous avons implanté l'architecture d'ordonnancement global adaptable décrite dans le chapitre 6 et implanté en son sein différentes politiques d'ordonnancement global, en particulier celles présentées dans le chapitre 9. Nous avons également mis en œuvre tous les mécanismes de gestion globale de processus sur lesquels s'appuie l'ordonnanceur global (création distante, duplication, migration, point de reprise de processus) et décrits dans le chapitre 8.

La mise en œuvre de nos propositions a été effectuée dans le contexte du noyau LINUX qui a été étendu par deux nouveaux modules, KERGHOST pour la mise en œuvre du concept de processus fantôme et KERPROC pour les mécanismes de déploiement, migration et point de reprise de processus sans compter les modules implantant les politiques d'ordonnancement global.

Nous avons pu mettre en place l'ordonnanceur global sans modifier l'ordonnanceur de processus du système LINUX. Des modifications mineures du noyau LINUX se sont avérées nécessaire pour créer un unique point d'entrée pour l'ensemble des mécanismes). La gestion globale des processus de KERRIGHED représente un patch d'une centaine de lignes au noyau LINUX 2.4.24 ainsi que 20.000 lignes de code C de niveau noyau dans des modules.

Ce chapitre est organisé de la manière suivante. Après une brève introduction au noyau LINUX, nous présentons dans les paragraphes qui suivent des éléments de mise en œuvre pour l'ordonnanceur global, les mécanismes de gestion globale de processus dans une grappe et la gestion des threads. Nous terminons par un résumé.

10 ÉLÉMENT DE MISE EN ŒUVRE

10.1 Le système d'exploitation LINUX

Le système d'exploitation hôte choisi pour la mise en œuvre du système KERRIGHED est GNU/LINUX [28]. Le système d'exploitation GNU/LINUX est un système d'exploitation de type Unix™développé par la communauté du logiciel libre. Fondé sur un noyau monolithique, il inclut la notion de **module** permettant d'étendre les fonctionnalités du noyau durant son exécution. KERRIGHED a été mis en œuvre par des modules intégrés au noyau LINUX.

Le noyau LINUX propose un mécanisme standard pour dialoguer avec le noyau. Pour cela, le répertoire /proc contient des fichiers virtuels qui offrent des informations sur l'état courant du noyau Linux. Cela permet aux utilisateurs de scruter une vaste gamme d'informations mais également de communiquer des changements de configuration au noyau. Le répertoire /proc et son contenu font partie d'un système de fichiers virtuel. De nombreux programmes utilisent le système de fichiers /proc pour obtenir les paramètres du système (état du noyau, attributs de la machine, état des processus) ou interagir avec le noyau. Pour cela, il est possible de consulter les informations via le répertoire /proc (lecture dans le système de fichiers virtuel) et de communiquer des paramètres au noyau (écriture dans le système de fichiers virtuel). Ces lectures/écriture dans le système de fichier virtuel génèrent des *ioctl*, appels systèmes permettant de faire appel à des fonctions du noyau via une interface utilisateur.

La mise en œuvre de nos propositions a été réalisée dans un premier temps sur le noyau 2.2.13. Un portage a été réalisé au cours de l'automne 2003 sur un noyau 2.4.24 avec l'aide de David Margery.

Dans ce chapitre, nous supposons que le lecteur a une assez bonne connaissance du fonctionnement interne du noyau LINUX.

10.2 L'ordonnanceur global

10.2.1 Éléments de mise en œuvre

Une instance de tous les composants constituant un ordonnanceur global (sondes, analyseurs locaux et gestionnaire d'ordonnancement global) est présente sur chaque nœud. Afin d'offrir des mécanismes tolérants aux fautes, aucun des composants n'est fondé sur un fonctionnement centralisé : les sondes et les analyseurs locaux ont un fonctionnement

strictement local, et une instance du gestionnaire d'ordonnancement est créée sur chaque nœud.

De même tous les échanges de messages entre composants d'un même nœud se font via des requêtes rangées dans des files d'attente. Ainsi, si le destinataire n'est pas actif, le message est ignoré, sans provoquer de blocage pour l'expéditeur ni pour les autres composants. Ces files d'attente sont gérées par le gestionnaire de communication qui est mis en œuvre par un serveur noyau sur chaque nœud. Le gestionnaire de communication analyse chaque requête pour en trouver les éventuels destinataires. Quand ceux-ci sont déterminés, une nouvelle requête est créée pour chacun d'entre eux puis rangée dans la file d'attente correspondante. Pour améliorer le fonctionnement général des communications locales entre composants, un modèle de communication de publication/souscription pourrait être mis en œuvre. En effet, avec le modèle de requêtes mis dans des listes d'attente, lors de la duplication des requêtes pour chacun des destinataires, une copie est à chaque fois générée. Avec un modèle publication/souscription, la requête n'est pas dupliquée, elle est seulement laissée à disposition tant que tous les destinataires ne l'ont pas traitée.

Les communications entre GOrG sont effectuées grâce à un gestionnaire de communication entre les GOrG de différents nœuds. Ce gestionnaire est chargé de recevoir toutes les communications transmises via le réseau et destinées à des GOrG s'exécutant localement. Après avoir reçu un message, le gestionnaire l'analyse pour déterminer la liste des GOrG destinataires. Lorsque cette liste est construite, une requête est envoyée pour chacun des GOrG. Encore une fois, ce fonctionnement garantit qu'aucun blocage ne peut intervenir si un composant n'est pas actif. Si le réseau supporte les communications non bloquantes, le mécanisme d'ordonnancement global est tolérant aux fautes : des nœuds peuvent disparaître/apparaître, tous les composants de l'ordonnanceur global continueront de fonctionner correctement.

10.2.1.1 Mise en œuvre du gestionnaire de communication

La gestion de communication entre les composants d'un ordonnanceur global est mis en œuvre par un thread noyau. Son fonctionnement est totalement indépendant du noyau et de l'utilisateur, et dépend uniquement des actions effectuées sur les différentes files de requêtes. Trois files de requêtes sont mises en œuvre au sein de KERRIGHED : une file de requêtes pour les sondes, une pour les analyseurs locaux et une pour les gestionnaires d'ordonnancement global. Nous avons vu dans le paragraphe 7.2 que lorsqu'une requête est insérée dans une file et si le gestionnaire est inactif, alors celui-ci devient actif, traite les requêtes une à une des différentes files tant que les files ne sont pas toutes vides.

Les composants d'un ordonnanceur global disposent d'une interface pour intervenir sur les différentes files de requêtes. Cette interface permet de manipuler les requêtes au sein d'une file, manipulations permettant au gestionnaire de communication de devenir actif.

```
request_queue_t* queue_request (request_queue_t *request_queue, int req
uest_type, void *arg, size_t arg_size);
```

Cette fonction permet de mettre une requête dans une file donnée (**request_queue**).

La requête est constituée d'un **type**, de données (**arg**) et d'une taille (**size**).

```
request_queue_t* dequeue_request (request_queue_t *request_queue);
```
Cette fonction permet de retirer la requête en tête d'une file (**request_queue**).

```
int get_request_type (request_queue_t *request_queue);
```
Cette fonction permet d'obtenir le type de la requête en tête de file (**request_queue**).

```
void* get_request_arg (request_queue_t *request_queue);
```
Cette fonction permet d'obtenir un pointeur sur les données de la requête en tête d'une file (**request_queue**).

```
size_t get_request_arg_size (request_queue_t *request_queue);
```
Cette fonction permet d'obtenir la taille des données de la requête en tête d'une file (**request_queue**).

10.2.1.2 Mise en œuvre des sondes

Les sondes peuvent être activées par un événement système (sondes passives) ou par un *timer* (sondes actives), et peuvent envoyer des informations (appelées ici *alarmes*) aux AL associés. KERRIGHED dispose d'une interface interne permettant la gestion du cycle de vie des sondes (création, activation, terminaison).

Deux types d'interface sont disponibles : une interface interne qui n'est pas normalement utilisée par le programmeur d'ordonnanceurs globaux et une interface de programmeur pour la mise en œuvre de nouveaux ordonnanceurs.

10.2.1.2.1 Interface interne d'utilisation des sondes Cette interface permet de gérer le cycle de vie entier d'une sonde, à savoir de la création d'une instance de sonde sur un nœud jusqu'à sa terminaison en passant par son déclenchement. Voici les principales fonctions disponibles :

```
probe_t* create_probe (int (*function) (void *arg), char desc[MAX_NAME_L
ENGTH]);
```
Cette fonction permet de créer une instance de sonde passive. La création d'une instance de sonde passive nécessite un pointeur sur la fonction définissant la sonde (*function*), ainsi que son nom (*desc*). Cette fonction est utilisée lors du lancement des sondes passives par le chargeur des sondes qui fournit comme nom le nom spécifié dans le fichier de configuration XML des sondes.

```
probeId_t create_probe_with_timer (int (*function) (void *arg), unsigned
 long time, char desc[MAX_NAME_LENGTH]);
```
Cette fonction permet de créer une instance de sonde active. La création d'une sonde active nécessite un pointeur sur la fonction définissant la sonde (*function*), la valeur que

l'on souhaite attribuer au *timer*, et le nom de la sonde (*desc*). Cette fonction est utilisée lors du lancement des sondes actives par le chargeur des sondes qui fournit comme nom le nom spécifié dans le fichier de configuration XML des sondes.

```
probe_list_t* add_in_probe_list (probe_t *probe, probe_list_t *probe_list);
```

Le système tient à jour une liste des instances de sondes en cours d'exécution au sein du système local. La fonction *add_in_probe_list* permet d'ajouter une instance de sonde (*probe*) déjà créée au sein de cette liste (*probe_list*).

```
weight_t call_probe (unsigned long probeId);
```

Cette fonction permet d'appeler une sonde en utilisant son identificateur (*probeId*). Cet identificateur est unique sur le nœud et attribué lors de la création d'une instance de la sonde.

```
weight_t call_probe_by_name (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet d'appeler une sonde en utilisant son nom (*name*). Ce nom est défini par le programmeur de l'ordonnanceur global dans le fichier de configuration XML des sondes.

```
int remove_analyzer_from_probe_list (probe_t *probe, analyzer_t *analyzer);
```

Cette fonction permet de dissocier un analyseur local (*analyzer*) et une sonde (*analyzer*). La sonde est alors incapable de communiquer avec l'analyseur.

```
int remove_analyzer_for_probe (analyzer_t *analyzer, probe_t *probe);
```

Cette fonction permet de dissocier une sonde (*probe*) et un analyseur (*analyzer*). L'analyseur est alors incapable de communiquer avec la sonde.

```
void remove_probes ();
```

Cette fonction permet de retirer toutes les sondes du système.

10.2.1.2.2 Interface de programmation des sondes Afin de faciliter la programmation de nouveaux ordonnanceurs globaux, un ensemble de fonctions permettant de manipuler les sondes sont disponibles.

```
int send_alarm (probeId_t probeId, void *arg, size_t arg_size, weight_t poids);
```

Cette fonction permet à une sonde d'envoyer une alarme (une information) vers les AL auxquels la sonde est associée (ces associations sont définies dans le fichier de configuration XML). L'envoi d'une alarme nécessite de fournir l'identificateur de la sonde (*probeID*), une information à transmettre (*arg*), la taille de cette information (*arg_size*) et un *poids*. Le

poids permet de quantifier l'importance de l'information transmise.

```
probe_t *find_probe (probeId_t probeId);
```
Cette fonction permet d'obtenir un pointeur vers la sonde dont l'identificateur est fourni (*probeId*).

```
probe_t *find_probe_by_name (char name[MAX_NAME_LENGTH]);
```
Cette fonction permet d'obtenir un pointeur vers la sonde dont le nom est fourni (*name*).

Les sondes peuvent également être activées ou désactivées dans le cas d'une reconfiguration du système. Pour cela, trois fonctions sont disponibles, utilisables pour la mise en œuvre de politique d'ordonnancement auto-adaptable.

```
int active_probe (char name[MAX_NAME_LENGTH]);
```
Cette fonction permet d'activer une instance de sonde identifiée par son nom (*name*).

```
int desactive_probe (char name[MAX_NAME_LENGTH]);
```
Cette fonction permet de désactiver une instance de sonde identifiée par son nom (*name*).

```
int get_probe_state (probe_t *);
```
Cette fonction permet d'obtenir l'état (activé ou désactivé) d'une instance de sonde.

10.2.1.3 Mise en œuvre des analyseurs locaux

Les analyseurs locaux sont chargés de filtrer les informations fournies par les sondes qui lui sont associées, mais également de détecter états locaux pouvant influencer l'ordonnancement global. Deux types d'interfaces sont disponibles : une interface interne pour la gestion du cycle de vie des instances d'analyseurs locaux et une interface de programmation utilisable pour la programmation de nouveaux analyseurs locaux.

10.2.1.3.1 Interface interne d'utilisation des analyseurs locaux Cette interface permet de gérer le cycle de vie entier d'une instance d'AL, à savoir sa création sur un nœud jusqu'à sa terminaison en passant par son déclenchement. Voici les principales fonctions disponibles :

```
analyzer_t* create_analyzer (int (*function) (void *arg), char name[MAX
_NAME_LENGTH]);
```
Cette fonction permet de créer une instance d'AL. La création d'une instance d'un AL nécessite un pointeur sur la fonction définissant l'AL (*function*), ainsi que son nom (*desc*). Cette fonction est utilisée lors du lancement des AL par le chargeur des AL qui fournit

comme nom le nom spécifié dans le fichier de configuration XML des AL.

```
int add_probe (analyzer_t *analyzer, char probe_name[MAX_NAME_LENGTH]);
```

Cette fonction permet d'associer une sonde identifiée par son nom (*probe_name*) à un AL (*analyzer*).

```
void remove_analyzers ();
```

Cette fonction permet de retirer tous les AL du système.

```
int remove_probe_for_analyzer (analyzer_t *analyzer, probe_t *probe);
```

Cette fonction permet de dissocier une sonde (*probe*) d'un AL (*analyzer*).

```
int remove_scheduler_for_analyzer (scheduler_t *scheduler, analyzer_t *a
nalyzer);
```

Cette fonction permet de dissocier un GOrG (*scheduler*) d'un AL (*analyzer*).

```
analyzer_list_t* add_in_analyzer_list (analyzer_t *analyzer, analyzer_li
st_t *analyzer_list);
```

Le système tient à jour une liste de toutes les instances d'AL actives au sein du système. La fonction *add_in_analyzer_list* permet d'ajouter une instance d'AL (*analyzer*) déjà créée au sein de cette liste (*analyzer_list*).

```
int for_each_analyzer(probeId_t probeId, int request_type, void *arg, si
ze_t arg_size);
```

Cette fonction permet d'envoyer une requête à tous les AL associés à une sonde donnée. La sonde est identifiée par son identifieur local (*probeId*), et la requête est définie par son type (*request_type*), ses données (*arg*) et la taille de ses données (*arg_size*).

10.2.1.3.2 Interface de programmation des analyseurs locaux Afin de faciliter la programmation de nouveaux ordonnanceurs globaux, un ensemble de fonctions permettant de manipuler les AL sont disponibles.

```
analyzer_t *find_analyzer_by_name (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet d'obtenir un pointeur vers l'instance d'un AL dont le nom est fourni (*name*).

```
int __analyzer_comp (analyzerId_t id, char name[MAX_NAME_LENGTH]);
```

Cette fonction permet de comparer deux AL dont on dispose de l'identifiant pour le premier (*id*) et du nom pour le second (*name*).

```
int send_scheduling_request (analyzerId_t analyzerId, priority_t priorit
```

```
y);
```

Cette fonction permet d'envoyer une requête d'ordonnancement aux GOrG associés à une instance d'AL. L'instance d'AL est identifiée par son identificateur (*analyzerId*). La requête est envoyée accompagnée d'une priorité (*priority*) permettant de quantifier l'importance de la requête.

```
int send_information_to_scheduler (analyzerId_t analyzerId, weight_t
weight, int sn);
```

Cette fonction permet d'envoyer une information système (*weight*) recueillie par un AL (*analyzerID*) grâce à une sonde vers les GOrG associés. Cette information système est accompagnée d'un numéro de séquence (géré par le système) permettant de gérer l'ordre d'arrivée des informations système, que ces informations soient locales ou provenant d'autres nœuds.

Les AL peuvent également être activés ou désactivés dans le cas d'une reconfiguration du système. Pour cela, trois fonctions sont disponibles, utilisables pour la mise en œuvre de politiques d'ordonnancement auto-adaptable.

```
int activate_analyzer (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet d'activer une instance d'AL identifié par son nom (*name*).

```
int desactivate_analyzer (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet de désactiver une instance d'AL identifié par son nom (*name*).

```
int get_analyzer_state (analyzer_t *);
```

Cette fonction permet d'obtenir l'état (activé ou désactivé) d'une instance d'AL.

10.2.1.4 Mise en œuvre des gestionnaires d'ordonnancement global

Les GOrG sont chargés de détecter et de résoudre les ordonnancements de processus ou threads non conformes à la politique d'ordonnancement appliquée, qu'ils soient locaux ou globaux. La résolution d'un problème d'ordonnancement nécessite de pouvoir si besoin désigner un processus à déplacer, de pouvoir désigner un nœud comme cible d'un nouvel ordonnancement. Les GOrG doivent de plus gérer la diffusion des informations système des nœuds au sein de la grappe. Cela nécessite de définir une fonction de diffusion mais également de définir une méthode de réception et de stockage de ces informations. Deux types d'interfaces sont donc disponibles : une interface interne pour la gestion du cycle de vie des instances de GOrG et une interface de programmation utilisable pour la programmation de nouveaux GOrG. L'interface interne permet de gérer non seulement le cycle de vie d'une instance de GOrG mais également la réception et le stockage des informations système.

10.2.1.4.1 Interface interne de fonctionnement des GOrG Cette interface permet de gérer le cycle de vie entier d'une instance de GOrG, à savoir sa création sur un nœud jusqu'à sa terminaison en passant par son déclenchement. Voici les principales fonctions disponibles :

```
scheduler_t* create_scheduler (int (*function) (void *arg), char name[16
]);
```

Cette fonction permet de créer une instance de GOrG identifiée par un nom (*name*) et définie par une fonction (*function*). Cette fonction est utilisée par le chargeur des GOrG en utilisant le nom donné dans le fichier de configuration XML.

```
kerrighed_node_t find_a_node_according_to_the_scheduling_policy ();
```

Cette fonction permet au système d'interroger la politique d'ordonnancement active. Cette fonction est notamment utilisée lors de la création d'un processus sur un nœud distant : on interroge dans un premier temps la politique d'ordonnancement pour obtenir un identifiant de nœud pouvant accueillir un processus. Lorsque le système possède cet identifiant, il envoie alors une référence sur le fichier objet de l'application à lancer, les paramètres de lancement et l'application peut alors être créée.

```
int do_global_schedule ();
```

Cette fonction est utilisée pour déclencher la politique d'ordonnancement active au sein du système. Cette fonction permet au gestionnaire de communication entre composants d'un ordonnanceur global de déclencher une politique d'ordonnancement lorsqu'une requête d'ordonnancement est traitée.

```
int update_node_information (local_information_t *info, global_table_t
tab[kerrighed_nb_nodes]);
```

Les informations système des nœuds de la grappe gérées par l'ordonnanceur global sont stockées dans des tables sur chaque nœud. La fonction *update_ node_ information* permet de mettre à jour une information (*info*) dans une table (*tab*).

```
int add_analyzer (scheduler_t *scheduler, char analyzer_name[16]);
```

Cette fonction permet d'associer un AL, identifié par son nom (*analyzer_ name*), conformément au nom donné dans le fichier de configuration XML des AL), à un GOrG (*scheduler*).

```
void assign_function_of_update_local_info (scheduler_t *scheduler,
int (*function) (void *arg));
```

Cette fonction permet de définir la fonction (*function*) appelée lors de la mise à jour d'une information système du nœud local pour une instance d'ordonnanceur global (*scheduler*). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
void assign_function_of_update_node_info (scheduler_t *scheduler,
int (*function) (void *arg));
```

Cette fonction permet de définir la fonction (*function*) créée par le programmeur et appelée lors de la mise à jour d'une information système d'un nœud distant pour une instance d'ordonnanceur global (*scheduler*). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
void assign_function_to_find_a_node (scheduler_t *scheduler, int
(*function) (void *arg));
```

Cette fonction permet de définir quelle est la fonction créée par le programmeur (*function*) pour désigner un nœud lors d'un nouvel ordonnancement des tâches, pour une instance de GOrG donnée (*scheduler*). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
void assign_function_to_find_a_process (scheduler_t *scheduler, struct
task_struct* (*function) (void *arg));
```

Cette fonction permet de définir quelle est la fonction créée par le programmeur (*function*) pour désigner un processus lors d'un nouvel ordonnancement des tâches, pour une instance de GOrG (*scheduler*). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
void assign_diffusion_function (scheduler_t *scheduler, int (*function)
(void *arg, size_t size));
```

Cette fonction permet de définir quelle est la fonction créée par le programmeur (*function*) pour mettre en œuvre la diffusion des informations système pour une instance de GOrG (*scheduler*). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
kerrighed_node_t find_a_remote_knode ();
```

Cette fonction permet de définir quelle est la fonction créée par le programmeur (*function*) pour la politique à appliquer pour une instance de GOrG (*scheduler*) pour effectuer un équilibrage dynamique (*i.e.* en déplaçant des processus). Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
void assign_function_of_process_placement (scheduler_t *scheduler, int
(*function) ());
```

Cette fonction permet de définir quelle est la fonction créée par le programmeur (*function*) pour la politique à appliquer pour une instance de GOrG (*scheduler*) lors de la création d'un nouveau processus. Cette politique de placement peut être différente de celle utilisée pour l'ordonnancement dynamique. Cette fonction est utilisée par le chargeur des GOrG lors de la création des instances de GOrG conformément aux fichiers de configuration XML.

```
int for_each_scheduler(analyzerId_t analyzerId, int request_type, void
*arg, size_t arg_size);
```

Cette fonction permet d'envoyer une requête à tous les GOrG associés à un AL donné. L'AL est identifié par son identifieur local (*analyzerId*) et la requête est définie par son type (*request_type*), ses données (*arg*) et la taille de ses données (*arg_size*).

```
void remove_schedulers ();
```

Cette fonction permet de retirer toutes les instances de GOrG du système local.

10.2.1.4.2 Interface de programmation des GOrG Afin de faciliter la programmation de nouveaux ordonnanceurs globaux, un ensemble de fonctions permettant de manipuler les GOrG sont disponibles.

```
int request_a_migration (kerrighed_node_t nodeId, schedulerId_t
schedulerId);
```

Cette fonction permet de demander la migration d'un processus à une instance distante (sur le nœud *nodeId*) de GOrG (*schedulerId*).

```
int get_local_info_on_local_update (local_information_t *req, kerrighed_
node_t node_id);
```

Cette fonction permet d'obtenir l'information système locale d'un nœud (*node_id*) lors du traitement d'une information système (*req*).

```
global_table_t* __find_global_table (scheduler_t *scheduler, analyzerId_
t analyzerId);
```

Cette fonction permet de trouver la table globale utilisée par une instance de GOrG (*scheduler*) pour stocker les informations système provenant d'un AL (*analyzerId*).

```
int get_node_info (global_table_t *table, int node_id);
```

Cette fonction permet d'obtenir l'information système d'un nœud (*node_id*) contenue dans une table locale (*table*).

Les GOrG peuvent également être activés ou désactivés dans le cas d'une reconfiguration du système. Pour cela, trois fonctions sont disponibles, utilisables pour la mise en œuvre de politique d'ordonnancement auto-adaptable.

```
int activate_scheduler (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet d'activer une instance de GOrG identifiée par son nom (*name*).

```
int desactivate_scheduler (char name[MAX_NAME_LENGTH]);
```

Cette fonction permet de désactiver une instance de GOrG identifiée par son nom (*name*).

```
int get_scheduler_state (scheduler_t *);
```
Cette fonction permet d'obtenir l'état (activé ou désactivé) d'une instance de GOrG.

10.2.2 Mécanisme de configuration dynamique de l'ordonnanceur global

Des fichiers de configuration XML, grâce à des feuilles XSL-T fournies dans le système KERRIGHED, permettent de générer le code de modules noyau. Ces modules permettent de charger et de décharger les différents éléments de l'ordonnanceur global, que ce soit les sondes (le chargeur est appelé *probe_ loader*), les analyseurs locaux (le chargeur est appelé *analyzer_ loader*) ou les gestionnaires d'ordonnancement global (le chargeur est appelé *scheduler_ loader*). Chaque type de composant (*i.e.* sondes, analyseurs locaux et gestionnaires d'ordonnancement global) est configuré dans un fichier XML (*i.e.* le fichier *scheduler_ conf.xml* pour la configuration des GOrG, le fichier *analyzer_ conf.xml* pour la configuration des AL et le fichier *probe_ conf.xml* pour la configuration des sondes). Chaque fichier XML est associé à une feuille XSL-T fournie (voir listings A.1, A.3 et A.4 en annexe) permettant de générer le code du chargeur correspondant (voir figure 10.1). Les fichiers XSL-T permettent de gérer automatiquement le code de trois modules noyau. Un premier module permet de créer les instances des sondes (fichier *probe_ loader.c*), un second les instances d'AL (fichier *analyzer_ loader.c* et un troisième les instances de GOrG (fichier *scheduler_ loader.c*). Le programmeur doit, après avoir mis en œuvre les différents com-

FIG. 10.1 – Détails de la génération du code des différents chargeurs

posants constituant son ordonnanceur global, écrire les fichiers de configurations propres à chaque type de composants (*i.e.* les fichiers *scheduler_ conf.xml*, *analyzer_ conf.xml* et *probe_ conf.xml*) afin que le système puisse créer et initialiser automatiquement chaque instance des composants.

10.3 Les mécanismes de gestion globale des processus

Le module KERPROC met en œuvre la gestion globale des processus. Pour limiter les modifications du noyau et faciliter la maintenance de la mise en œuvre de cette gestion globale, un unique point d'entrée est utilisé pour l'ensemble des mécanismes. Ce point d'entrée unique permet ensuite de mettre en œuvre l'ensemble des mécanismes de gestion globale des processus.

Le paragraphe 10.3.1 présente le fonctionnement général du module KERPROC. Le paragraphe 10.3.2 présente des éléments de mise en œuvre du mécanisme de processus fantôme. Le paragraphe 10.3.4 présente la mise en œuvre de la duplication de processus alors que la mise en œuvre de la migration de processus est détaillée dans le paragraphe 10.3.3. Le paragraphe 10.3.6 présente la mise en œuvre de la création de points de reprise.

10.3.1 Accès aux mécanismes de KERPROC

La mise en œuvre de la gestion globale des processus au sein de KERPROC peut être divisée en plusieurs catégories :

1. L'accès aux services fournis par le module KERPROC, à travers l'interface */proc/kerrighed/services*. La liste de ces services, ainsi que le nom des fonctions correspondantes, est donnée dans le paragraphe 10.3.1.1.
2. Le cœur du module, la fonction *start_aragorn*, point d'entrée unique du module KERPROC, qui permet de transformer l'état d'un processus dans le contexte système de celui-ci à un moment où les valeurs des registres tels que vus par le processus sont accessibles. Ce point d'entrée unique est décrit dans le paragraphe 10.3.1.2.
3. Les fonctions générales du module KERPROC, permettant de connaître et de manipuler l'état d'un processus KERRIGHED. Les différents états d'un processus au sein du système KERRIGHED sont décrits dans le paragraphe 10.3.1.3.
4. La mis en œuvre des mécanismes de manipulation des processus qui sont détaillés dans les paragraphes 10.3.3, 10.3.4, 10.3.5 et 10.3.6.

10.3.1.1 Services du module KERPROC

L'utilisation du module KERPROC se fait soit à l'initiative du noyau *Linux* pour l'ordonnancement, soit par appel direct des fonctions des services fournis par */proc/kerrighed/services*, soit par appel aux fonctions de la bibliothèque de gestion des threads (appelée *libkrgthread*) ou aux fonctions de la librairie KERRIGHED (appelée *libkerrighed*). La librairie *libkerrighed* met en œuvre une interface utilisateur pour utiliser les services natifs offerts par KERRIGHED. La librairie *libkrgthread* offre, quant à elle, une interface utilisateur *pthread* fondée sur les services de KERRIGHED. La librairie *libkrgthread* fait donc appel à la librairie *libkerrighed*. Les fonctions de ces deux bibliothèques font elles-même appel, pour leur mise en œuvre, à */proc/kerrighed/services*. La liste des fonctions du module qu'il est possible d'appeler à travers cette interface est donnée dans le tableau 10.1.

Appel	Fonction
Fonction générale du module :	
GET_KERRIGHED_ID	sys_get_kerrighed_id
Fonctions pour la mise en œuvre des gthreads :	
ARAGORN_GET_GTHREAD_ID	get_gthread_id
ARAGORN_IS_THE_MASTER_THREAD	sys_process_is_the_master_thread
ARAGORN_GTHREAD_CREATE	aragorn_create_gthread
ARAGORN_GTHREAD_EXIT	aragorn_gthread_exit
ARAGORN_WAIT_GTHREAD_END	sys_aragorn_wait_gthread_end
Fonctions pour le placement et le déplacement de threads et de processus :	
ARAGORN_PROCESS_MIGRATION	sys_migration
ARAGORN_PROCESS_MIGRATION_WITHOUT_CNTRS	sys_migration_without_containers
ARAGORN_PROCESS_MIGRATION_USING_BLOB	sys_process_migration_using_blob
ARAGORN_THREAD_MIGRATION	thread_migration
ARAGORN_PROCESS_DUPLICATION	sys_task_duplication
ARAGORN_CURRENT_PROCESS_MIGRATION	sys_current_process_migration
KRG_FORK	sys_krg_fork
CREATE_PROCESS_ON_REMOTE_NODE	sys_remote_process_creation
Fonctions pour la tolérance aux fautes :	
ARAGORN_CHECKPOINT_ON_MEMORY	sys_memory_checkpoint
ARAGORN_CHECKPOINT_ON_DISK	sys_disk_checkpoint
ARAGORN_CHECKPOINT_ON_MEMORY...	sys_memory_checkpoint...
_WITHOUT_CONTAINERS	_without_containers
ARAGORN_CHECKPOINT_ON_DISK...	sys_disk_checkpoint...
_WITHOUT_CONTAINERS	_without_containers
ARAGORN_PROCESS_RESTART_FROM_DISK	sys_restart_from_disk
ARAGORN_PROCESS_RESTART_FROM_MEMORY	sys_restart_from_memory
ARAGORN_PARALLEL_CHECKPOINT	aragorn_parallel_checkpoint
ARAGORN_PROCESS_ROLLBACK_FROM_DISK	sys_process_rollback_from_disk
ARAGORN_PROCESS_ROLLBACK_FROM_MEMORY	sys_process_rollback_from_memory
ARAGORN_SYNCH	synch_api
Fonctions pour la gestion de groupes :	
ADD_PROCESS_IN_GROUP	sys_add_process_in_group
REMOVE_PROCESS_OF_GROUP	sys_remove_process_from_group
ADD_NODE_IN_GROUP	sys_add_node_in_group
REMOVE_NODE_OF_GROUP	sys_remove_node_from_group
Fonctions pour la mesure de performance :	
GET_LAST_MIGRATION_TIME	sys_get_last_migration_time
GET_MIGRATION_NUMBER	sys_get_migration_number
GET_TOTAL_MIGRATION_TIME	sys_get_total_migration_time
GET_PROCESS_EXECUTION_TIME	sys_get_execution_time
Fonctions utilitaires :	
ARAGORN_MAKE_KERRIGHED_PROCESS	sys_make_kerrighed_process
ARAGORN_CENTRALIZED_FOR_EACH_GTHREAD	centralized_for_each_gthread

TAB. 10.1 – Correspondance entre les appels Kerrighed à travers /proc/kerrighed/services et les fonctions du module les mettant en œuvre.

10.3.1.2 Fonction d'entrée des mécanismes du module ARAGORN : la fonction start_aragorn

L'ensemble des mécanismes de gestion globale des processus utilise un unique point d'entrée dans le module KERPROC. Cet unique point d'entrée est concevable car la majorité des mécanismes de gestion globale des processus nécessite les mêmes informations, en particulier les valeurs des registres processeur. La mise en œuvre de ce point d'entrée au sein du système est détaillée dans le paragraphe 10.3.2.1. Pour obtenir ces informations un mécanisme a été mis en place pour retarder la réalisation des mécanismes mis en œuvre dans KERPROC à un moment au cours duquel le noyau se trouve dans le contexte noyau du processus ciblé. Pour retarder la réalisation des mécanismes, il est nécessaire de marquer les processus. Dans le système KERRIGHED (comme dans Linux), un processus est représenté par un ensemble d'informations : son espace d'adressage, sa liste des fichiers ouverts, l'état des registres processeur et sa pile. La majorité de ces informations est disponible depuis la structure noyau propre à chaque processus (*task_struct*). Un champ (nommé *need_aragorn*) a été ajouté à la structure du noyau regroupant les informations d'un processus (structure nommée *task_struct*).

Le fonctionnement du module KERPROC nécessite de stocker des informations propres à la gestion des processus par KERPROC. Ces informations sont stockées dans une structure de donnée appelée *aragorn_struct* et a été ajoutée à la structure de donnée où sont stockées les informations propres à la gestion des processus par le système KERRIGHED. Pour minimiser les modifications du noyau LINUX, une seule structure a été ajoutée à la *task_struct* d'un processus. Cette structure est appelée *krg_task*.

Les principales informations stockées dans la structure *aragorn_struct* sont :

1. l'identifiant du mécanisme (*aragorn_action*) que l'on souhaite exécuter lorsqu'un processus est marqué en attente grâce à la structure *need_aragorn*,
2. les valeurs des registres processeur,
3. les informations des threads si l'application est multi-threadée (voir paragraphe 10.4).

Pour passer dans un état d'attente et entrer dans le module KERPROC, il faut :

1. passer à 1 le champ *need_aragorn* de la *task_struct* et décrire l'action à effectuer en positionnant le champ *aragorn_action*. Cette opération est réalisée par la fonction *set_aragorn_action*, qui prend comme paramètre l'action à effectuer.
2. le noyau teste le champ *need_aragorn* d'un processus lorsque celui-ci est ordonnancé, revient d'un appel système, d'une interruption ou d'un traitement d'exception (voir fichier *arch/i386/kernel/entry.S*). Le moment choisi est le même que celui utilisé par le système pour gérer le traitement des signaux. Si *need_aragorn* est non nul, KERPROC fait appel à la fonction *start_aragorn* du module ARAGORN.
3. la fonction *start_aragorn* consulte les structures de données du processus grâce à la fonction *get_aragorn_action*, et effectue l'action correspondante.

10.3.1.3 Etats d'un processus au sein de Kerrighed

Selon les fonctionnalités de KERRIGHED utilisées, un processus LINUX peut être dans l'un des états suivants :

1. *Natif* : c'est l'état par défaut des processus qui correspond à un processus qui n'a utilisé aucune fonctionnalité de KERRIGHED directement ou indirectement. Dans ce cas, le champ `krg_ops` de la `task_struct` décrivant le processus au sein du système est égal à `NULL`, de même que le champ `krg_task`. Le processus n'est pas décrit dans */proc/kerrighed*.
2. *Kerrighedisé* : c'est le cas d'un processus qui a utilisé au moins une fonctionnalité propre à KERRIGHED. Sa structure `krg_task` est complètement initialisée, il est décrit dans */proc/kerrighed* et la structure `krg_ops` est initialisée de façon à ce qu'à la destruction du processus par le noyau, la structure `krg_task` soit détruite. En revanche, sa mémoire n'est pas liée à des conteneurs, mais la structure dédiée au module aragorn (`aragorn_struct`) a été initilisée par l'appel à la fonction `init_aragorn_struct` du module. Cependant, certaines informations liées à la gestion des processus, en particulier le numéro de processus et de thread KERRIGHED sont mémorisés dans la `krg_task` et initialisées par le module KERTOOLS.
3. *Conteneurisé* : c'est le cas d'un processus *Kerrighedisé* dont la mémoire a été liée à des conteneurs.

Pour être utilisés par KERPROC, les processus doivent au moins avoir été *Kerrighedisés*, ce qui est rendu possible par le service `ARAGORN_MAKE_KERRIGHED_PROCESS`

Le paragraphe 10.4 détaille l'état d'un processus propre à la gestion des threads dans KERRIGHED.

10.3.2 Mise en œuvre du mécanisme de processus fantôme

La création du processus fantôme est composée de trois étapes : l'extraction de l'espace d'adressage, l'extraction des données privées du processus, et l'extraction de la valeur des registres processeur.

10.3.2.1 Extraction des valeurs des registres processeur

Au sein du noyau Linux (dont KERRIGHED est une extension), la valeur des registres processeur n'est pas disponible en permanence. Ces valeurs ne sont disponibles que dans trois cas : lors d'un retour d'appel système, d'exception ou d'interruption (voir figure 10.2). Ces trois cas aboutissent dans un état du système où les valeurs des registres processeur sont placées sur la pile du processus courant. Cet état intervient juste avant qu'un processus ne devienne actif : ces registres sont sur la pile du processus mais ne sont pas chargés dans le processeur (le chargement est effectué par le noyau LINUX par la fonction *RESTORE_ALL*), alors que son contexte est déjà considéré comme étant celui du processus courant. C'est également dans cet état système que les signaux en attente sont

traités (le contexte du processus courant est celui du processus considéré, toutes les informations système du processus sont donc accessibles, et les valeurs des registres processeur sont disponible). Comme pour les mécanismes de gestion globale des processus, les signaux doivent pouvoir être envoyés à tout moment et la valeur des registres processeur est nécessaire.

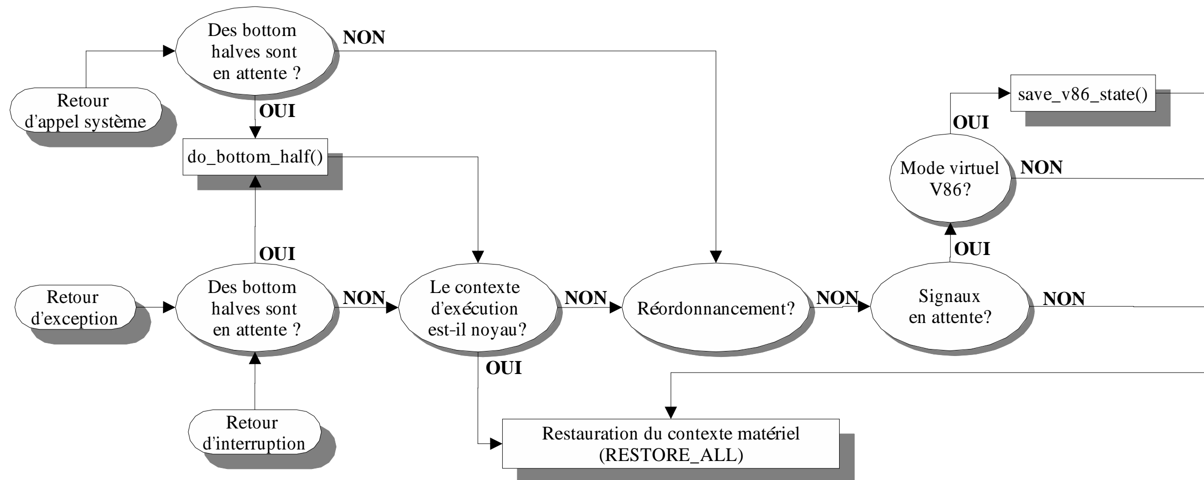

FIG. 10.2 – Fonctionnement du noyau LINUX lors d'un retour d'appel système, d'exception ou d'interruption

Pour obtenir la valeur des registres, et étant donné que la problématique pour les mécanismes de manipulation globale des processus est similaire à celle des signaux, un nouvel état, similaire à celui du traitement des signaux a été créé (voir figure 10.3). Les modifications du noyau nécessaires à la mise en œuvre de ce nouvel état sont très légères : la définition de l'état est très courte (mais en assembleur donc relativement complexe à mettre en œuvre), et un champ interne est ajouté pour chaque tâche. Ce champ interne permet de marquer un processus comme étant en attente d'une manipulation globale par KERRIGHED, tout comme les processus sont marqués lorsqu'un signal leur a été délivré et est en attente de traitement. Cette approche permet donc de pouvoir déclencher à tout moment l'une des opérations que nous avons définies, même si le processus ciblé par cette opération est inactif. De plus, l'état utilisé pour effectuer une manipulation globale d'un processus garantit que le processus n'a aucun signal en attente et que celui-ci est actif (il n'est donc pas par exemple dans une barrière de synchronisation), évitant ainsi grandement les cas critiques pour toute manipulation globale de processus.

10.3.2.2 Extraction des données d'un processus

Il y a deux types d'information pour gérer la mémoire d'un processus : d'une part, les tables des pages qui permettent de traduire une adresse virtuelle en une adresse physique et d'autre part, les structures mémoire permettant d'accéder aux pages en mémoire physique.

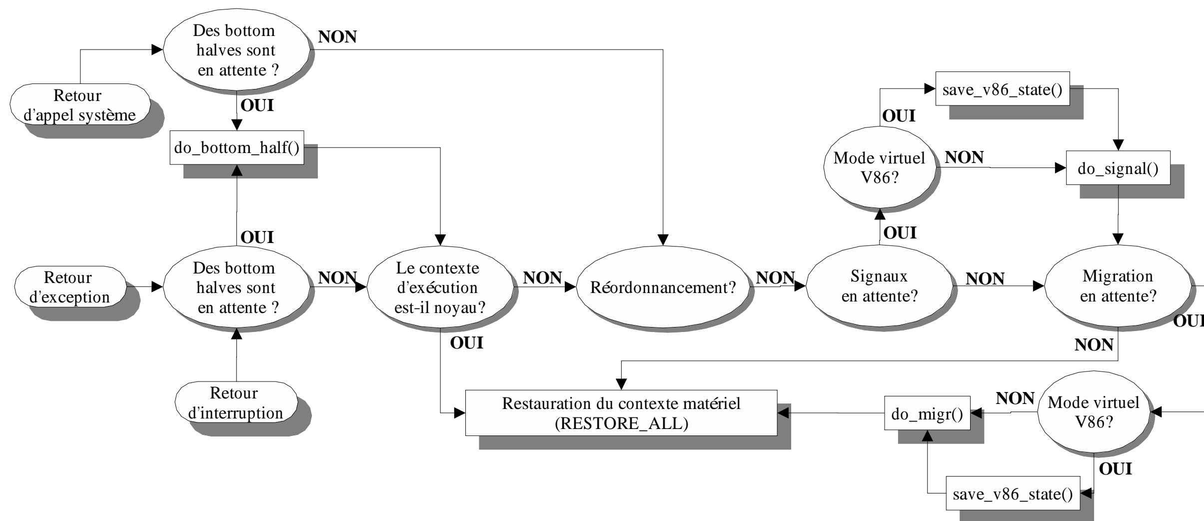

FIG. 10.3 – Fonctionnement du noyau lors d'un retour d'appel système, d'exception et d'interruption après modification afin d'obtenir la valeur des registres processeur pour les mécanismes de gestion globale des processus de KERRIGHED

La structure noyau **mm_struct** gère les informations générales sur l'espace d'adressage du processus. Les segments mémoire sont gérés par la structure **vm_area_struct** et tous ces segments (espace d'adressage linéaire) sont associés à des pages de mémoire physique. Dans KERRIGHED, certains de ces **vm_area_struct** sont liés à des conteneurs (cf. figure 10.4), ce lien permettant d'accéder à des pages mémoire distantes (mettant ainsi en œuvre la MPR).

Les structures mémoire du noyau permettant de gérer les fichiers ouverts par un processus sont plus complexes. Il y a trois structures noyau différentes : les **inode**, les **dentry** et les structures **file**. Les structures **inode** permettent d'accéder physiquement à un fichier sur disque. Mais plusieurs processus peuvent accéder simultanément au même fichier. Les processus manipulent donc des structures **file** lors de leurs accès fichier, plutôt que les structures **inode**. Les structures **dentry** permettent de gérer les liens symboliques. Le noyau manipule également deux types de listes : la liste des fichiers ouverts localement au sein du système et la liste des fichiers ouverts par le processus. De plus, certains segments mémoire sont liés à des fichiers (le segment de code par exemple). Dans KERRIGHED, pour accéder à un fichier distant, un **inode** virtuel et un **dentry** virtuel sont créés, puis liés à un conteneur : les données du fichier sont alors accessibles au sein de la grappe (cf. Figure 10.5) à travers la MPR de KERRIGHED.

KERRIGHED permet donc de simplifier grandement l'extraction des données du processus. Pour les informations mémoire, il suffit d'extraire les structures **mm_struct**, **vm_area_struct** et l'identificateur du conteneur associé [89]. Il n'est pas nécessaire de s'occuper des tables de pages, ni de la migration des pages de mémoire physique. Pour les informations fichiers, seule l'extraction des structures **file** et de l'identificateur du conteneur

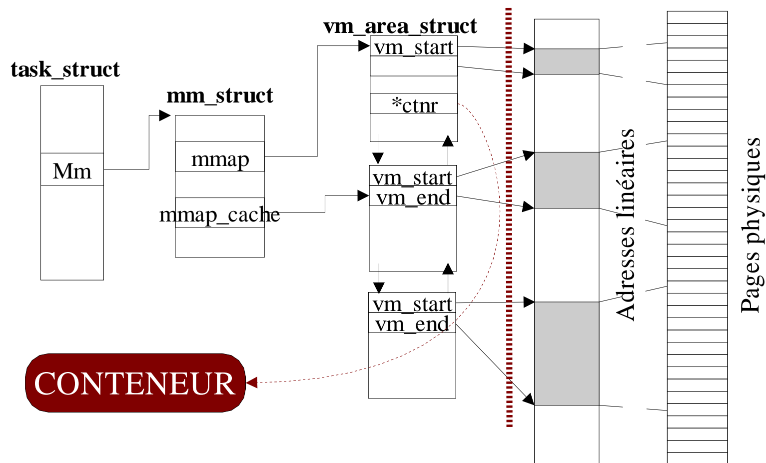

FIG. 10.4 – Représentation noyau de la mémoire associée à un processus avec conteneurs

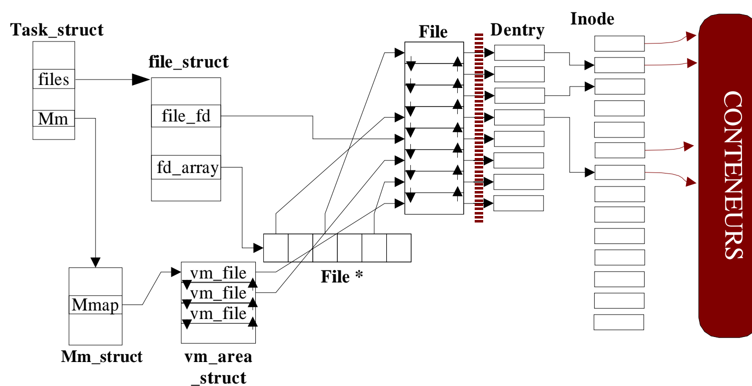

FIG. 10.5 – Représentation noyau des fichiers associés à un processus avec conteneurs

est nécessaire, permettant de faire abstraction des structures **dentry** et **inode**.

10.3.2.3 Transfert des données et insertion du processus au sein du système

Sans les conteneurs, il faut transférer toutes les informations mémoire lors de la migration : les pages de mémoire physique, les tables des pages, ainsi que les informations sur l'espace d'adressage virtuel.

Le mécanisme des conteneurs de KERRIGHED permet d'accéder à une page depuis n'importe quel nœud de la grappe, il n'est donc pas nécessaire de les transférer lors de la migration.

Pour les fichiers, la situation est similaire à celle de la mémoire. Sans le mécanisme des conteneurs, il faut soit déplacer les fichiers manipulés par le processus (très coûteux), soit déplacer les informations propres à chaque fichier dans le cas où le système fonctionne avec un système de fichiers distribué tel que NFS. Dans ce cas, les informations à transférer restent relativement nombreuses : il faut transférer les structures de haut niveau propre au processus (structure noyau **file**), les informations permettant de gérer le cache de fichiers (structure **dentry**) et les informations d'accès au support physique (structure noyau **inode**). Avec le système KERRIGHED, seules les informations sur les structures **file** doivent être transférées, toutes les autres étant gérées de manière transparente par le mécanisme des conteneurs.

Une fois les informations du processus migrées, le processus est inséré au sein du nouveau système hôte, l'image reçue lors de la migration n'étant pas active. Pour cela, on se fonde sur le processus fantôme reçu et comme pour créer un nouveau processus, on utilise les fonctionnalités du noyau (utilisation de la fonction du noyau LINUX **do_fork**). Un processus clone de l'image reçue est donc créé et le processus fantôme peut être détruit.

10.3.2.4 Fonctions pour la manipulation des processus fantômes

Le mécanisme de processus fantôme présenté dans le paragraphe 8.1 permet de manipuler des images de processus indépendamment de leur localisation. Ce mécanisme de processus fantôme est utilisé de façon transparente pour l'utilisateur ou le programmeur de la grappe par les mécanismes de migration, de duplication ou de création de points de reprise de processus.

Néanmoins, pour que les programmeurs puissent développer de nouveaux mécanismes de gestion globale des processus fondés sur les processus fantômes, une interface d'importation/exportation des processus fantômes est offerte.

Les fonctions d'interface permettent de manipuler simplement les processus fantômes, ceux-ci étant représentés au sein du système par une structure *ghost_info_t* (voir figure 10.6). Cette structure regroupe toutes les informations propres à un processus fantôme : (i) son *action*, (ii) sa taille (*ghost_size*) et (iii) la méthode d'accès à la ressource utilisée (*ghost_maker_info_t*). Ce dernier élément regroupe les différents types d'accès aux ressources.

Pour le moment, trois type d'accès ont été mis en œuvre :

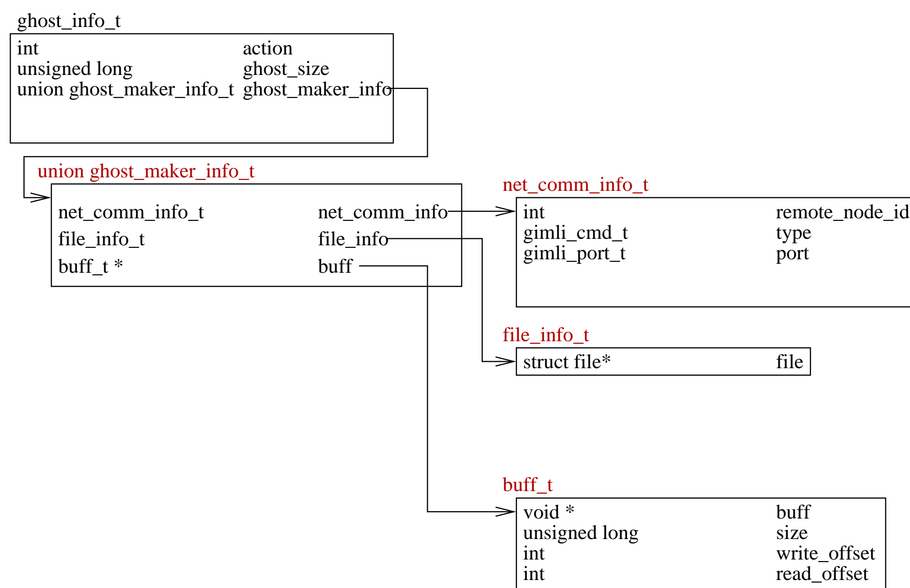

FIG. 10.6 – *Structures de données d'un processus fantôme*

1. accès au réseau via l'interface du système de communication de KERRIGHED. Il est alors nécessaire de connaître l'identifiant du nœud destinataire (*remote_node_id*), le *type* du message et le *port* utilisé pour la communication.
2. accès à un fichier. Il est alors nécessaire de fournir un pointeur sur une structure de donnée noyau propre à un fichier ouvert (*file_struct*). Ce fichier doit donc être ouvert par le noyau avant son utilisation.
3. accès à une zone mémoire. Il est alors nécessaire de fournir un pointeur vers une zone mémoire tampon de KERRIGHED (*buff_t*). Ces zones mémoires sont caractérisées par une zone de mémoire virtuelle allouée (*buff*), par une taille (*size*), par une position logique au sein de la zone mémoire pour l'écriture (*write_buff*) et pour la lecture (*read_offset*). Une interface simple que nous ne détaillons pas ici est fournie dans KERRIGHED et permet de manipuler simplement ces mémoires tampons.

10.3.2.5 Interface d'utilisation des processus fantômes

Le mécanisme des processus fantôme est fourni avec un ensemble de fonctions permettant de manipuler de façon transparente la structure *ghost_info_t*. Il est à noter que pour le moment, le système ne peut manipuler qu'un seul processus fantôme à un instant donné, ce processus fantôme étant appelé *processus fantôme courant*. Cette limitation dûe à un problème de mise en œuvre est en cours de résolution.

10.3.2.5.1 Fonctions permettant d'obtenir des informations sur un processus fantôme

`int get_ghost_maker_action ();`
Cette fonction permet d'obtenir l'action du processus fantôme courant.

`int get_ghost_remote_node_id ();`
Cette fonction permet d'obtenir l'identifiant du nœud distant utilisé lors d'une exportation ou d'une importation d'un processus fantôme via le réseau.

`unsigned long get_ghost_size ();`
Cette fonction permet de récupérer la taille d'un processus fantôme. Cette fonction n'est disponible que pour les processus fantômes associés à une zone mémoire.

10.3.2.5.2 Fonctions permettant de définir des informations d'un processus fantôme

`void set_ghost_maker_action (int action_type);`
Cette fonction permet de définir l'action du processus fantôme courant.

`void set_ghost_net_comm_infos (int destination_node_id, gimli_cmd_t type , gimli_port_t port);`
Cette fonction permet de définir les informations nécessaires au système de communication KerCom pour effectuer une communication point à point.

`void set_ghost_file_infos (struct file *file);`
Cette fonction permet de définir le fichier utilisé pour effectuer un accès fichier pour la manipulation du processus fantôme. L'information à fournir est un pointeur vers une structure noyau *file* obtenue lors de l'ouverture d'un fichier au sein du noyau.

`void set_ghost_buff_infos (buff_t *buff);`
Cette fonction permet d'associer une zone mémoire tampon Kerrighed à un processus fantôme. Pour cela, il est nécessaire de fournir un pointeur vers une telle zone mémoire, ce pointeur pouvant être obtenu par un appel à la fonction `buff_t* create_new_buff (size_t buff_size)` de Kerrighed.

10.3.2.5.3 Fonctions permettant de manipuler les processus fantômes

`int init_ghost_infos ();`
Cette fonction permet d'initialiser un processus fantôme, *i.e.* d'initialiser la structure *ghost_info_t* qui lui est associée.

`int make_ghost (void* data, size_t length);`
Cette fonction permet d'exporter ou d'importer (suivant le type d'accès à une ressource associé) une donnée système dans un processus fantôme.

```
void* make_ghost_of_pointer (void* data,  size_t  length);
```

Cette fonction comme la précédente permet d'exporter ou d'importer une donnée système dans un processus fantôme, celle-ci étant ici un pointeur. Ceci permet principalement de gérer de façon transparente l'allocation de la mémoire nécessaire lors de l'importation de la donnée système.

10.3.3 Mécanisme de migration de processus

Lors de la création d'un nouveau processus, celui-ci est placé sur un nœud choisi par application de la politique d'ordonnancement active au sein de la grappe. Si aucune politique d'ordonnancement n'est active, le système applique par défaut une politique d'allocation « *round robin* » (les nœuds sont sélectionnés tour à tour dans l'ordre croissant des identificateurs). La création à distance par duplication du processus courant est mise en œuvre par la fonction *send_process* et un serveur situé sur chaque nœud de la grappe et qui sait recevoir des processus à travers le réseau grâce à la fonction *g_recv_task*.

10.3.4 Mécanisme de duplication de processus dans Kerrighed

Comme nous venons de le voir, le principe de la migration de processus est de prendre une image d'un processus pour l'envoyer sur un nœud distant, cette image permettant de créer un clone actif du processus initial. Avec cette image, il est possible de mettre en œuvre un mécanisme de duplication de processus à distance, qui permet d'offrir une interface de programmation de type *fork* de manière répartie au sein de la grappe. Les processus créés au sein de la grappe par ce mécanisme ont donc tous le même code, l'exécution suivant le numéro d'identification du processus étant spécifiée par programmation.

Pour la duplication de processus, les procédures d'extraction et de transfert des informations d'un processus restent les mêmes que celles de la migration. En revanche, dans la phase finale, alors que pour la migration, le processus sur la machine initiale est arrêté, il continue ici son exécution.

On peut étendre ce mécanisme à des groupes de processus. Dans ce cas, la phase de transfert de l'image d'un processus est différente : l'image du processus est prise et envoyée simultanément sur plusieurs machines distantes. Un tel mécanisme permet de créer plus efficacement des groupes de processus, en profitant par exemple de caractéristiques de certains types de réseaux tel que le *broadcast* Ethernet. Ceci facilite grandement et rend plus performant le déploiement d'applications parallèles sur une grappe.

10.3.5 Mécanisme de création distante de processus

La création distante de processus est un mécanisme différent de la migration de processus ou de la duplication de processus car nous avons vu qu'il n'est pas nécessaire de transférer l'image d'un processus.

Nous avons également vu dans le paragraphe 8.2.1.2 que deux approches étaient possibles : une approche en mode utilisateur et une approche noyau. Ces deux approches ont été mises en œuvre dans le système KERRIGHED.

10.3.5.1 Approche en mode utilisateur

La création de processus par une approche utilisateur se déroule en deux phases :

1. récupération d'un identifiant de nœud distant où créer le processus par interrogation de l'instance de l'ordonnanceur global s'exécutant localement. Pour cela, un appel système a été créé.
2. lancement d'une commande de création distante de processus (*i.e.* « *rsh* » dans le prototype), avec référence aux fichiers binaire de l'application et paramètres de lancement.

Cette approche est très simple à mettre en œuvre mais elle ne permet pas de contourner le coût de fonctionnement de « *rsh* ».

10.3.5.2 Approche noyau

Pour la mise en œuvre de l'approche noyau, un appel système a été créé pour déclencher la création distante de processus. Cet appel système recherche l'identifiant d'un nœud distant où créer le processus puis expédie une requête de création vers ce nœud. Cette requête comprend une référence vers le fichier binaire de l'application à lancer et l'ensemble des paramètres de lancement. Le nœud distant reçoit la requête et crée un nouveau processus grâce aux informations fournies avec la requête. La création de processus utilise le code noyau utilisé pour la création du premier processus du système lors de son démarrage, évitant ainsi une duplication de processus au sein du système local pour ensuite recouvrir l'image du processus créé par celle du processus à lancer.

10.3.6 Mécanisme de création de points de reprise

La mise en œuvre de mécanismes de point de reprise pour les applications séquentielles est simple en utilisant les processus fantôme. En effet, par définition, un processus fantôme, s'il est stocké peut servir de point de reprise. Le mécanisme de point de reprise associe donc un accès à la ressource disque ou mémoire au processus fantôme.

Afin de faciliter la gestion des points de reprise, les processus fantômes stockés dans le cadre de la création de points de reprise suivent quelques règles :

- un point de reprise est identifié par le *KPID* du processus auquel il correspond.
- les points de reprise sur disque sont tous stockés dans un répertoire donné, présent sur tous les nœuds de la grappe (répertoire local ou partagé par les nœuds). Chaque fichier contenant un point de reprise est identifié par le *KPID* du processus correspondant.
- les points de reprise en mémoire sont gérés par une liste propre à chaque noeud comportant un pointeur vers les différents points de reprise. Chaque élément de la liste est identifié par le *KPID* processus correspondant.

Le mécanisme de restauration de points de reprise n'est pas très différent de celui de réception d'un processus lors d'une migration : les informations du processus fantôme sont importées depuis un périphérique. Pour la restauration d'un point de reprise stocké en mémoire, le *KPID* permet de retrouver l'emplacement mémoire où le processus fantôme est stocké puis restauré. Pour la restauration d'un point de reprise stocké sur disque, la démarche est la même, le *KPID* permettant dans ce cas de trouver le fichier du point de reprise sur disque.

Ces mécanismes ont été conçus pour des processus séquentiels avec seulement la possibilité de disposer d'un seul point de reprise par processus. Ils ne sont donc pas satisfaisants pour la création de points de reprise pour applications parallèles communiquant par mémoire partagée ou par échange de messages. Ces mécanismes ont donc été modifiés par Ramamurthy Badrinath [10], enseignant-chercheur à l'IIT de Kharagpur en Inde, au cours de son séjour sabbatique à l'IRISA, pour supporter la création de points de reprise pour applications parallèles à mémoire partagée.

10.4 Gestion des threads dans KERRIGHED

Le système KERRIGHED offre un support d'exécution de threads utilisant les conteneurs. Cette gestion de threads est fondée sur des mécanismes de gestion globale de ressources de KERRIGHED comme les conteneurs pour le partage de mémoire ou encore la duplication pour la création de threads. En plus de l'utilisation de ces mécanismes, un ensemble de mécanismes propres à la gestion des threads ont été mis en œuvre.

Nous détaillons dans le paragraphe 10.4.1 les éléments propres à la gestion de threads qui ont été mis en œuvre dans le système KERRIGHED. Nous détaillons ensuite les mécanismes principaux de la gestion de threads dans KERRIGHED. Le paragraphe 10.4.2 présente la création de thread alors que le paragraphe 10.4.3 présente la terminaison de threads. Enfin, nous présentons dans le paragraphe 10.4.4 des fonctions générales propres à la manipulation des threads pouvant être utilisées par les programmeurs système. Ces fonctions permettent d'obtenir les informations principales des threads.

10.4.1 Gestion des threads dans KERRIGHED

La gestion globale de thread dans une grappe nécessite de disposer de mécanismes globaux particuliers. Les principaux mécanismes sont :

1. un mécanisme d'identification des threads,
2. un mécanisme de localisation des threads,
3. un mécanisme de communication entre threads,
4. un mécanisme de gestion des états des threads.

Tous ces mécanismes nécessitent de disposer d'informations propres aux threads (*e.g.* état des threads, localisation des threads). Toutes les informations des threads sont contenues dans une structure de donnée noyau appelée *gthread_info*. La figure 10.7 décrit l'état

d'un processus KERRIGHED relativement aux différents threads qui le composent. Comme chaque thread est un processus LINUX, un processus KERRIGHED doit conserver la liste des autres processus LINUX qui sont en réalité un des threads du processus examiné. Cette liste doit contenir le numéro de processus LINUX et le nœud sur lequel il s'exécute, ainsi que le numéro de thread KERRIGHED auquel ce processus correspond. Or, dans la mise en œuvre actuelle, le processus maître joue un rôle particulier. C'est pourquoi la liste des autres processus est stockée de manière non-uniforme. Les informations décrivant le thread maître sont mémorisées dans la structure `master_infos` de la structure KERPROC, et les informations concernant les autres processus LINUX du processus KERRIGHED courant sont mémorisées dans une liste décrite par le champ `gthread_infos` de la structure *aragorn* (structure utilisée pour stocker les informations propres au fonctionnement de KERPROC). La structure `master_infos` n'est pas pertinente pour un processus mono-thread ni pour le thread maître d'un processus multi-thread. Pour tester si un processus LINUX est un thread d'un processus KERRIGHED, il faut utiliser la fonction `process_has_multiple_threads` qui teste les différents cas de figure qui résultent de cette structuration : un processus LINUX est un thread d'un processus KERRIGHED s'il a sa structure `master_infos` initialisée ou s'il y a des threads décrits par la liste `gthread_infos`.

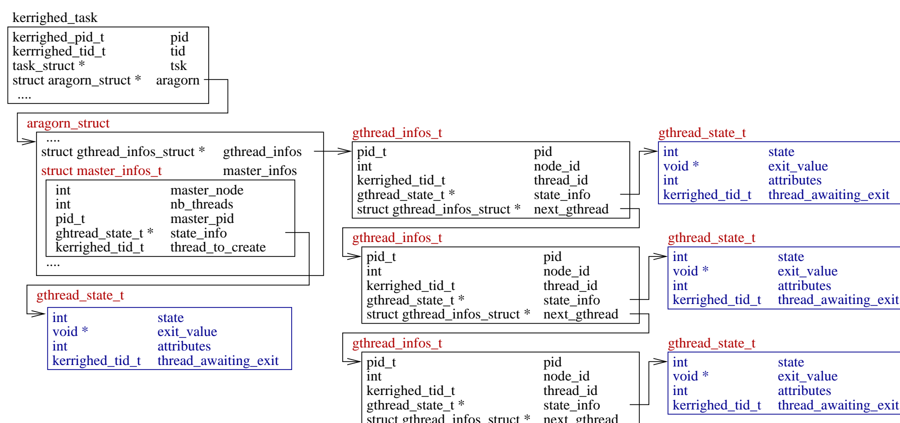

FIG. 10.7 – *Structure de donnée associée à un processus pour la gestion de l'état de ces threads*

L'état de chaque thread du processus KERRIGHED est mémorisé uniquement par le thread maître dans les structures de type `gthread_state_t` se rapportant à chaque description de thread. Cet état correspond aux différents états d'un thread nécessaires à la mise en œuvre de la norme POSIX pour les threads, et décrits dans la figure 10.8. Nous reviendrons sur les messages échangés lors des transitions dans le paragraphe dédié à la mise en œuvre de l'interface *pthread* au dessus du module KERPROC.

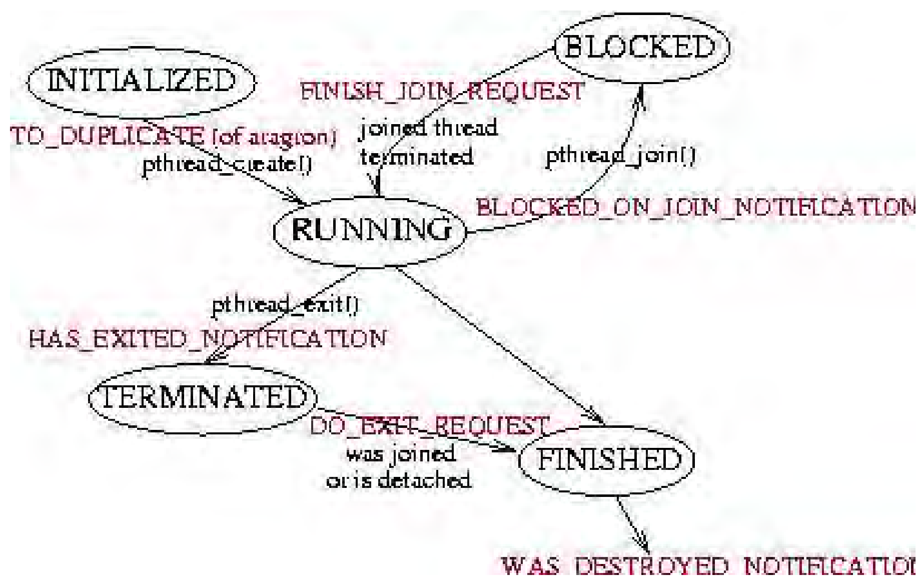

FIG. 10.8 – *Diagramme d'état d'un thread, et messages échangés lors des transitions*

10.4.2 Création d'un threads

La fonction de création de thread repose sur la duplication de processus. La duplication de processus avec la gestion des informations propres aux threads est mise en œuvre par la fonction *aragorn_ create_ gthread*. Cette fonction permet donc de mettre en œuvre la fonction *pthread_ create*. Elle est mise en œuvre en deux temps : la fonction *aragorn_ create_ gthread* qui prépare les paramètres pour la création effective puis qui appelle *gthread_ create* pour déclencher une duplication en positionnant les variables pour un prochain appel à *start_ aragorn*.

La préparation des paramètres consiste en la création et l'insertion dans les structures de données du processus appelant d'une structure `gthread_info_struct` qui contient :

1. le numéro du thread à créer,
2. la taille de la pile du thread à créer si celle-ci n'a pas été fournie par le processus appelant,
3. l'adresse de la pile du thread à créer si celle-ci n'a pas été fournie par le processus appelant. Celle-ci est calculée de façon à placer la nouvelle pile juste en dessous de la pile du thread précédent.

Dans cette version de l'appel, on suppose que c'est le thread maître qui crée un nouveau thread, et qu'on ne cherche pas à réutiliser l'espace utilisé par la pile d'un thread qui aura terminé son exécution.

La section correspondante de *start_ aragorn* initialise la nouvelle pile avec les valeurs de l'ancienne. Pour que le retour de l'appel système puisse se faire sans problème, il faut en

plus parcourir la nouvelle pile pour recalculer les fenêtres de pile[1]. Une fois la nouvelle pile correctement initialisée[2], on change la valeur des registres de la copie du processus LINUX, de façon à ce que la version copiée du processus LINUX démarre en utilisant cette nouvelle pile. La copie est alors envoyée sur un autre nœud de la grappe, l'état du processus tel que vu par le thread maître comme étant en cours d'exécution et les informations liées au nouveau thread sont diffusées à tous les threads déjà créés.

10.4.3 Terminaison d'un thread

La terminaison d'un thread dans la norme POSIX est mise en œuvre par la fonction *pthread_exit*. Pour correspondre à la norme POSIX, la mise en œuvre de la fonction *pthread_exit* impose que (i) les ressources liées au thread ne soient pas libérées avant que le thread soit rejoint par l'ensemble des autres threads de l'application (*i.e.* que tous les threads n'exécutent *pthread_join*) et que (ii) si le thread appelant est le dernier thread actif, le programme se termine, une mise en œuvre distribuée est souhaitable et plus efficace. Par conséquent, KERPROC fournit deux services correspondant à ces deux appels définis pour les threads POSIX et mettant en œuvre en partie les mécanismes de terminaison des threads. De plus, afin de détecter la terminaison d'un thread, le noyau LINUX a été modifié pour appeler *__aragorn_front_exit* lors de la terminaison d'un thread. Enfin, la gestion de l'attribut de thread `GTHREAD_DETACHED` complète la liste des mécanismes système utilisés pour mettre en œuvre les fonctionnalités d'une bibliothèque de thread de style POSIX fondée sur KERPROC.

La gestion de l'état des threads est documenté par la figure 10.8 : le processus LINUX correspondant au thread maître est le seul processus à tenir à jour dans le noyau l'état d'un thread dans la structure de données `state_info` (voir la figure 10.7). Chaque appel à un service qui nécessite la consultation ou la modification de cet état se traduit par l'envoi d'un message au processus LINUX correspondant au thread maître (les messages `BLOCKED_ON_JOIN_NOTIFICATION` et `HAS_EXITED_NOTIFICATION`) et par la supension du processus. De plus, la terminaison d'un thread conduit à l'envoi du message `WAS_DESTROYED_NOTIFICATION` de manière à ce que le processus maître soit averti de la terminaison d'un thread et puisse déterminer si elle était prévue ou si elle doit conduire à la terminaison du processus Kerrighed et donc de tous les threads.

À la réception de ces messages, le processus maître examine l'état du thread ayant émis la requête, stocke l'éventuelle valeur de terminaison et envoie aux threads concernés des requêtes appropriées (`DO_EXIT_REQUEST` ou `FINISH_JOIN_REQUEST`) pour autoriser la terminaison ou pour les débloquer s'ils étaient bloqués en attente de la terminaison d'un autre thread.

[1] Les fenêtres de pile peuvent être calculées grâce aux valeurs du registre processeur du pointeur sur pile (le registre *esp* sur les architectures x86) et du registre processeur contenant l'adresse de l'ancienne de pile (le registre *ebp* sur les architectures x86).

[2] La méthode d'initialisation choisie n'est pas la seule possible : on pourrait essayer de construire une pile fraîche, mais le code produit serait alors beaucoup plus difficilement maintenable.

10.4.4 Interface générale propre aux applications constituées de threads

Nous avons défini une interface permettant d'obtenir des informations propres aux applications constituées de threads. Cette interface permet de savoir si une application est une application multi-threadée, si un thread est le thread maître ou un simple thread , ou encore de trouver les informations propres à un thread donné d'une application parallèle.

```
int is_a_thread (struct task_struct *) ;
```

La fonction *is_ a_ thread* permet de savoir si un processus est un thread d'une application parallèle.

```
int is_the_master_thread (struct task_struct *) ;
```

La fonction *is_ the_ master_ thread* permet de savoir si un processus est le processus maître d'une application parallèle.

```
int process_has_multiple_threads (struct task_struct *p) ;
```

La fonction *process_ has_ multiple_ threads* permet de savoir si l'application représentée par un processus est une application parallèle communiquant par mémoire partagée.

```
gthread_infos_t* find_gthread_by_thread_id (kerrighed_tid_t gthread_id,
struct task_struct *thread) ;
```

La fonction *find_ gthread_ by_ thread_ id* permet d'obtenir un pointeur vers les informations propres à un thread donné (*gthread_ id*).

10.5 Résumé

L'architecture modulaire d'ordonnanceur global et les mécanismes de gestion globale des processus ont été mis en œuvre au sein du système KERRIGHED. La version actuelle de KERPROC offre des mécanismes de migration de processus (avec différentes options comme la migration à la demande, l'envoi des processus en un seul message), la duplication de processus et le création/restauration de points de reprise. De plus, le développement de nouveaux mécanismes de gestion globale de processus est facilité par le mécanisme de processus fantôme, mécanisme unique permettant de gérer les processus indépendamment de leur localisation.

Cette mise en œuvre est légère en terme de modifications du noyau LINUX (une centaine de lignes) grâce à la mise en œuvre d'un seul point d'entrée utilisé par la majorité des mécanismes de gestion globale des processus.

L'utilisation du mécanisme de gestion globale des ressources blocs conjointement à la gestion globale des processus a permis d'étendre les mécanismes à la gestion globale de threads. La gestion globale des ressources blocs permet également de simplifier et d'optimiser les mécanismes de gestion globale des processus en proposant une migration de l'espace d'adressage et des informations fichiers à la demande.

11 ÉVALUATION DE PERFORMANCES

Afin de valider nos travaux, nous avons conduit une étude des performances des mécanismes d'ordonnancement global de processus que nous avons conçus et mis en œuvre dans le SSI KERRIGHED. Cette évaluation de prformance comporte deux volets. Le premier concerne l'évaluation des mécanismes de gestion globale des processus utilisés par l'ordonnanceur global. Le second est relatif à l'évaluation de l'architecture d'ordonnancement global adaptable.

Après avoir présenté la plate-forme et les applications utilisées pour l'évaluation de performance, nous présentons les résultats obtenus pour les différents mécanismes de manipulation globale des processus puis nous présentons les résultats obtenus pour l'ordonnancement global. Nous terminons par un résumé.

11.1 Plate-forme d'expérimentation

L'architecture cible sur laquelle le système KERRIGHED est mis en œuvre est la grappe de calculateurs PARASKI. Cette grappe est composée de 40 PCs bi-processeurs de générations différentes. Les nœuds les plus anciens sont équipés de processeurs Pentium Pro cadencés à 200 MHz et de 128 Mo de mémoire centrale, tandis que les nœuds les plus récents sont équipés de processeurs Pentium III cadencés à 1 GHz et de 512 Mo de mémoire centrale.

La technologies d'interconnexion utilisée est un réseau Fast Ethernet.

Deux types de mesures ont été effectuées : (i) une évaluation des mécanismes de gestion des processus (ii) une évaluation des mécanismes d'ordonnancement global. Les systèmes KERCOM offre une latence de 61 micro-secondes et la figure 11.1 montre le débit réseau en fonction de la taille des données à transmettre obtenus avec le système de communication KERCOM de KERRIGHED.

11.2 Description des applications de test

Les tests ont été réalisés avec un ensemble d'applications, la plupart étant des applications du monde industriel fournies par EDF R&D, en plus d'applications synthétiques et d'applications de benchmarking. Les applications industrielles utilisées pour évaluer nos travaux sont de deux types : des applications de type étude paramétrique et des applications parallèles à mémoire partagée. Nous n'avons pas utilisé d'applications parallèles à

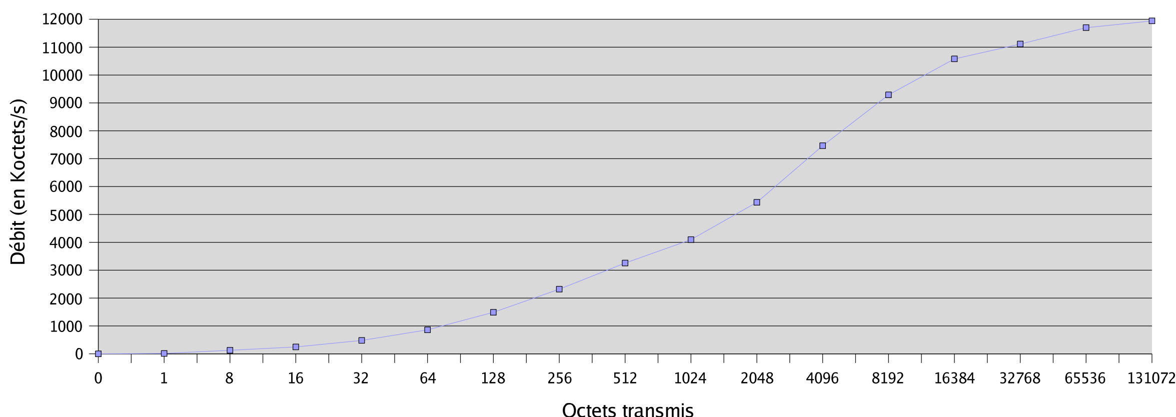

FIG. 11.1 – Débit du système KERCOM pour un réseau de type Fast Ethernet

échange de message, les travaux de thèse de Pascal Gallard sur la gestion globale des flux de données n'étant pas encore intégrés au prototype.

L'application *Gram-Schmidt* est un algorithme produisant une base orthonormée de l'espace généré par un ensemble de vecteurs indépendants, grâce à l'algorithme de *Gram-Schmidt* modifié. À chaque itération, un nouveau vecteur de la base orthonormée est calculé. À chaque fois qu'un processeur produit un vecteur, ce dernier est lu par chacun des autres processeurs. Ce vecteur est alors utilisé afin de corriger les vecteurs restant à normaliser dont chaque processeur a la charge. La distribution des calculs est réalisée grâce à une distribution cyclique des colonnes sur les différents processeurs. Chaque processeur i effectue les calculs intervenant sur les vecteurs v tels que $v \; modulo \; p = i$. Cette application est disponible en version séquentielle (appelée *up-gs*) et en version « *pthread* » (appelée *mgs*).

L'application *Ecoss* [16] est une application de EDF R&D mettant en œuvre un modèle de relaxation homogène pour calculer les écoulements critiques d'eau subissant le phénomène de flashing (évaporation à la sortie des soupapes de sécurité de centrales nucléaires). L'application est lancée en fournissant en paramètre le nombre de points de l'étude et la durée du phénomène simulé. Cette application est disponible en version séquentielle (Fortran et quelques lignes de C pour la mesure de temps), en version OPENMP (Fortran77 avec quelques lignes de C pour la mesure de temps), en version MPI et en version HPF. Une série de mesures a été effectuée par EDF sur différentes plate-formes (PC, grappe de calculateurs, Compaq Proliant, SGI Origin 2000, T3E ou encore HP Class V)[16].

L'application *Cyrano3* est une application d'étude thermo-mécanique du crayon combustible utilisée par EDF en production. Cette application est de type paramétrique : après dégrossissement des cas les plus dimensionnants, plusieurs scénarios correspondant chacun à un crayon et à un historique sont lancés. L'application est fournie avec divers tests, chacun consommant diverses quantités de mémoire (de 1,9 à 3,7 mega-octets).

11.3 Évaluation des mécanismes de gestion globale de processus

Le prototype actuel de KERRIGHED offre des mécanismes de migration de processus, de duplication de processus et de création/restauration de points de reprise, chacun de ces mécanismes se fondant sur le mécanisme de processus fantôme. Une évaluation de chacun des mécanismes a été effectuée en évaluant le coût de construction du processus fantôme, avec notamment le coût d'accès à la ressource utilisée. Pour cette évaluation, nous avons utilisé les nœuds équipés de processeurs Pentium III cadencés à 1 GHz et équipés de 512 Mo de mémoire centrale. La technologie d'interconnexion utilisée est le réseau Fast Ethernet.

Pour chaque évaluation, nous avons effectué un minimum de cinq mesures pour ensuite calculer une moyenne.

11.3.1 Évaluation du mécanisme de processus fantôme

Le processus fantôme est le mécanisme utilisé par les mécanismes de migration, de duplication et de création/restauration de points de reprise. La taille d'un processus fantôme est particulièrement importante car elle influe directement sur les performances globales des mécanismes s'appuyant sur les processus fantômes.

Pour calculer la taille d'un processus fantôme, nous séparons l'espace utilisé pour le stockage de l'espace d'adressage de celui utilisé pour les autres données du processus (appelé *partie privée* du processus fantôme). La taille de l'espace d'adressage varie principalement en fonction de la taille des données manipulées par la tâche, alors que la taille de la partie privée du processus fantôme varie principalement en fonction du nombre de fichiers utilisés par la tâche.

L'utilisation des conteneurs permet de diminuer la taille d'un processus fantôme, son espace d'adressage n'étant alors pas inclus dans celui-ci.

Le paragraphe 11.3.1.1 montre la taille des processus fantômes pour différentes applications sans utilisation des conteneurs, alors que le paragraphe 11.3.1.2 montre la taille des processus fantômes avec utilisation des conteneurs.

11.3.1.1 Taille des processus fantôme sans utilisation des conteneurs

Le tableau 11.1 indique la taille du processus fantôme de l'application *up-gs* en fonction de la taille de la matrice utilisée. La taille du processus fantôme ne varie pas au cours de l'exécution de l'application puisque l'ensemble des données sont chargées à l'initialisation de l'application. Le processus fantôme est en majorité constitué de l'espace d'adressage du processus. Le tableau 11.1 montre que la taille du processus fantôme augmente proportionnellement à la taille de la matrice utilisée. Plus la matrice utilisée est grande et plus l'espace d'adressage du processus est important.

A contrario, la taille des données privées du processus ne varie pas car le nombre de fichiers ouverts ne varie pas.

	Taille de l'espace d'adressage du processus fantôme (en Ko)	Taille de la partie privée du processus fantôme (en Ko)	Taille totale (en Ko)
500x500	4 224	5	4 229
750x750	9 349	5	9 354
1000x1000	12 424	5	12 429
1250x1250	20 624	5	20 629
1500x1500	24 724	5	24 729
1750x1750	28 828	5	28 833
2000x2000	32 928	5	32 933

TAB. 11.1 – Taille du processus fantôme de l'application *up-gs* en fonction de la taille de la matrice utilisée sans utilisation des conteneurs

11.3.1.2 Taille des processus fantôme avec utilisation des conteneurs

Contrairement au processus fantôme d'une tâche n'utilisant pas les conteneurs, le processus fantôme d'une tâche utilisant les conteneurs n'a pas une taille qui varie en fonction de la taille de l'espace d'adressage. Le tableau 11.2 montre que la taille du processus est constante quelque soit la taille de la matrice utilisée. Cela est dû au fait que le processus, quelque soit la taille de la matrice, est constitué du même nombre de segments mémoire et qu'il manipule le même nombre de fichiers.

11.3.2 Création distante de processus

Le mécanisme de création distante de processus est différent des mécanismes utilisant les processus fantôme. Dans le cas de la création distante de processus, seuls une référence au fichier source de l'application et les paramètres de lancement sont à transférer. Nous avons vu dans le paragraphe 8.2.1.2 que deux types de création distante ont été mis en œuvre dans KERRIGHED : une création distante en espace utilisateur et une création distante en espace noyau.

Pour ces deux méthodes, le temps de la création distante ne dépend pas de l'application car dans les deux cas, l'image du processus n'est pas transmise. Seule la réference sur le fichier objet est transmise. En revanche, chaque paramètre de l'application engendre une communication entre les deux nœuds concernés par la création distante de processus. Ces communications impliquent un coût.

Pour évaluer les deux mécanismes de création distante de processus, nous avons utilisé les applications *up-gs* et *Cyrano3* (toutes deux sont des applications séquentielles). Aucun paramètre n'est fourni lors du lancement de ces applications.

Pour ces deux applications, le temps de création distante en espace utilisateur est en moyenne de 800 milli-secondes. Le temps de création distante de processus en espace noyau est inférieur à 10 milli-secondes pour ces deux applications.

	Taille de l'espace d'adressage du processus fantôme (en Ko)	Taille de la partie privée du processus fantôme (en Ko)	Taille totale (en Ko)
500x500	0,9	5,3	6,2
750x750	0,9	5,3	6,2
1000x1000	0,9	5,3	6,2
1250x1250	0,9	5,3	6,2
1500x1500	0,9	5,3	6,2
1750x1750	0,9	5,3	6,2
2000x2000	0,9	5,3	6,2

TAB. 11.2 – Taille du processus fantôme de l'application *up-gs* en fonction de la taille de la matrice utilisée avec utilisation des conteneurs

Le coût important de la solution en espace utilisateur par rapport à la solution en espace noyau est dû au coût du protocole *rsh* utilisé (le protocole *rsh* est fondé sur des communications TCP/IP en mode utilisateur). Le mécanisme de création en espace noyau, quant à lui, dispose d'un protocole plus simple et surtout bénéficie des performances du système KERCOM.

11.3.3 Duplication de processus

Le mécanisme de duplication consiste à extraire le processus fantôme de la machine initiale, puis à l'envoyer sur un nœud distant pour créer un processus clone.

Le mécanisme de duplication est notamment utilisé lors du déploiement des threads d'une application parallèle communiquant par partage de données. L'évaluation de la création distante d'un processus a donc été faite en effectuant des mesures sur la création d'un thread sur un nœud distant.

Les threads partagent leurs données via le mécanisme des conteneurs. Aussi, tous les segments mémoire sont liés à un conteneur, l'espace d'adressage du processus est donc déplacé à la demande lors de l'accès à des pages mémoire.

Pour évaluer le mécanisme de duplication de processus, nous avons mesuré le temps total nécessaire à la création d'un thread, *i.e.* le temps écoulé entre le déclenchement de la duplication et la réception d'un acquittement du nœud distant confirmant la création d'un nouveau processus. Nous avons également évalué le temps de transfert inclus dans le temps de duplication.

La figure 11.2 montre le coût de la création d'un thread sur un nœud distant pour l'application *mgs*. La quantité d'informations à transmettre lors de la duplication étant peu importante (le transfert des informations propres au processus est en grande partie géré par le mécanisme des conteneurs), la duplication de processus est en moyenne égale à 10 milli-secondes. Ce temps de duplication ne dépend pas de la taille de l'espace d'adressage

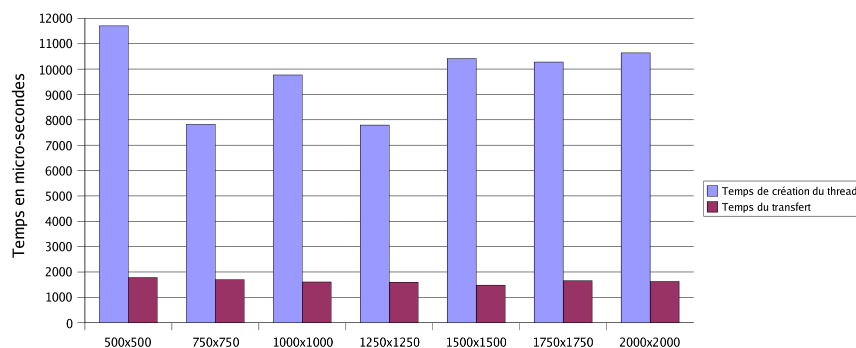

FIG. 11.2 – Temps de création d'un thread de l'application *mgs* sur un nœud distant

du processus puisque celui-ci est géré par les conteneurs et n'est donc pas transféré lors de la duplication. Le temps de transfert est donc constant et de moins de 2 milli-secondes. Le temps de duplication comprend le pré-traitement (exportation des informations), le transfert et le post-traitement (importation des informations et création d'un processus). Les informations à transférer étant peu nombreuses, une grande partie du coût est dûe à l'exportation et à l'importation des informations. Ces deux traitements se recouvrent en partie et représente chacune environ la moitié du coût total de duplication sans le temps de transfert, soit environ 8 milli-secondes.

11.3.4 Migration de processus

Deux protocoles différents pour la migration de processus ont été évalués. Le premier protocole envoie chaque donnée du processus fantôme à travers le réseau pendant l'exportation du processus. Le second protocole stocke le processus fantôme en mémoire dans son ensemble, pour ensuite l'envoyer par un seul accès au réseau sur le nœud distant.

Pour chacun de ces prototypes, une évaluation avec et sans migration à la demande de l'espace d'adressage a été effectuée. Si les segments mémoire d'un processus sont reliés à des conteneurs, ce sont les conteneurs qui déplacent les pages mémoire à la demande, après la migration du processus, lorsque celui-ci est de nouveau actif. Pour chacune de ces évaluations, plusieurs mesures ont été effectuées :

1. *le temps total de migration* : ce temps est le temps pendant lequel le processus n'est pas accessible par le système. Ce temps est particulièrement intéressant car s'il est trop important, le processus est inactif pendant une longue période au sein de la grappe et le système ou d'autres processus peuvent se trouver en attente d'une réponse de processus en cours de migration.
2. *le coût de l'exportation et d'importation du processus fantôme* : ce temps permet d'évaluer l'incidence du type d'accès réseau effectué (qui constitue le principal coût de l'exportation et de l'importation) et permet par exemple de savoir si effectuer des envois pour chaque donnée du processus fantôme est réellement pénalisant.

11.3.4.1 Migration en effectuant un accès réseau pour chaque donnée système

Migration d'un processus sans utilisation de conteneurs L'évaluation du temps total de la migration, *i.e.* le temps écoulé entre le début de la migration et la réception de l'acquittement du nœud distant lorsque la migration est terminée, montre que plus l'espace d'adressage d'un processus est important et plus le temps de migration, et donc le temps d'indisponibilité du processus, est important (voir figure 11.3) . Cette solution est

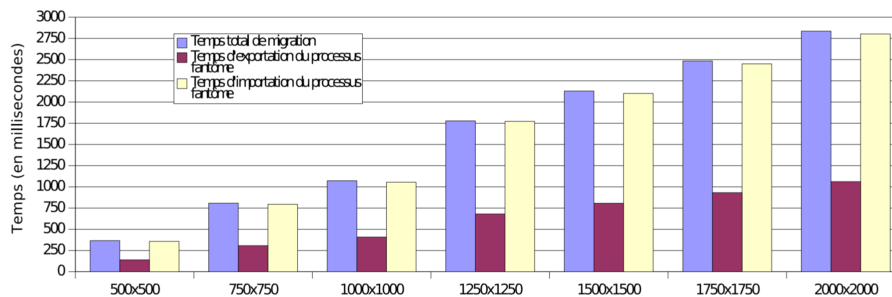

FIG. 11.3 – Temps de migration de *up-gs* sans utilisation des conteneurs

donc envisageable pour des processus ayant un petit espace d'adressage. Le temps total de migration est inférieur à la somme des temps de pré-traitement, d'émission, de réception et de post-traitement des informations nécessaires à la migration car il existe un recouvrement entre le pré-traitement, la transfert et le post-traitement. On voit également que le temps passé pour l'importation est très supérieur à celui passé pour l'exportation. Cela est dû au système de communication qui n'est pas adapté à ce type d'échanges : les envois sont effectués mais le nœud distant n'est pas prêt à recevoir les informations, les informations sont donc copiées dans une zone mémoire pour être ultérieurement remises au processus destinataire, créant ainsi des copies supplémentaires coûteuses. Le nœud initial envoie donc rapidement ces données (temps d'exportation) mais il est ensuite obligé d'attendre l'acquitement du nœud distant. Un système de communication plus adapté de type « *pack/unpack* » devrait permettre d'améliorer ces performances. Un tel système est en cours de développement au sein de KERCOM. Cela permettrait de réduire le temps total de migration puisque la nœud initial est actuellement bloqué dans l'attente de l'acquittement du nœud distant.

Nous avons ensuite évalué le surcoût sur le temps d'exécution de l'application introduit par une migration. Comme le montre la figure 11.4, le surcoût introduit par la migration d'un processus sans l'utilisation de conteneurs est assez faible, moins de 10% dans tous les cas. Toutes les pages mémoire, ainsi que toutes les informations système propres au processus sont localement présentes dès que la migration est terminée, les accès à la mémoire et toute requête système dues à l'exécution du processus (*appel système*) peuvent donc être résolus localement, de manière optimale. Plus le temps d'exécution de l'application est long, plus le surcoût induit par la migration de processus sans conteneurs est faible.

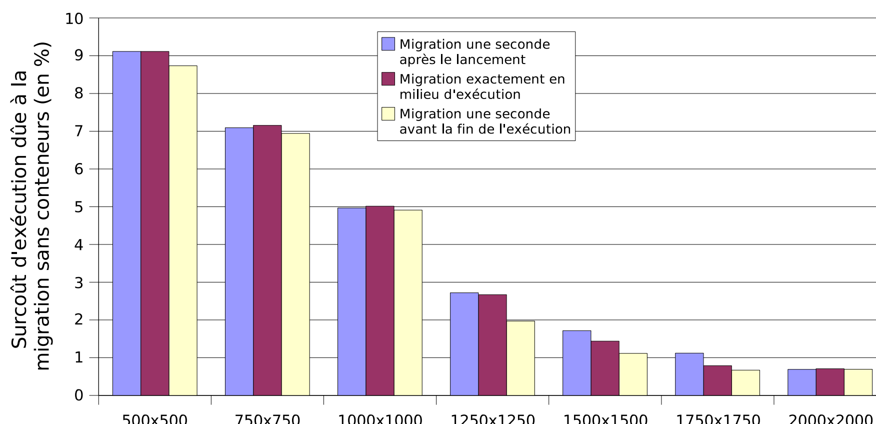

FIG. 11.4 – Surcoût d'exécution dû à la migration de *up-gs* sans utilisation des conteneurs

Migration d'un processus avec utilisation des conteneurs Les conteneurs peuvent être utilisés pour partager de la mémoire, ou pour partager des accès aux fichiers au sein de la grappe. Dans chaque cas, la migration des données système est plus simple et plus rapide que dans le cas où les conteneurs ne sont pas utilisés.

L'évaluation du temps total de la migration pour un processus indépendant utilisant les conteneurs (voir figure 11.5) montre que l'utilisation des conteneurs permet de rendre la migration très rapide : beaucoup moins d'informations système sont à traiter lors d'une migration. Cela permet donc de garantir une migration efficace quelque soit la taille de

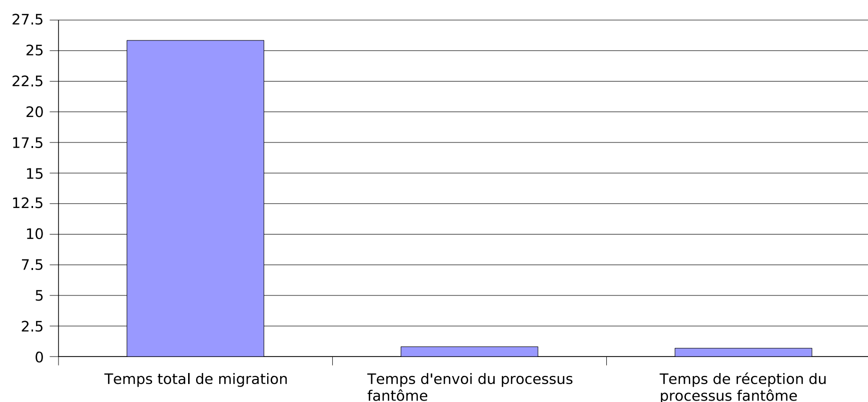

FIG. 11.5 – Temps de migration de *up-gs* avec utilisation des conteneurs

l'espace d'adressage du processus et du nombre de fichier manipulés.

L'utilisation des conteneurs implique un surcoût lors de l'exécution après la migration : chaque page mémoire doit être migrée à la demande, lors de son premier accès sur le nœud accueillant le processus après la migration. Néanmoins, la figure 11.6 montre que le

surcoût d'exécution créé par la migration à la demande des pages mémoires lors de leur accès est peu important par rapport au temps d'exécution de l'application. La migration

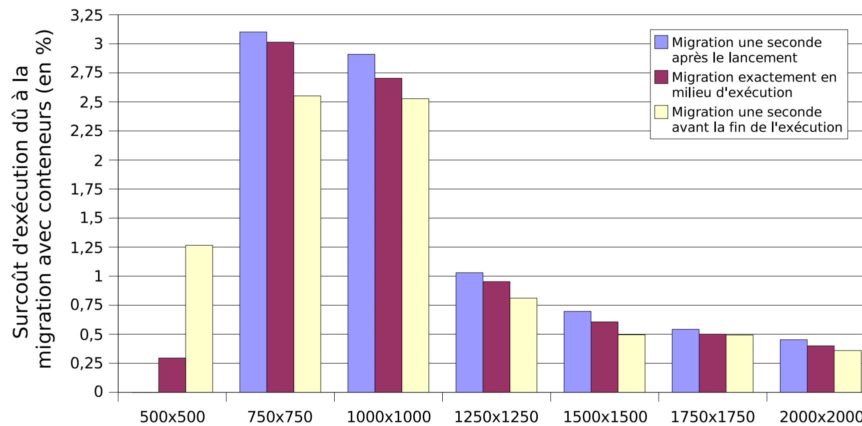

FIG. 11.6 – Surcoût d'exécution dû à la migration de *up-gs* avec utilisation des conteneurs

des pages mémoires à la demande grâce au mécanisme des conteneurs est plus efficace que la migration des pages mémoires via le processus fantôme : les coûts de migration à la demande sont suffisamment faibles pour que le surcoût dû à la migration des pages mémoire via le processus fantôme soit bien plus élevé que celui dû à la migration des pages mémoires à la demande. Les résultats obtenus pour une matrice de taille 500x500 sont certainement dûs au manque de précision des méthodes de calcul utilisé pour obtenir le temps d'exécution de l'application.

11.3.4.2 Migration en effectuant un unique accès réseau pour le transfert du processus fantôme

L'utilisation du protocole de migration par un seul transfert via le réseau vise à diminuer les latences induites par de multiples accès réseau. Ce protocole est donc *a priori* intéressant pour les processus dont la taille du processus fantôme est importante. Les processus n'utilisant pas les conteneurs ont des processus fantôme de grande taille (voir tableau 11.1).

Pour évaluer la migration avec la technique d'un seul accès au réseau, nous utilisons l'application *up-gs* sans utilisation des conteneurs, en utilisant diverses tailles de matrice pour faire varier la taille du processus fantôme. Pour chaque taille de matrice, nous mesurons le temps de transfert, le temps d'importation et d'exportation des processus fantôme (l'importation et l'exportation sont principalement constitués d'accès à une ressource).

La figure 11.7 montre que l'utilisation d'un tampon mémoire pour créer le processus fantôme pour l'envoyer ensuite par un seul envoi n'est pas plus efficace que de faire de multiples envois. En effet, le temps de migration par de multiples accès au réseau est compris entre 300 et 2800 milli-secondes, alors que pour la migration par un seul envoi, le temps de migration est compris entre 500 et 3250 milli-secondes. La migration par un

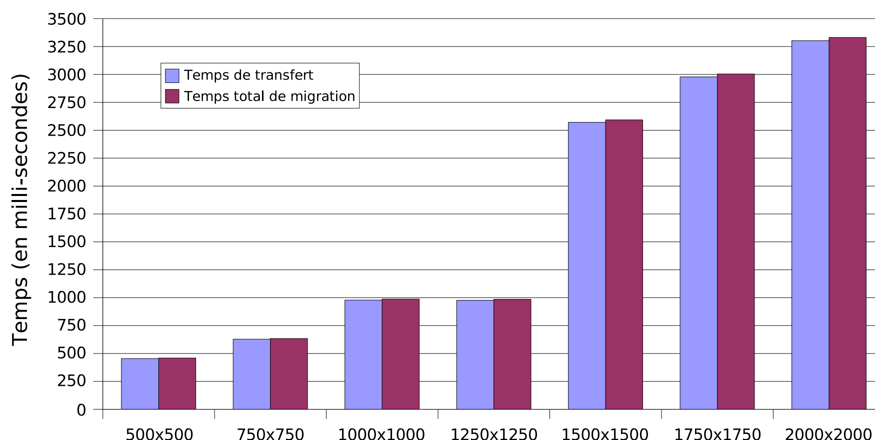

FIG. 11.7 – Temps de migration de *up-gs* avec un seul accès au réseau

seul envoi ne permet d'améliorer les performances de la migration de processus car lorsque le transfert du processus fantôme est effectué par de multiples envois, le traitement des données du processus fantôme peuvent recouvrir certains transferts. En revanche, dans le cas d'un seul envoi, ce recouvrement est impossible et le temps de migration est alors directement lié aux caractéristiques du réseau utilisé.

11.3.5 Création de points de reprise

La performance de la création de point de reprise dépend grandement de la ressource utilisée pour stocker celui-ci. Pour évaluer l'impact de la ressource utilisée, nous avons mesuré le temps de création d'un point reprise (qui dépend de la taille de celui-ci) en fonction de la ressource utilisée. Le temps total de création d'un point de reprise est le temps ecoulé entre le moment où la création d'un point de reprise débute et le moment où le point de reprise est stocké sur une ressource de la grappe (la création du point de reprise est donc achevée). Nous avons sélectionné les ressources suivantes pour effectuer nos mesures : (i) la mémoire locale et (ii) le système de fichier local.

Pour évaluer le coût de construction d'un point de reprise, nous avons utilisé l'application *up-gs*, sans utilisation des conteneurs. La figure 11.1 donne la taille du processus fantôme de cette application en fonction de la taille de la matrice utilisée. Le temps nécessaire à la création du point de reprise de l'application *up-gs* ne dépend pas du moment où la création du point est faite car la taille du processus fantôme ne varie pas pendant l'exécution de l'application.

La figure 11.8 montre que le temps pris pour créer un point de reprise en mémoire est principalement dû au temps passé pour accéder à la mémoire. Contrairement à la migration, les performances du mécanisme de création de points de reprise ne sont pas limitées par la couche d'accès à la ressource utilisée, la création d'un point de reprise d'un processus est environ 10% plus rapide que la migration de ce processus.

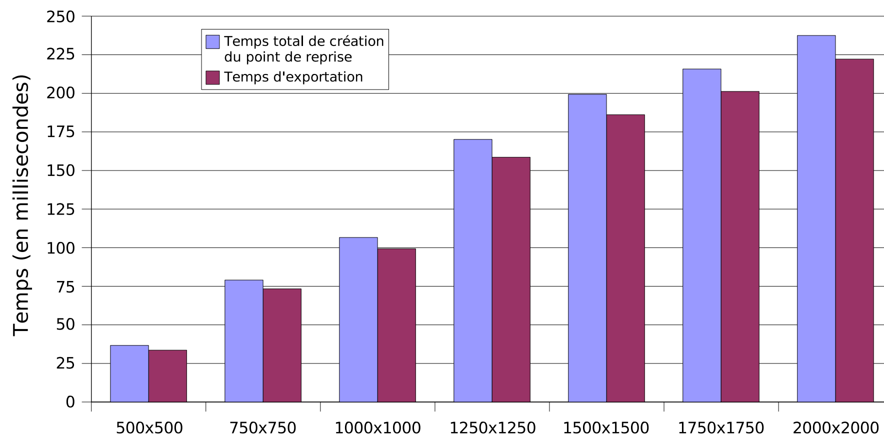

FIG. 11.8 – Temps de création d'un point de reprise en mémoire de l'application up-gs

La figure 11.9 montre le temps pour la création d'un point de reprise sur disque local. On peut voir que le coût d'exportation est plus important que le coût d'exportation lors

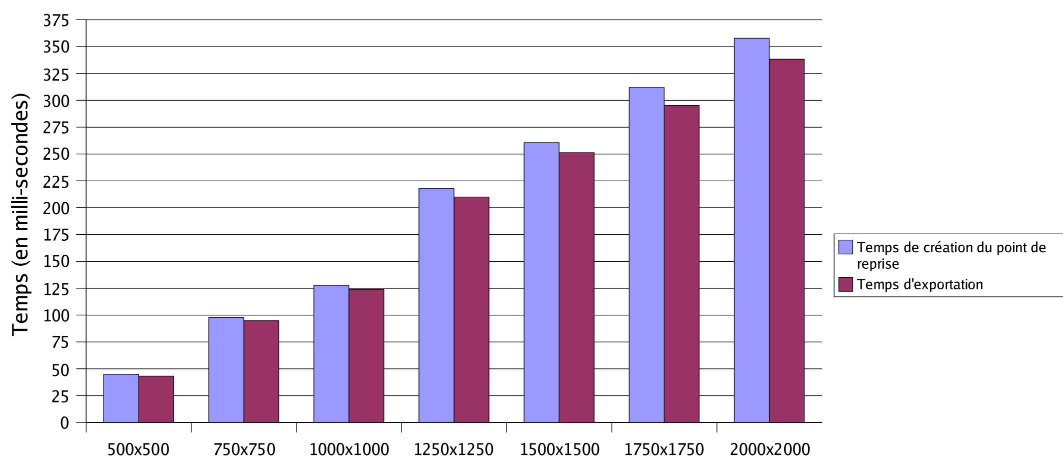

FIG. 11.9 – Temps de création d'un point de reprise sur disque de l'application up-gs

de la création d'un point de reprise en mémoire, les accès disque étant plus coûteux que les accès mémoire.

11.3.6 Restauration d'un point de reprise

Comme pour l'évaluation de la création d'un point de reprise, nous avons utilisé l'application *up-gs* sans utilisation des conteneurs. L'évaluation de la restauration d'un point de reprise a été faite en fonction de la ressource utilisée pour stocker le point de reprise. Pour cela, nous évaluons le temps nécessaire à la création d'un processus depuis un point de

reprise mais également le temps de l'importation du processus fantôme, ce temps dépendant principalement de la ressource utilisée.

La figure 11.10 montre que le temps pour restaurer un point de reprise dépend de la taille du processus fantôme. Les performances de la restauration d'un point de reprise sont d'ailleurs très proches de celles de création du point de reprise en mémoire. La majorité du coût de la restauration d'un point de reprise est dûe à des accès mémoire pour l'importation du processus fantôme.

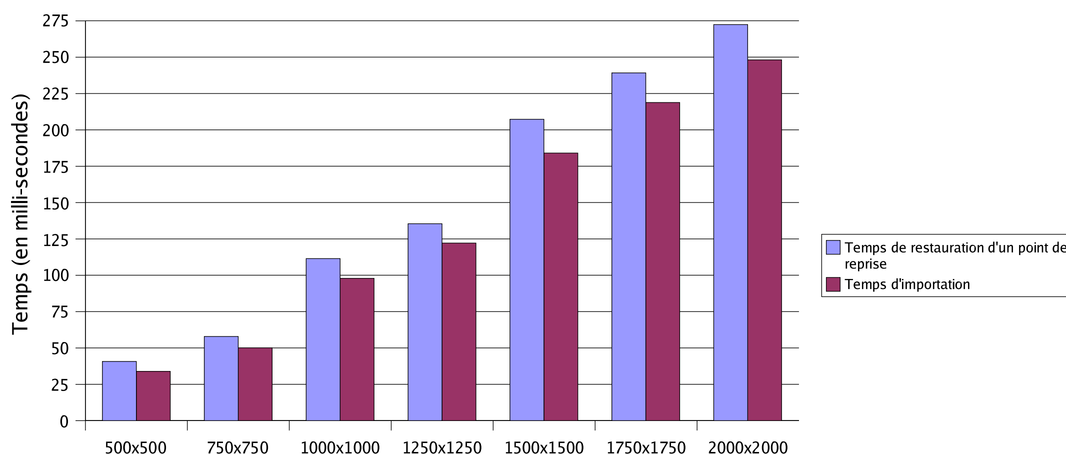

FIG. 11.10 – Temps de restauration d'un point de reprise en mémoire de l'application up-gs

La restauration d'un point de reprise depuis un fichier se fait par un accès aux interfaces d'accès aux fichiers du noyau. Le coût de la lecture d'un fichier dépend donc de l'état du cache de fichier. Il est faible si le fichier se trouve dans le cache et plus important si le fichier ne s'y trouve pas. Nous avons évalué l'influence du cache de fichier sur le mécanisme de restauration de point de reprise. Pour cela, nous avons restauré un point de reprise sur disque immédiatement après sa création (le fichier est donc présent dans le cache de fichier) et après un redémarrage de la grappe (le fichier est donc absent du cache de fichier).

La figure 11.11 montre que les performances de restauration d'un point de reprise disque en utilisant le cache de fichier sont similaires aux performances de la restauration d'un point de reprise mémoire. Le fichier étant en mémoire, l'importation du processus fantôme est rapide et constitue l'ensentiel du coût de la restauration.

En revanche, la figure 11.12 montre que si le fichier n'est pas dans le cache de fichier, le coût de restauration d'un point de reprise disque est environ huit fois plus coûteux que la restauration depuis un point de reprise mémoire. Néanmoins, comme pour la restauration depuis un point de reprise mémoire, l'essentiel du coût de la restauration d'un point de reprise sur disque est dû à l'importation du processus fantôme, *i.e.* aux accès à la ressource utilisée.

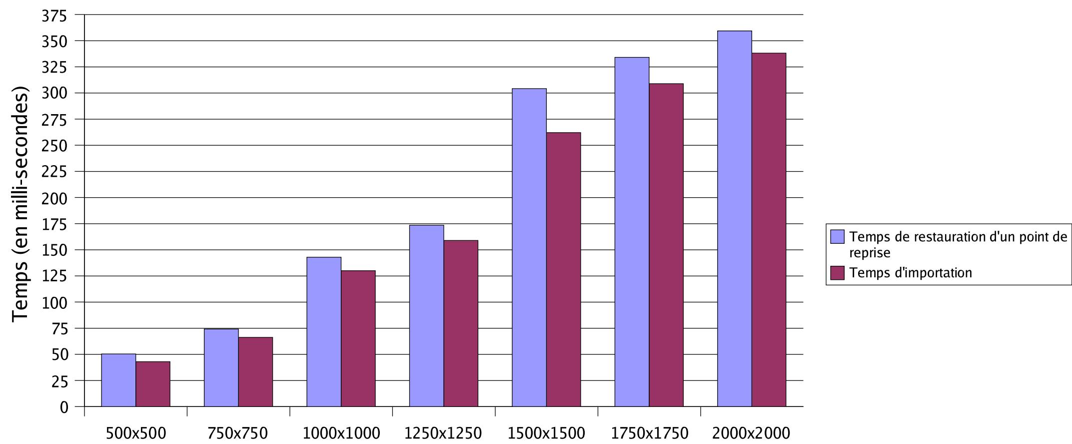

FIG. 11.11 – Temps de restauration d'un point de reprise sur disque de l'application up-gs avec le cache de fichier chaud

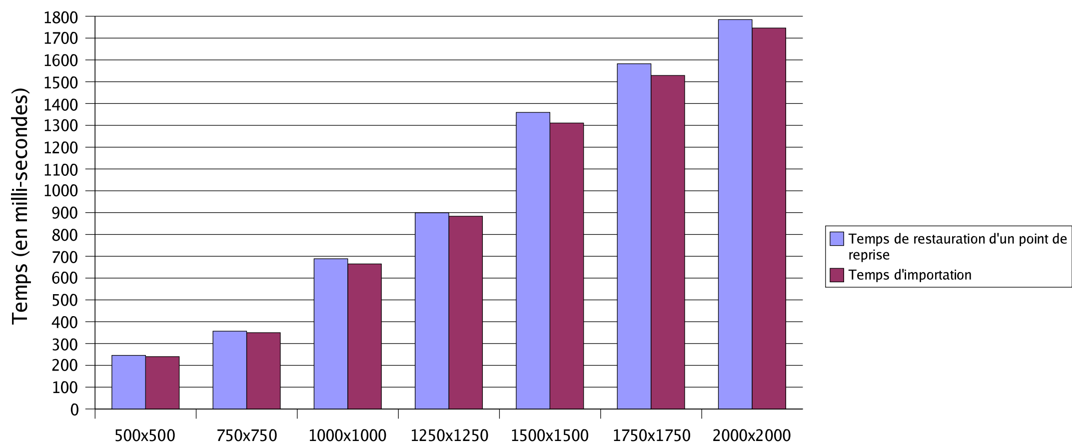

FIG. 11.12 – Temps de restauration d'un point de reprise sur disque de l'application up-gs avec le cache de fichier froid

11.4 Évaluation de l'ordonnanceur global

Afin d'évaluer l'architecture d'ordonnancement global de KERRIGHED, plusieurs politiques d'ordonnancement ont été évaluées avec différentes charges applicatives. Nous n'avons pas cherché à valider une politique d'ordonnancement donnée, notre contribution étant une architecture d'ordonnanceur global pour grappe permettant de spécialiser la politique d'ordonnancement et non pas la proposition d'une politique d'ordonnancement particulière.

Pour toutes les mesures de performance du mécanisme d'ordonnancement global, la migration avec conteneurs est utilisée. De plus, le prototype actuel du système KERRIGHED ne supportant pas encore la migration de flux de données, les applications MPI se sont pas utilisées pour évaluer le mécanisme d'ordonnancement global.

11.4.1 Description des charges applicatives utilisées

11.4.1.1 Charge applicative 1 : applications synthétiques

La charge applicative 1 a été créée de toute pièce. Elle est constituée d'applications « *pthreads* » ayant un schéma d'exécution parfaitement connu. Trois applications s'exécutent : une application *pthread* (appelée *application synthétique 1*) composée de trois threads dont le premier nécessite trois fois plus de temps d'exécution que les deux autres. Les deux autres applications sont des applications séquentielles indépendantes (appelées *applications synthétiques 2 et 3*) dont le temps d'exécution est le double de celui des deux threads les plus lents de la première application. La figure 11.13 montre l'ordonnancement idéal pour ce type d'application.

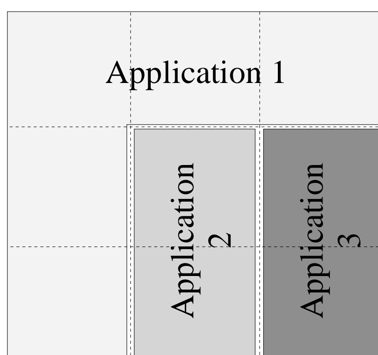

FIG. 11.13 – Schéma d'exécution optimale pour la charge applicative 1

11.4.1.2 Charge applicative 2 : applications multi-threadées

Les applications multi-threadées sont particulièrement intéressantes pour tester les politiques d'ordonnancement car ces applications sont susceptibles de faire apparaître des phénomènes de ping-pong de pages mémoire : deux variables différentes se trouvent sur

	test1c3	test7c3
Temps d'exécution en secondes	53.6	351.05

TAB. 11.3 – Temps d'exécution de reférence de *Cyrano3* pour les jeux de test *test1c3* et *test7c3*

une même page et sont accédées par deux *threads* ne s'exécutant pas sur le même nœud. La page mémoire fait alors des aller-retours entre ces deux nœuds. Cette charge applicative regroupe uniquement des applications *pthread* constituées de l'application *mgs* et de l'application synthétique multi-threadée (*application synthétique 1*).

11.4.1.3 Charge applicative 3 : applications séquentielles indépendantes

Cette charge, uniquement constituée d'applications séquentielles, regroupe trois applications *up-gs* avec une matrice de taille 1500x1500, ainsi que deux instances de l'applications EDF *Cyrano3*. La première application *Cyrano3* est exécutée avec les données *test1c3*, alors que la seconde utilise les données *test7c3*. Les temps d'exécution de ces deux instances de *Cyrano3* sont différents (voir tableau 11.3).

11.4.1.4 Charge applicative 4 : mélange d'applications séquentielles indépendantes et d'applications multi-threadées

Cette charge est composée de *Cyrano3*, *up-gs* et des applications synthétiques. Pour cette charge de travail, deux instances de *Cyrano3* (une instance avec le jeu de test *test1c3* et une autre avec le jeu de test *test7c3*), trois instances de *up-gs* et la charge applicative 1 sont lancées.

11.4.2 Description des politiques utilisées

11.4.2.1 Politique 1 : politique de placement en tourniquet (*round-robin*)

La politique 1 place chaque processus des tâches tour à tour sur chaque nœud. Cette politique ne prend pas en compte d'information sur la durée d'exécution des processus à créer. Il peut donc en résulter un déséquilibre de charge dans le cas d'une charge applicative importante : plusieurs processus ayant un temps d'exécution long sont créés sur un nœud alors que d'autres nœuds accueillent des processus ayant un temps d'exécution court. Dans ce cas, les nœuds ayant des processus avec un important temps d'exécution restent chargés même si des nœuds se trouvent sans processus à exécuter.

11.4.2.2 Politique 2 : politique de placement en tourniquet avec ordonnancement dynamique

La politique 2 comble le problème de déséquilibre potentiel dû à un placement de processus par tourniquet. Si un nœud se trouve avec une faible charge applicative par

rapport à d'autres nœuds, des processus peuvent être déplacés en cours d'exécution pour équilibrer la charge et donc améliorer globalement l'exécution des tâches dans la grappe.

L'équilibrage dynamique est fondé sur la politique présentée dans le paragraphe 9.3, seule la politique de placement a été modifiée pour effectuer du tourniquet. L'équilibrage dynamique est donc fondé sur la charge moyenne du processeur des nœuds durant la dernière minute.

11.4.3 Résultats

Pour valider les travaux de cette thèse, nous avons évalué différentes politiques d'ordonnancement en soumettant différentes charges applicatives. Pour effectuer cette évaluation, nous avons cherché à mesurer le débit applicatif, l'impact de l'utilisation de la grappe et d'un ordonnanceur global sur les applications et la réactivité du système. Cela nous a permis d'évaluer la mise en œuvre conjuguée de politiques statiques et dynamiques. On a également pu évaluer les coûts de fonctionnement de l'ordonnanceur global et des mécanismes sous-jacents de gestion globale des processus.

Le débit applicatif est obtenu en mesurant le temps nécessaire pour exécuter l'ensemble des tâches d'une charge de travail. Pour cela, les tâches sont toutes soumises en même temps et le débit applicatif correspond au temps écoulé entre le moment de soumission de la charge de travail et la terminaison de la dernière application. Pour avoir un bon débit applicatif, une politique doit exécuter le plus rapidement possible une charge applicative donnée.

L'impact de l'utilisation de la grappe et d'un ordonnanceur global sur l'exécution des applications consiste à savoir si l'exécution des applications ont été globalement pénalisé. Pour cela, nous calculons la somme des temps d'exécution de toutes les applications constituant une charge applicative (appelé *temps cumulé*) et nous la comparons à la somme des temps d'exécution de référence des applications (appelé *temps cumulé de référence*). Le *temps de référence* est le temps nécessaire pour exécuter une application ayant l'usage exclusif de la grappe.

Enfin, le débit applicatif est évalué en mesurant le rapport entre le temps nécessaire pour exécuter les applications sur la grappe par rapport au temps nécessaire pour exécuter les applications tour à tour de manière exclusive.

Pour effectuer cette évaluation, nous avons utilisé six nœuds dotés d'un processeur Pentium III cadencé à 1 GHz et de 512 Mo de mémoire centrale. Pour chaque évaluation, nous avons exécuté chaque charge applicative cinq fois pour ensuite calculer une moyenne. Ces moyennes sont données sous forme de tableaux où les chiffres en gras correspondent au débit applicatif, *i.e.* au temps nécessaire pour exécuter l'ensemble des applications de la charge soumise. L'analyse des résultats est effectuée en deux phases : (i) l'analyse de l'exécution moyenne des charges applicatives pour les différentes politiques et (ii) une comparaison des résultats obtenus pour les différentes politiques.

	Temps d'exécution de l'application 1 (en secondes)	Temps d'exécution de l'application 2 (en secondes)	Temps d'exécution de l'application 3 (en secondes)	Temps cumulé (en secondes)
Temps de référence	342.72	227.53	227.39	797.64
Politique 1	**343.09**	228.65	228.63	800.36
Politique 2	**342.72**	228.37	228.37	799.46

TAB. 11.4 – Temps d'exécution de la charge applicative 1 avec les différentes politiques d'ordonnancement

11.4.3.1 Analyse de l'exécution des charges applicatives

Pour évaluer chaque politique d'ordonnancement, nous avons exécuté les différentes charges applicatives en mesurant le temps d'exécution pour chaque tâche ainsi que le temps global de la charge applicative.

Le tableau 11.4 montre le temps d'exécution de la charge applicative 1 pour les différentes politique d'ordonnancement. La charge applicative 1 est exécutée en 343 secondes par la politique 1 et par la politique 2 (le temps nécessaire pour exécuter toute la charge applicative est le temps d'exécution de la tâche la plus longue, et apparaît en gras dans le tableau). Le placement étant efficace, la politique 2 n'a pas besoin d'effectuer un équilibrage dynamique des processus et offre donc des performances similaires à la politique 1. Ces deux politiques ont donc le même débit applicatif. En effet, la charge applicative 1, étant constitué de cinq processus, le placement par tourniquet (politique 1) permet de placer un processus par nœud et ainsi de garantir une bonne exécution des applications. La dégradation du temps d'exécution dûe au déploiement au sein de la grappe est inférieure à 1%. Cela montre également que l'utilisation de la grappe en mode partagé est intéressant par rapport à un mode exclusif où les applications s'exécutent tour à tour. Le partage de la grappe permet d'améliorer les temps d'exécution de 57%, une telle configuration de la grappe offre donc un bon débit applicatif pour cette charge applicative.

Le tableau 11.5 montre le temps d'exécution de la charge applicative 2 pour les différentes politiques d'ordonnancement. Pour la politique 1, la charge de travail est exécutée en 506 secondes alors que pour la politique 2, la charge est exécutée en 665 secondes. Pour cette charge applicative tous les nœuds de la grappe sont utilisés, *mgs* étant constitué de six threads, un thread est placé sur chaque nœud. Les deux instances de l'application 1 regroupent au total six processus. Au final, les nœuds de la grappe accueillent un processus de l'application *mgs* et un processus d'une instance de l'application 1. Les applications sont donc en concurrence sur tous les nœuds de la grappe. Cette concurrence pénalise l'exécution des applications. De plus, l'équilibrage dynamique de charge ne change pas le débit applicatif de la politique 1. En effet, lorsque *mgs* se termine, la charge de tous les nœuds diminue et seulement quatre nœuds de la grappe se trouvent avec une charge constituée

	Temps d'exécution de *mgs*	Temps d'exécution de l'application 1	Temps d'exécution de l'application 1	Temps cumulé
Temps de référence	167.22 s	343.23 s	343.23 s	853.68 s
Politique 1	185.1 s	505.64 s	**505.74** s	1196.48 s
Politique 2	182.53 s	457.39 s	**664.67** s	1304.59 s

TAB. 11.5 – Temps d'exécution de la charge applicative 2 avec les différentes politiques d'ordonnancement

	Cyrano3 (test1c3)	*Cyrano3* (test7c3)	*up-gs*	*up-gs*	*up-gs*	Temps cumulé
Temps de référence	53.6 s	351.05 s	158.8 s	158.8 s	158.8 s	881.05 s
Politique 1	56.36 s	**353.7** s	160.22 s	160.24 s	160.24 s	890.75 s
Politique 2	59.54 s	**354.36** s	160.33 s	160.39 s	160.38 s	895 s

TAB. 11.6 – Temps d'exécution (en secondes) de la charge applicative 3 avec les différentes politiques d'ordonnancement

de processus de l'application 1. Les différences de charge entre les nœuds ne sont alors pas assez forte pour qu'un équilibrage de charge soit effectué. Dans les expérimentations effectuées, la politique 2 a toujours défavorisé l'exécution de la deuxième instance de l'application1. Néanmoins, le débit applicatif des deux politiques devrait être similaire, aucun processus n'étant déplacé, des mesures supplémentaires sont nécessaires pour appuyer ce point. L'utilisation de la grappe en mode partagé est moins intéressant que pour la charge applicative 1 puisque le gain en temps d'exécution est de 41% pour la politique 1 et de 22% pour la politique 2. La grappe a donc un débit applicatif moins intéressant que pour la charge applicative 1.

Le tableau 11.6 montre le temps d'exécution de la charge applicative 3 pour les différentes politiques d'ordonnancement. Les politiques 1 et 2 permettent d'exécuter la charge de travail en 354 secondes, elles ont le même débit applicatif. En effet, la charge de travail 3 regroupe cinq processus indépendants, le déploiement en tourniquet place donc un processus par nœud. Aucune application ne se trouve donc en concurrence avec d'autre application et le débit applicatif est donc très proche d'une utilisation optimale de la grappe. Comme pour la charge de travail 1, on voit que le temps de création distante a un faible impact sur le temps d'exécution. De même, l'équilibrage dynamique n'apporte rien puisque la répartition de la charge lors du déploiement est satisfaisant. Enfin, pour ce type de charge applicative, l'utilisation de la grappe en mode partagé permet un gain de 60% en temps

	Cyrano3 (test1c3) (en s)	*Cyrano3* (test7c3) (en s)	*up-gs* (en s)	*up-gs* (en s)	*up-gs* (en s)	Appli. 1 (en s)	Appli. 2 (en s)	Appli. 3 (en s)	Temps cumulé (en s)
Temps de référence	53.6	351.05	158.8	158.8	158.8	342.72	227.53	227.39	1678.69
Politique 1	113.47	**638.03**	160.7	160.72	262.93	571.96	343.9	364.95	2616.64
Politique 2	117.68	**476.94**	162.47	258.35	275.36	457.13	282.97	357.52	2388.42

TAB. 11.7 – Temps d'exécution (en secondes) de la charge applicative 4 avec les différentes politiques d'ordonnancement

d'exécution. Le débit applicatif de la grappe est donc intéressant dans cette configuration.

Le tableau 11.7 montre le temps d'exécution de la charge applicative 4 pour les différentes politiques d'ordonnancement. La politique 1 exécute la charge applicative en 638 secondes alors que la politique 2 exécute la charge applicative en 477 secondes. La politique 2 a donc un débit applicatif meilleur pour cette charge applicative. En effet, cette charge applicative est composée de dix processus placés dans la grappe suivant la politique de tourniquet. Sur chaque nœud, des processus d'applications différentes sont donc en concurrence. De plus, les différentes applications de la charge applicative n'ont pas toutes le même temps d'exécution. Certaine applications se terminent plus rapidement que d'autres. Si l'équilibrage dynamique n'est pas disponible, la charge de certains nœuds va donc diminuer sans que les processus s'exécutant sur des nœuds chargés puissent en profiter. En revanche, avec un équilibrage dynamique de charge (politique 2), les ressources libérées par la terminaison d'applications peuvent être utilisées par des processus déplacés automatiquement par l'ordonnanceur global. Enfin, l'utilisation de la grappe est très intéressant pour cette charge applicative car elle permet un gain de 62% pour la politique 1 et de 71,5% pour la politique 2, le temps de réponse est donc intéressant.

11.4.3.2 Analyse de l'impact des différentes politiques

Le tableau 11.8 donne la somme des temps d'exécution de chaque tâche. Ce tableau permet d'évaluer les pénalités en temps pour l'exécution des charges applicatives lorsque la grappe est utilisée par rapport au *temps cumulé de référence*. Si le temps cumulé d'exécution d'une charge applicative est similaire au temps cumulé d'une exécution optimale des applications (*temps cumulé de référence*) constituant la charge, les ressources disponibles sont suffisantes et le partage des ressouces de la grappe est globalement efficace.

Pour les charges applicatives 1 et 3, l'exécution sur grappe, aussi bien avec la politique 1 qu'avec la politique 2, est efficace, le temps d'exécution individuel des applications est similaire à celui de référence. Les ressources sont donc suffisantes et leur partage est

	Charge 1	Charge 2	Charge 3	Charge 4
Temps cumulé de référence	797.64 s	853.68 s	881.05 s	1678.69 s
Politique 1	800.36 s	1196.48 s	890.75 s	2616.64 s
Politique 2	799.46 s	1304.59 s	895 s	2388.42 s

TAB. 11.8 – Temps total d'exécution (en secondes) des différentes charges applicatives suivant les politiques d'ordonnancement utilisées

	2 threads	4 threads	6 threads
Temps d'exécution (en secondes)	50.45	83.86	71.51

TAB. 11.9 – Temps d'exécution de l'application *Ecoss*

efficace. Ces résultats permettent également de voir que le coût de fonctionnement du mécanisme d'ordonnancement global est négligeable. En revanche, pour la charge applicative 2, l'exécution sur grappe pénalise l'exécution des applications de 50% pour la politique 1 et de 64% pour la politique 2. L'utilisation des ressources n'est pas suffisamment efficace. Pour la charge applicative 4, la politique 1 pénalise l'exécution des applications de 56% alors que la politique 2 ne la pénalise que de 42%. La politique 2 partage donc mieux les ressources que la politique 1.

Dans tous les cas, les évaluations ont montré que l'utilisation d'un ordonnanceur global réduit le temps d'exécution par rapport au temps cumulé des applications prises séparément. Il peut être intéressant d'utiliser les grappes peuvent donc être utilisées dans un cadre de multi-programmation pour certaine charge applicative.

11.5 Évaluation du support Posix thread

Afin de valider le support Posix thread, nous avons exécuté sur KERRIGHED des applications OPENMP compilées avec un compilateur ciblant l'interface *pthread*. Nous avons utilisé le compilateur OMNI [77] pour l'exécution de programmes OPENMP sur KERRIGHED. L'ensemble des 288 tests fournis avec le compilateur ont pu être exécutés avec succès[57].

Nous avons également exécuté l'application OPENMP *Ecoss* fournie par EDF R&D. Actuellement, le support d'OPENMP sur KERRIGHED ne vise pas la performance mais uniquement à valider le support *pthread*.

Le tableau 11.9 montre le temps d'exécution de l'application *Ecoss* en fonction du nombre de threads utilisés. Le temps d'exécution augmente en fonction du nombre de threads utilisés. En effet, plus le nombre de threads utilisés est important et plus l'application génère du faux partage des pages mémoire. D'importants phénomènes de ping-pong apparaissent donc entre les nœuds pénalisant fortement l'exécution de l'application. Cette apparition de

ping-pong est principalement dû au fait que le compilateur utilisé n'est pas optimisé pour grappe où le grain de partage est la page mémoire, mais prévu pour les machines SMP où le grain de partage est la ligne de cache. Des travaux complémentaires devront être entrepris pour l'amélioration des performances. Il serait par exemple intéressant d'étudier les travaux menés dans le cadre du projet *Performance Portability of OpenMP* dont le but est de développer des outils de programmation et un environnement de programmation OpenMP adapté aux grappes[41, 42].

11.6 Résumé

Les résultats obtenus lors de l'évaluation de notre architecture d'ordonnancement global ont confirmé que l'efficacité d'une politique d'ordonnancement dépend du type de charge applicative à exécuter, validant ainsi le concept d'ordonnanceur global adaptable permettant de configurer la politique d'ordonnancement.

Si un ordonnanceur global permet d'améliorer le débit applicatif, le temps de réponse de certaines applications peut se trouver diminué. L'utilisation d'un système de batch interfacé avec l'ordonnanceur global de Kerrighed pourrait permettre de gérer des priorités d'accès aux ressources pour les applications. Le mécanisme de point de reprise s'avèrait intéressant dans un tel cadre pour suspendre provisoirement l'exécution d'une application peu prioritaire.

Nous avons également vu que l'adaptabilité de l'ordonnanceur global n'est intéressante que parce que les coûts de fonctionnement de l'ordonnanceur global et des mécanismes de manipulation globale des processus sont faibles. Cela permet de se concentrer sur la problématique de la mise en œuvre d'une politique d'ordonnancement efficace uniquement en fonction de la nature de la charge applicative à exécuter. Le mécanisme de processus fantôme sous-jacent aux mécanismes de migration, de duplication et création/restauration de points de reprise est primordial pour les performances de ces derniers. Le coût de ce mécanisme dépend des ressources utilisées pour le stockage de l'image du processus fantôme. permet De nouveaux mécanismes peuvent donc être mis en œuvre pour mieux gérer certaines ressources. Par exemple, le coût de la migration d'un processus n'utilisant pas les conteneurs est important (quelque soit la méthode de transfert utilisée), car les accès au réseau créent un important coût. La mis en œuvre d'un mécanisme de migration spécifique à des réseaux hautes performances tels que Myrinet[18] peuvent donc être mis en œuvre, améliorant ainsi directement l'efficacité de l'ordonnanceur global.

12 CONCLUSION

12.1 Bilan

Actuellement, les grappes de PCs sont de plus en plus utilisées comme support d'exécution d'applications scientifiques même si la programmation et l'utilisation des grappes est rendue complexe en raison de la distribution des ressources sur les nœuds. Les systèmes à image unique visent à offrir une gestion globale des ressources de manière transparente pour l'utilisateur et le programmeur De nombreux systèmes tentent d'offrir un système à image unique mais beaucoup de ces projets ne remplissent que partiellement les objectifs (voir tableau 12.1).

L'ordonnanceur joue un rôle très important au sein d'un système à image unique : il est chargé de répartir au mieux les applications exécutées sur une grappe pour tirer le meilleur profit des ressources disponibles. Pour cela, la politique mise en œuvre par l'ordonnanceur doit être adaptée aux besoins des applications (de très nombreuses politiques d'ordonnancement pour différents types d'applications sont disponibles dans la littérature), mais également à l'état courant de la grappe, rendant les ordonnanceurs adaptables particulièrement intéressants. Un ordonnanceur global pour grappe suppose l'existence de mécanismes de gestion globale des processus efficaces et capables de gérer une large gamme de processus. La littérature est important sur ces mécanismes mais aujourd'hui il existe encore des limitations sur le type de processus qui peut être déployé ou sur l'efficacité. Dans le cadre de la conception du système d'exploitation pour grappe KERRIGHED, nous avons proposé une architecture modulaire pour la conception et la mise en œuvre d'ordonnanceurs globaux adaptables pour grappe. Cette architecture fournit les primitives de base et les outils nécessaires à la conception de nouvelles politiques d'ordonnancement. Grâce à une gestion dynamique des différents composants de l'ordonnanceur, il est possible de mettre en place des politiques s'auto-adaptant à l'état de la grappe. Ces travaux ont également abouti au développement d'un outil graphique de création de politiques d'ordonnancement, permettant de générer automatiquement les prototypes de chacun des composants nécessaires au développeur pour la mise en place d'une nouvelle politique. La difficulté de programmation est alors en grande partie masquée au développeur de nouvelles politiques. En outre, cette architecture modulaire est associée à un mécanisme de configuration dynamique de l'ordonnanceur global permettant de charger et de décharger à volonté les différents composants constituant une politique d'ordonnancement, sans interruption des services système ni des applications en cours d'exécution en sein de la grappe.

Pour pouvoir ordonnancer tout type d'application, l'ordonnanceur global doit disposer

	Kerrighed	OpenSSI[92]	OpenMosix[11]	Genesis[40]
Gestion globale des fichiers	partielle	oui	oui	oui
Gestion globale de la mémoire	oui	non	non	oui
Gestion globale des flux de données	oui	partielle	partielle	oui
Migration de processus	oui	oui	oui	oui
Migration de L'espace d'adressage à la demande	oui	non	non	?
Dépendances résiduelles après migration	non	non	oui	non
Création de points de reprise	oui	non ?	non	oui
Ordonnancement global	oui	oui	oui	oui
Ordonnancement adaptable	oui	non	non	?
Interface Unix	oui	oui	oui	oui
OpenSource	oui	oui	oui	?
Type de noyau	Linux	Linux	Linux	Micro-noyau spécifique
Gestion de groupe de processus	partielle	oui	non	oui

TAB. 12.1 – Comparaison des caractéristiques de différents système d'exploitation pour grappe

des mécanismes de manipulation globale de processus efficaces et capables de gérer aussi bien des applications séquentielles que des applications parallèles. Pour cela, nous avons proposé un mécanisme commun pour tous les mécanismes de gestion globale des processus : les processus fantôme. Un processus fantôme permet de manipuler un processus indépendamment de sa localisation. Il est alors possible, par extension de ce mécanisme, de mettre en place des mécanismes efficaces de gestion globale de processus que sont la création distante, la migration sans dépendance résiduelle et la création/restauration de point de reprise de processus. Les mécanismes conçus dans le cadre de cette thèse ont été étudiés pour pouvoir tirer profit des autres mécanismes de fédération des ressources tels que par exemple la gestion globale de la ressource mémoire. Il est ainsi possible pour l'ordonnanceur de manipuler des *threads* de manière efficace. L'ordonnanceur global n'est donc pas limité par le type d'applications pouvant être manipulées, contrairement à beaucoup de systèmes actuels comme OpenMosix.

L'ensemble des mécanismes de gestion globale de processus que nous avons conçus a également permis d'offrir à l'échelle de la grappe des interfaces de programmation standard comme les *threads* POSIX. Ce travail a également permis d'offrir un premier support d'exécution d'applications OPENMP, en utilisant un compilateur transformant un code OPENMP en code *pthread* pouvant être exécuté sur KERRIGHED. A notre connaissance, aucun système pour grappe n'offre de fonctionnalités aussi complètes en matière de gestion de mémoire et de threads.

Le support de ces interfaces de programmation standard permet de garantir un accès transparent aux mécanismes de gestion globale des ressources.

L'ensemble de ces travaux a permis de mettre en œuvre un prototype GPL opérationnel et distribué sur le site *http ://www.kerrighed.org*. Le prototype dispose des mécanismes de gestion globale de processus que sont la migration de processus, la duplication de processus, la création distante de processus et enfin la création de points de reprise pour processus séquentiels. Il dispose également de l'ordonnanceur global adaptable présenté dans ce document, accompagné de quelques sondes processeurs, des analyseurs locaux et des politiques pouvant les exploiter. La mise en œuvre de tous ces mécanismes a également permis de développer quelques outils logiciels permettant de tirer profit de la gestion globale de la ressource processeur. Par exemple, un environnement de programmation simple de nouvelles politiques d'ordonnancement a été mis en œuvre, de même que des scripts permettant d'utiliser manuellement en ligne de commande les différents mécanismes de manipulation des processus. Une interface de programmation propre à KERRIGHED a également été développée en complément du support *pthread* pour tirer pleinement profit des fonctionnalités de gestion globale des processus.

Ce prototype a été validé d'une part par l'évaluation des performances des mécanismes de gestion globale des processus et d'autre part par la mise en œuvre de différentes politiques d'ordonnancement et leur évaluation en exécutant quelques charges de travail. Pour le premier aspect, l'évaluation de performance a montré l'efficacité de l'ensemble des mécanismes de gestion globale des processus fondés sur le concept de processus fantôme. Pour le second aspect, les évaluations montrent l'intérêt de l'architecture modulaire et con-

figurable. Les applications industrielles fournies par EDF R&D incluses dans les charges de travail ou utilisées pour valider le support POSIX thread montrent la robustesse du prototype.

Les travaux présentés dans ce document sont une contribution directe à la création d'un système à image unique pour grappe. Une partie du projet industriel COCA, action du PEA ARCOSIUM regroupant l'ONERA, la DGA et l'INRIA, vise à offrir une infrastructure haute performance dédiée aux simulations scientifiques et techniques distribuées, fondée le prototype du système KERRIGHED incluant mes travaux de thèse.

12.2 Perspectives

Les travaux présentés dans ce document ont deux conséquences directes : la diffusion du prototype du système KERRIGHED à plus grande échelle et l'utilisation des mécanismes développés durant la thèse pour répondre aux besoins des utilisateurs.

Pour améliorer la diffusion de KERRIGHED, il est prévu d'intégrer le système KERRIGHED à OSCAR, une suite de logiciels et d'outils pour utiliser efficacement une grappe de calculateurs. Pour le moment, OSCAR regroupe principalement des composants permettant l'administration simple des grappes (*e.g.* installation de nœuds), la haute disponibilité et des bibliothèques de programmation telles que MPI. L'un des futurs objectifs d'OSCAR est de proposer un composant SSI fondé sur KERRIGHED. Cette intégration sera effectuée dans le cadre de mon post-doctorat industriel co-financé par EDF R&D et qui se déroulera en partie au sein de l'équipe de développement d'OSCAR à l'*Oak Ridge National Laboratory* (ORNL) aux États-Unis.

Un second axe de travail suite aux travaux de cette thèse est de compléter le prototype mis en œuvre et d'en tirer profit pour d'autres travaux. Un stage de DEA a été initié pour l'ordonnancement d'applications parallèles communiquant par partage de données. En effet, l'exécution de ces applications sur grappe peut être pénalisé par du faux partage qui crée un phénomène de *ping-pong* de pages mémoire entre les nœuds de la grappe. Ce phénomène peut être minimisé en rapprochant les threads générant un faux partage. Pour cela, une politique d'ordonnancement est en cours d'étude et sera mise en œuvre dans KERRIGHED. Ces travaux de stages sont fondés sur des travaux préliminaires effectués durant le stage de DEA de Louis Rilling[75]. Un second stage de DEA a été initié par la mise en œuvre d'un système de batch fondé sur les mécanismes de gestion globale de processus de KERRIGHED. Ce système de batch permettrait d'offrir une interface aux utilisateurs identiques aux systèmes traditionniellement utilisés sur les grappes industrielles.

Le mécanisme d'ordonnancement global de KERRIGHED ne permet pas pour le moment de mettre en œuvre des politiques de co-scheduling. Ce type d'ordonnanceur nécessite de pouvoir interagir avec l'ordonnanceur local des nœuds de la grappe. Pour le moment, l'ordonnanceur global ne dispose pas des mécanismes de primitives permettant l'interaction avec l'ordonnanceur local. L'environnement de développement de nouveaux ordonnanceur globaux doit donc être étendu pour offrir des primitives permettant le partage temporel

des ressources. De plus, l'architecture modulaire du mécanisme d'ordonnancement global de KERRIGHED doit permettre la mise en œuvre de telles politiques d'ordonnancement.

La version actuelle de KERRIGHED n'offre qu'une gestion globale des ressources de type bloc et des ressources processeur. Les mécanismes de gestion globale de la ressource processeur ont été conçus pour tirer profit de la fédération de toutes les ressources disponibles au sein de la grappe. Nous avons montré comment les *conteneurs* permettant de fédérer les ressources de type bloc. De même les mécanismes de fédération des flux de données (*e.g.* « *sockets* », « *pipe* », signaux), ou des disques (à travers un système de fichier distribué ou parallèle). La fédération des flux de données et des disques sont en cours d'étude en sein de l'activité de recherche KERRIGHED du projet PARIS de l'IRISA, et restent à intégrer aux mécanismes de gestion globale des processus.

Lorsque cette intégration sera effectuée, il sera alors possible de mettre en œuvre au sein de KERRIGHED des politiques d'ordonnancement plus complexes et plus variées. Il sera donc possible de mettre en œuvre les politiques d'ordonnancement décrites dans la littérature et de les évaluer avec des applications industrielles. KERRIGHED pourra donc servir de plate-forme réaliste d'expérimentation et d'évaluation de politiques d'ordonnancement.

Nous avons également vu que les travaux de gestion globale de processus présentés dans ce document ont permis d'exécuter des simulations numériques OPENMP sur KERRIGHED. Ce premier support d'exécution OPENMP n'offre pas de bonnes performances, mais des travaux d'optimisation peuvent être menés, notamment sur une optimisation des synchronisations au sein des applications OPENMP, et de l'allocation distribuée de la mémoire. Des travaux de recherche sur la conception d'un compilateur OPENMP pour grappe, *i.e.* un compilateur prenant en compte le grain de partage qui est la page et non la ligne de cache, pourraient être menés dans une grappe pour permettre au système KERRIGHED d'offrir de meilleures performances pour l'exécution d'applications OPENMP.

Durant cette thèse, des travaux sur la création de points de reprise pour applications parallèles utilisant la mémoire partagée ont menés conjointement avec R. Badrinath, enseignant-chercheur à l'IIT de Kharagpur en Inde. Ces travaux ont permis d'isoler un ensemble de mécanismes communs pour la mise en œuvre de différents protocoles de création de points de reprise. Un prototype permettant la création de points reprise pour applications à mémoire partagée a été mis en œuvre, mais le protocole de reprise n'a pas pu être finalisé, faute de temps. De prochains travaux pourraient donc finaliser cette procédure de reprise. Une étude sur l'extension de ce protocole aux applications parallèles communiquant par échange de messages pourra également être menée. Avec ces différents protocoles, il serait alors possible d'arrêter et/ou de reprendre l'exécution d'une application, quelle soit parallèle ou séquentielle. Le système disposerait donc de mécanismes particulièrement intéressants pour offrir des services de tolérance aux fautes et de haute disponibilité, les applications pouvant être déplacées, stoppées ou reprises suite à une erreur ou à un arrêt de nœud volontaire. De plus, KERRIGHED pourrait alors servir de plate-forme de comparaison de protocoles de sauvegarde de point de reprise, comparaison qui pourrait être menée avec des applications industrielles. Jusqu'à présent, une telle comparaison n'a jamais été effectuée autrement que par simulation.

De plus en plus, les institutions et les grandes entreprises disposent de plusieurs grappes réparties géographiquement, reliées par des réseaux longue distance. Sur ce type d'architecture, des liens réseau et des nœuds dans les grappes peuvent subir des défaillances. De plus, des grappes se joignent ou quittent la fédération à tout momnent, De nouvelles problématiques apparaissent donc pour tirer profit des ressources disponibles sur l'ensemble de ces grappes. Une étude sur l'extension des mécanismes présentés dans ce document aux grappes fédérées pourrait être menée. Des mécanismes permettant de déplacer les applications sont donc particulièrement intéressants pour le cas de retraits programmés de nœuds, alors que les mécanismes de gestion de points de reprise sont utiles pour la reprise de l'exécution d'une application dans le cas du retrait non programmé d'un nœud. L'extension des mécanismes de gestion globale des processus à des architectures de plus grande taille, comme les grappes fédérées, est donc intéressant. En particulier, il serait pertinent d'étudier l'extension de l'architecture modulaire de l'ordonnanceur à des architectures de grande taille et dynamiques, reliées entre elles par des réseaux de communication de différente nature et performance. L'approche modulaire et la manipulation de groupes de processus et de groupes de nœuds devraient permettre de mettre en place des stratégies intéressantes pour grappes de grappes. De plus, l'extension du mécanisme de configuration dynamique de l'ordonnanceur est particulièrement intéressant pour configurer de telles architectures, permettant de modifier la configuration de sous-ensembles (*i.e.* grappes ou ensembles de nœuds) sans interruption des services système, ni de l'exécution des applications.

BIBLIOGRAPHIE

[1] Accetta (Mike), Baron (Robert), Bolosky (William), Golub (David), Rashid (Richard), Tevanian (Avadis) et Young (Michael). – Mach : A new kernel foundation for Unix development. *In : USENIX Conference Proceedings*, pp. 93–112. – 1986.

[2] Amnon (Lior Amar). – The MOSIX scalable cluster file systems for Linux.

[3] Anderson (T.), Culler (D.) et Patterson (D.). – A case for now (networks of workstations). *IEEE Micro*, vol. 15, 1995, pp. 54–64.

[4] Antoniu (Gabriel) et Bougé (Luc). – DSM-PM2 : A portable implementation platform for multithreaded DSM consistency protocols. *In : 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01).* Held in conjunction with IPDPS 2001. IEEE TCPP, pp. 55–70. – San Francisco, avril 2001.

[5] Antoniu (Gabriel), Bougé (Luc) et Namyst (Raymond). – An efficient and transparent thread migration scheme in the PM2 runtime system. *In : Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99).* Held in conjunction with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, pp. 496–510. – San Juan, Puerto Rico, avril 1999.

[6] Apache Software Foundation. – *Xalan-C++ Overview.* http ://xml.apache.org/xalan-c/overview.html.

[7] Artsy (Yeshayahu) et Finkel (Raphael A.). – Designing a process migration facility : The Charlotte experience. *IEEE Computer*, vol. 22, n° 9, 1989, pp. 47–56.

[8] Badrinath (Ramamurthy) et Morin (Christine). – Common mechanisms for supporting fault tolerance in DSM and message passing systems. *In : Concurrent Information Processing and Computing*, éd. par Grigoras (Dan), Nicalau (Alex) et Tiplea (Ferucio Laurentiu). NATO Advanced Research Workshop, pp. 37–45. – Alexandru loan Cuza, University Press, juillet 2003.

[9] Badrinath (Ramamurthy) et Morin (Christine). – *Locks and Barriers in Checkpointing and Recovery.* – Research Report n° RR-5021, IRISA, Rennes, France, INRIA, octobre 2003.

[10] Badrinath (Ramamurthy), Morin (Christine) et Vallée (Geoffroy). – Checkpointing and recovery of shared memory parallel application in a cluster. *In : Proc. of the workshop on Distributed Shared Memory (DSM2003) in CCGRID 2003.* pp. 471–477. – Tokyo, Japan, mai 2003.

[11] Bar (Moshe) et Krushna (Maya Anu Asmita Snehal). – Introduction to openmosix, 2003.

[12] Barak (Amnon) et Braverman (Avner). – Memory Ushering in a Scalable Computing Cluster. *Journal of Microprocessors and Microsystems*, vol. 22, n° 3–4, août 1998, pp. 175–182.

[13] Barak (Amnon) et La'adan (Oren). – The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, vol. 13, n° 4–5, 1998, pp. 361–372.

[14] Barreto (L.), Douence (R.), Muller (G.) et Sudholt (M.). – Programming os schedulers with domain-specific languages and aspects : New approaches for OS kernel engineering, 2002.

[15] Basney (Jim) et Livny (Miron). – Deploying a high throughput computing cluster. *In : High Performance Cluster Computing : Architectures and Systems, Volume 1*, éd. par Buyya (Rajkumar). – Prentice Hall PTR, 1999.

[16] Berthou (Jean-Yves) et Fayolle (Eric). – Comparing OpenMP, HPF and MPI programming, a study case. *The International Journal of High Performance Computing Applications*, August 2001.

[17] Bode (Brett), Halstead (David M.), Kendall (Ricky) et Lei (Zhou). – The portable batch scheduler and the MAUI scheduler on Linux clusters. *In : 4th Annual Linux Showcase and Conference.* – octobre 2000.

[18] Boden (Nanette J.) et al. – Myrinet : A gigabit-per-second local area network. *In : IEEE Micro*, pp. 29–39. – février 1995.

[19] Burlett (Jean-Yves). – *Support OpenMP sur un système à image unique.* – Rapport de stage de DEA, IFSIC, Université de Rennes 1, France, juin 2002.

[20] Buyya (Rajkumar). – *High Performance Cluster Computing : Architectures and Systems.* – Paperback, juin 1999.

[21] Buyya (Rajkumar), Cortes (Toni) et Jin (Hai). – Single system image (SSI). *The International Journal of High Performance Computing Applications*, vol. 15, n° 2, 2001, pp. 124–135.

[22] Calinescu (Gruia), Karloff (Howard J.) et Rabani (Yuval). – An improved approximation algorithm for multiway cut. *In : ACM Symposium on Theory of Computing*, pp. 48–52. – 1998.

[23] Campbell (Roy H.), Islam (Nayeem), Raila (David) et Madany (Peter). – Designing and implementing choices : an object-oriented system in c++. *Commun. ACM*, vol. 36, n° 9, 1993, pp. 117–126.

[24] Casavant (T.L.) et Kuhl (J.G.). – A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, vol. 14, n° 2, février 1988, pp. 141–154.

[25] Chandy (K.M.) et Lamport (L.). – Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Computer Systems*, vol. 3, n° 1, février 1985, pp. 63–75.

[26] Dagum (Leonardo). – Openmp : A proposed industry standard api for shared memory programming, octobre 1997.

[27] Dahlhaus (Elias), Johnson (David S.), Papadimitriou (Christos H.), Seymour (P. D.) et Yannakakis (Mihalis). – The complexity of multiterminal cuts. *SIAM J. Comput.*, vol. 23, n° 4, 1994, pp. 864–894.

[28] Daniel P. Bovet (Marco Cesati). – *Understanding the Linux Kernel, 2nd Edition.* – o'reilly, décembre 2002.

[29] Denneulin (Yves). – Clic, cluster linux pour le calcul. – http ://clic.mandrakesoft.com.

[30] des Ligneris (Benoît), Scott (Stephen), Naughton (Thomas) et Gorsuch (Neil). – Open source cluster application resources (oscar) : Design, implementation and interest for the computer scientific community. – First OSCAR Symposium, mai 2003.

[31] Douglis (F.). – *Transparent Process Migration in the Sprite Operating System.* – Thèse de PhD, Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720, septembre 1990.

[32] Eager (D. L.), Lazowska (E. D.) et Zahorjan (J.). – The limited performance benefits of migrating active processes for load sharing. *In : ACM Sigmetrics Conference on Measuring and Modeling of Computer Systems*, pp. 662–675. – mai 1988.

[33] Elnozahy (M.), Alvisi (L.), Wang (Y-M.) et Johnson (D.B.). – *A Survey of Rollback-Recovery Protocols in Message-Passing Systems.* – Rapport technique n° CMU-CS-99-148, Carnegie Mellon University, juin 1999.

[34] Feitelson, G. (Dror) et Rudolph (Larry). – Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, vol. 23, n° 5, mai 1990, pp. 65–77.

[35] Feitelson (D.). – Job Scheduling in Multiprogrammed Parallel Systems, août 1997. http ://www.cs.huji.ac.il/ feit/papers/survey.ps.gz.

[36] Folliot (Bertil) et Sens (Pierre). – GATOSTAR : A fault tolerant load sharing facility for parallel applications. *In : European Dependable Computing Conference*, pp. 581–598. – 1994.

[37] Frachtenberg (Eitan), Feitelson (Dror G.), Petrini (Fabrizio) et Fernandez (Juan). – Flexible coscheduling : Mitigating load imbalance and improving utilization of heterogeneous resources. *In : ipdps.* – avril 2003.

[38] Friedman (Roy), Goldin (Maxim), Itzkovitz (Ayal) et Schuster (Assaf). – MILLIPEDE : Easy parallel programming in available distributed environments. *Software Practice and Experience*, vol. 27, n° 8, 1997, pp. 929–965.

[39] Gallard (Pascal) et Morin (Christine). – Dynamic streams for efficient communications between migrating processes in a cluster. *Parallel Processing Letters*, vol. 13, n° 4, décembre 2003. – to appear.

[40] Goscinski (A.), Hobbs (M.) et Silcock (J.). – Genesis : an efficient, transparent and easy to use cluster operating system. *Parallel Comput.*, vol. 28, n° 4, 2002, pp. 557–606.

[41] Hadjidoukas (Panagiotis E.), Polychronopoulos (Eleftherios D.) et Papatheodorou (Theodore S.). – OpenMP runtime support for clusters of multiprocessors. *In : International Workshop on OpenMP Applications and Tools (WOMPAT 2003)*. – Toronto, Canada, juin 2003.

[42] Hadjidoukas (Panagiotis E.), Polychronopoulos (Eleftherios D.) et Papatheodorou (Theodore S.). – A modular OpenMP implementation for clusters of multiprocessors. *Parallel and Distributed Computing Practices (PDCP)*, To appear.

[43] Henderson et Robert (L.). – Job scheduling under the portable batch system. *In : Job Scheduling Strategies for Parallel Processing*, éd. par Feitelson (Dror G.) et Rudolph (Larry). pp. 279–294. – Springer-Verlag, 1995. Lecture Notes in Computer Science vol. 949.

[44] Hendriks (Erik). – BProc : the Beowulf distributed process space. *In : Institute for Crustal Studies (ICS) 2002*, pp. 129–136. – New York City, USA, juin 2002.

[45] IEEE. – *IEEE P1003.4a/D4 draft standard, Threads Extension for Portable Operating Systems*, AUG 1990, 69–70p.

[46] InfiniBand Trade Association. – *InfiniBand Specification 1.0a*, juin 2001.

[47] International Organization of Standardization. – *Standard Generalized Markup Language (SGML)*, August 2001.

[48] Keleher (P.), Dwarkadas (S.), Cox (A. L.) et Zwaenepoel (W.). – Treadmarks : Distributed shared memory on standard workstations and operating systems. *In : Proc. of the Winter 1994 USENIX Conference*, pp. 115–131. – 1994.

[49] Kramer (W. T. C.) et Craw (J. M.). – Effective Use of Cray Supercomputers. *In : Proceedings of the Supercomputing 89*. pp. 721–731. – New York, NY, 1989.

[50] Kunz (T.). – The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, vol. 17, n° 7, 1991, pp. 725–730.

[51] Lampson (Butler W.). – Atomic transactions. *In : Distributed Systems - Architecture and Implementation, An Advanced Course*. pp. 246–265. – Springer-Verlag, 1981.

[52] Lawall (Julia L.), Muller (Gilles) et Barreto (Luciano Porto). – Capturing os expertise in a modular type system : The bossa experience. *In : ACM SIGOPS European Workshop*. – Saint-Emillion, France, septembre 2002.

[53] Litzkow (M.), Tannenbaum (T.), Basney (J.) et Livny (M.). – *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*. – Technical Report n° 1346, Computer Sciences Department, University of Wisconsin, avril 1997.

[54] Lottiaux (Renaud). – *Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système GOBELINS*. – Thèse de PhD, Université de Rennes 1, IFSIC, France, décembre 2001.

[55] Lu (Chin), Lui (John C.S.), Lie (Peter W.K.), Tang (M.K.), Lau (S.Y.) et Li (H.K.). – Distributed scheduling framework - a load distribution facility on mach.

[56] Luethke (Brian), Scott (Stephen) et Naughton (Thomas). – Oscar cluster administration with c3. – First OSCAR Symposium, mai 2003.

[57] Margery (David), Vallée (Geoffroy), Lottiaux (Renaud), Morin (Christine) et Berthou (Jean-Yves). – Kerrighed : a SSI cluster OS running OpenMP. *In : Proc. 5th European Workshop on OpenMP (EWOMP '03).* – septembre 2003.

[58] Martin (Cyrille) et Richard (Olivier). – Parallel launcher for clusters of pcs. *In : Parco.* – 2001.

[59] Milojicic (Dejan S.), Douglis (Fred), Paindaveine (Yves), Wheeler (Richard) et Zhou (Songnian). – Process migration. *ACM Computing Surveys (CSUR)*, vol. 32, n° 3, 2000, pp. 241–299.

[60] Morin (C.) et Puaut (I.). – A survey of recoverable distributed shared memory systems. *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, n° 9, 1997, pp. 959–969.

[61] Morin (C.) et Puaut (I.). – A survey of recoverable distributed shared memory systems. *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, n° 9, 1997, pp. 959–969.

[62] Morin (Christine), Gallard (Pascal), Lottiaux (Renaud) et Vallée (Geoffroy). – Towards an efficient single system image cluster operating system. *In : Proc. of Intl. Conf. on Architecture and Algorithms for Parallel Processing (ICA3PP 2002).* pp. 370–377. – Beijing, China, octobre 2002.

[63] Morin (Christine), Gallard (Pascal), Lottiaux (Renaud) et Vallée (Geoffroy). – Towards an efficient Single System Image cluster operating system. *Future Generation Computer Systems*, vol. 20, n° 2, janvier 2004.

[64] Morin (Christine), Lottiaux (Renaud), Vallée (Geoffroy), Gallard (Pascal), Utard (Gaël), Badrinath (Ramamurthy) et Rilling (Louis). – Kerrighed : a single system image cluster operating system for high performance computing. *In : Proc. of Europar 2003 : Parallel Processing.* pp. 1291–1294. – Springer Verlag, août 2003.

[65] Mueller (F.). – On the design and implementation of DSM-threads. *In : Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 315–324. – juin 1997.

[66] Nlong (Jean-Michel) et Denneulin (Yves). – Migration de processus linux sous i-cluster. *In : 15èmes rencontres francophones en Parallèlisme*, pp. 614–623. – octobre 2003.

[67] Osman (Steven), Subhraveti (Dinesh), Su (Gong) et Nieh (Jason). – The design and implementation of zap : A system for migrating computing environments. *In : The Fifth Symposium on Operating Systems Design and Implementation.* Boston, MA. – décembre 2002.

[68] Ousterhout (John K.). – Scheduling Techniques for Concurrent Systems. *In : Third International Conference on Distributed Computing Systems*, pp. 22–30. – mai 1982.

[69] Pfister (Gregory). – *In Search of Clusters (2nd Edition).* – Paperback, décembre 1997.

[70] Pinheiro (E.) et Bianchini (R.). – Nomad : A scalable operating system for clusters of uni and multiprocessors. *In : 1st IEEE International Workshopon Cluster Computing.* – décembre 1999.

[71] Pinheiro (Eduardo). – Truly-transparent checkpointing of parallel applications.

[72] Powell (Michael L.) et Miller (Barton P.). – Process migration in demos/mp. *In : Proceedings of the ninth ACM symposium on Operating systems principles.* pp. 110–119. – ACM Press, 1983.

[73] Randell (B.). – System structure for software fault tolerance. *In : Proceedings of the international conference on Reliable software*, pp. 437–449. – 1975.

[74] Richard (Bruno) et Augerat (Philippe). – *I-Cluster : Intense Computing with Untapped Resources.* – Rapport technique nº HPL-2002-81, HP Labs, 2002.

[75] Rilling (Louis). – *Ordonnancement global de processus dans les grappes de calculateurs, mise en oeuvre dans le système Gobelins.* – Rapport de stage de DEA, IFSIC, Université de Rennes 1, France, juin 2002.

[76] Sandberg (R.), Goldberg (D.), Kleiman (S.), Walsh (D.) et Lyon (B.). – Design and implementation of the sun network filesystem. *In : Proceedings of the Summer.* – 1985.

[77] Sato (M.), Satoh (S.), Kusano (K.) et Tanaka (Y.). – Design of openmp compiler for an smp cluster, 1999.

[78] Schuster (Assaf) et Shalev (Lea). – Using remote access histories for thread scheduling in distributed shared memory systems. *In : Proc. of the 12th Int'l. Symp. on Distributed Computing (DISC98).* – 1998.

[79] Seifert (Rich). – *Gigabit Ethernet : Technology and Applications for High Speed LANs.* – Addison-Wesley, mai 1998.

[80] Shivaratri (Niranjan G.), Krueger (Phillip) et Singhal (Mukesh). – Load distributing in locally distributed systems. *IEEE Computer*, vol. 25, nº 12, décembre 1992, pp. 33–44.

[81] Steketee (Chris). – Process migration and load balancing in amoeba.

[82] Steketee (Chris), Zhu (Weiping) et Moseley (Philip). – Implementation of process migration in amoeba. *In : International Conference on Distributed Computing Systems*, pp. 194–201. – 1994.

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

[84] Sunderam (V. S.). – PVM : A framework for parallel distributed computing concurrency. *In : Practice and Experience*, pp. 315–339. – décembre 1990.

[85] Takahash (T.), O'Carroll (F.), Tezuka (H.), Hori (A.), Sumimoto (S.), Harada (H.), Ishikawa (Y.) et Beckman (P.H.). – Implementation and evaluation of MPI on an

SMP cluster. *In : Parallel and Distributed Processing. IPPS/SPDP'99 Workshops.* – avril 1999.

[86] Theimer (Marvin M.), Lantz (Keith A.) et Cheriton (David R.). – Preemptable remote execution facilities for the v-system. *In : Proceedings of the tenth ACM symposium on Operating systems principles.* pp. 2–12. – ACM Press, 1985.

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

[88] Vallée (Geoffroy). – Un ordonnanceur de processus pour grappe adaptable : mise en oeuvre dans le système Kerrighed. *In : Actes des Rencontres francophones du parallélisme (RenPar 15).* – La Colle sur Loup, France, octobre 2003.

[89] Vallée (Geoffroy), Morin (Christine), Berthou (Jean-Yves), Malen (Yvan Dutka) et Lottiaux (Renaud). – Process migration based on gobelins distributed shared memory. *In : DSM 2002 : Distributed Shared Memory on Clusters.* Held in conjunction with CCGRID 2002, IEEE/ACM, pp. 325–330. – Berlin, Germany, mai 2002.

[90] Vallée (Geoffroy), Morin (Christine), Berthou (Jean-Yves) et Rilling (Louis). – A new approach to configurable dynamic scheduling in clusters based on single system image technologies. *In : Industrial Track of the International Parallel an Distributed Processing Symposium.* – Nice, France, avril 2003.

[91] Walker (Bruce), Popek (Gerald), English (Robert), Kline (Charles) et Thiel (Greg). – The locus distributed operating system. *In : Proceedings of the ninth ACM symposium on Operating systems principles.* pp. 49–70. – ACM Press, 1983.

[92] Walker (Bruce J.). – *Open Single System Image (openSSI) Linux Cluster Project.* – Rapport technique, Hewlett-Packard, 2000.

[93] World Wide Web Consortium. – *XSL Transformations (XSLT) Version 1.0*, novembre 1999.

[94] World Wide Web Consortium. – *Extensible Markup Language (XML) 1.0 (Second Edition)*, octobre 2000.

[95] World Wide Web Consortium. – *Extensible Stylesheet Language Family (XSL)*, octobre 2001.

[96] Zandy (Victor C.), Miller (Barton P.) et Livny (Miron). – Process hijacking. *In : The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC8).* Redondo Beach, California, pp. 177–184. – août 1999.

[97] Zayas (E.). – *The use of copy-on-reference in a process migration system.* – Thèse de PhD, University of Carnegie - Mellon, avril 1987.

[98] Zhong (Hua) et Nieh (Jason). – *CRAK : Linux Checkpoint/Restart As a Kernel Module.* – Rapport technique n° CUCS-014-01, Department of Computer Science, Columbia University, novembre 2001.

[99] Zhu (Weiping), Socko (Piotr) et Kiepuszewski (Bartek). – Migration impact on load balancing - an experience on amoeba. *ACM SIGOPS Operating Systems Review*, vol. 31, n° 1, 1997, pp. 43–53.

Annexes

A FICHIERS XSL-T POUR LA GÉNÉRATION DES CHARGEURS DES COMPOSANTS D'UN ORDONNANCEUR GLOBAL

A.1 Feuille XSL-T générant fichier scheduler_loader.c

La génération du fichier *scheduler_loader.c* s'effectue en deux phase. Ces deux phases permettent de gérer les tables locales permettant de stocker les informations de nœuds de la grappe remontées par les AL ou envoyées par d'autres GOrG. Le listing A.1 montre le code de la première feuille de style.

Listing A.1 – Premier fichier XSL permettant de générer le fichier scheduler_loader.c

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">
#define MODULE
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/module.h
<xsl:text disable-output-escaping="yes">&gt;</xsl:text>
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/sched.h
<xsl:text disable-output-escaping="yes">&gt;</xsl:text>
#include "gtypes.h"
#include "request_queue_types.h"
#include "scheduler.h"

<xsl:for-each select="schedulers">
<xsl:for-each select="scheduler">
#include "<xsl:value-of select="file"/>.h"
<xsl:apply-templates/>
</xsl:for-each>
</xsl:for-each>

MODULE_AUTHOR("Geoffroy Ã©Valle");
MODULE_DESCRIPTION("Scheduler Loader");
MODULE_LICENSE("GPL");

int init_module (void)
{
<xsl:apply-templates/>
```

```
  rwlock_init (&amp;scheduler_lock);
  printk ("Scheduler initialized\n");
  return 0;
}

void cleanup_module (void)
{
  remove_schedulers ();
}
</xsl:template>


<xsl:template match="scheduler">
</xsl:template>

<xsl:template match="nodes"/>
<xsl:template match="name"/>
<xsl:template match="file"/>
<xsl:template match="node_election"/>
<xsl:template match="process_election"/>
<xsl:template match="diffusion_function"/>
<xsl:template match="on_node_info_update"/>
<xsl:template match="on_local_info_update"/>
<xsl:template match="process_placement"/>

<xsl:template match="analyzers">
<xsl:for-each select="analyzer">
<xsl:for-each select="global_infos">
<!--
/* WARNING, the static cluster size is here 64 nodes */
extern global_table_t <xsl:value-of select="."/>[64];
-->
</xsl:for-each>
</xsl:for-each>
</xsl:template>

<xsl:template match="scheduler">
        scheduler_t *scheduler;

  printk ("Initializing scheduler...\n");
        scheduler = create_scheduler (<xsl:text disable-output-escaping="yes">&amp;
                </xsl:text><xsl:value-of select="file"/>,
                "<xsl:value-of select="name"/>");
<xsl:for-each select="analyzers">
<xsl:for-each select="analyzer">
<xsl:for-each select="name">
        add_analyzer (scheduler, "<xsl:value-of select="."/>");
</xsl:for-each>
</xsl:for-each>
<xsl:for-each select="analyzer/global_info"/>
  init_global_table (scheduler, "<xsl:value-of select="analyzer/name"/>");
```

```
</xsl:for-each>
<xsl:for-each select="node_election">
  assign_function_to_find_a_node (scheduler,
        <xsl:text disable-output-escaping="yes">
        &amp;</xsl:text><xsl:value-of select="."/>);
</xsl:for-each>
  assign_function_to_find_a_process (scheduler,
        <xsl:text disable-output-escaping="yes">
        &amp;</xsl:text><xsl:value-of select="process_election"/>);
  <xsl:for-each select="diffusion_function">
  assign_diffusion_function (scheduler,
        <xsl:text disable-output-escaping="yes">&amp;
        </xsl:text><xsl:value-of select="."/>);
  </xsl:for-each>
  <xsl:for-each select="on_node_info_update">
    assign_function_of_update_node_info (scheduler,
        <xsl:text disable-output-escaping="yes">&amp;
        </xsl:text><xsl:value-of select="."/>);
  </xsl:for-each>
  <xsl:for-each select="on_local_info_update">
    assign_function_of_update_local_info (scheduler,
        <xsl:text disable-output-escaping="yes">&amp;
        </xsl:text><xsl:value-of select="."/>);
  </xsl:for-each>
  <xsl:for-each select="process_placement">
    assign_function_of_process_placement (scheduler,
        <xsl:text disable-output-escaping="yes">&amp;
        </xsl:text><xsl:value-of select="."/>);
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Cette feuille de style génère le code du module noyau nécessaire à la création et l'initialisation des instances des GOrG, mais pas des tables que ces GOrG manipulent. Le listing A.2 permet de générer le code du module nécessaire à la création et l'initialisation des tables locales.

Listing A.2 – Second fichier XSL permettant de générer le fichier scheduler_loader.c

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>
<xsl:template match="/">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="schedulers">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="scheduler">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="nodes"/>
```

```
<xsl:template match="name"/>
<xsl:template match="file"/>
<xsl:template match="node_election"/>
<xsl:template match="process_election"/>
<xsl:template match="on_node_info_update"/>
<xsl:template match="on_local_info_update"/>
<xsl:template match="process_placement"/>
<xsl:template match="analyzers">
<xsl:for-each select="analyzer">
#ifndef SCHEDULER_LOADER_H
#define SCHEDULER_LOADER_H
<xsl:for-each select="global_infos">
/* WARNING, the static cluster size is here 64 nodes */
extern global_table_t <xsl:value-of select="."/>[64];
</xsl:for-each>
#endif
</xsl:for-each>
</xsl:template>


</xsl:stylesheet>
```

A.2 Feuille XSL-T générant fichier analyzer_loader.c

Le listing A.3 montre le code de la feuille de style XSL-T permettant de générer le fichier *analyzer_loader.c*.

Listing A.3 – Fichier XSL permettant de générer le fichier analyzer_loader.c

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>

<xsl:variable name="n"/>
<xsl:template match="/">
#define MODULE
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/module.h
        <xsl:text disable-output-escaping="yes">&gt;</xsl:text>
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/kernel.h
        <xsl:text disable-output-escaping="yes">&gt;</xsl:text>
<xsl:apply-templates/>
#include "request_queue_types.h"
#include "analyzer.h"

MODULE_AUTHOR("Geoffroy Ã©Valle");
MODULE_DESCRIPTION("Analyzer Loader");
MODULE_LICENSE("GPL");

int init_module (void)
{
<xsl:for-each select="analyzers">
```

```
<xsl:for-each select="analyzer">
  analyzer_t *__<xsl:value-of select="name"/>;
</xsl:for-each>
<xsl:for-each select="analyzer">
  __<xsl:value-of select="name"/> = create_analyzer (
        <xsl:text disable-output-escaping="yes">&amp;
        </xsl:text> <xsl:value-of select="file"/>, "<xsl:value-of select="name"/>");
 <xsl:variable name="n"><xsl:value-of select="name"/></xsl:variable>
  <xsl:for-each select="probes">
    <xsl:for-each select="probe">
  add_probe (__<xsl:value-of select="$n"/>, "<xsl:value-of select="."/>");
    </xsl:for-each>
  </xsl:for-each>
</xsl:for-each>
</xsl:for-each>
  return 0;
}

void cleanup_module (void)
{
  remove_analyzers ();
}
</xsl:template>

<xsl:template match="analyzer">
#include "<xsl:value-of select="file"/>.h"
</xsl:template>

</xsl:stylesheet>
```

A.3 Feuille XSL-T générant fichier probe_loader.c

Le listing A.4 montre le code de la feuille de style XSL-T permettant de générer le fichier *probe_loader.c*.

Listing A.4 – Fichier XSL permettant de générer le fichier probe_loader.c

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:output method="text"/>

<xsl:template match="/">
#define MODULE
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/module.h
        <xsl:text disable-output-escaping="yes">&gt;</xsl:text>
#include <xsl:text disable-output-escaping="yes">&lt;</xsl:text>linux/kernel.h
        <xsl:text disable-output-escaping="yes">&gt;</xsl:text>
#include "request_queue_types.h"
#include "probe.h"
<xsl:apply-templates/>
```

```
MODULE_AUTHOR("Geoffroy éValle");
MODULE_DESCRIPTION("Probe Loader");
MODULE_LICENSE("GPL");

int init_module (void)
{
<xsl:for-each select="probes">
<xsl:for-each select="timer_probe">
        create_probe_with_timer (<xsl:text disable-output-escaping="yes">&amp;
                </xsl:text><xsl:value-of select="file"/>,
                <xsl:value-of select="time"/>,
                "<xsl:value-of select="name"/>");
</xsl:for-each>
<xsl:for-each select="probe">
        create_probe (<xsl:text disable-output-escaping="yes">&amp;</xsl:text>
                <xsl:value-of select="file"/>,"<xsl:value-of select="name"/>");
</xsl:for-each>
</xsl:for-each>
        return 0;
}

void cleanup_module (void)
{
        remove_probes ();
}
</xsl:template>

<xsl:template match="probe">
#include "<xsl:value-of select="file"/>.h"
</xsl:template>

<xsl:template match="timer_probe">
#include "<xsl:value-of select="file"/>.h"
</xsl:template>

</xsl:stylesheet>
```

B FICHIERS GÉNÉRÉS POUR LA GESTION DYNAMIQUE DES COMPOSANTS D'UN ORDONNANCEUR

B.1 Prototype du fichier scheduler_loader.c

Listing B.1 – Prototype du fichier scheduler_loader.c

```
#define MODULE
#include <linux/module.h>
#include <linux/sched.h>
#include "gtypes.h"
#include "request_queue_types.h"
#include "scheduler.h"

MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");

int init_module (void)
{
  rwlock_init (&scheduler_lock);
  printk ("Scheduler initialized\n");
  return 0;
}

void cleanup_module (void)
{
  remove_schedulers ();
}
```

B.2 Prototype du fichier analyzer_loader.c

Listing B.2 – Prototype du fichier analyzer_loader.c

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

#include "request_queue_types.h"
```

```
#include "analyzer.h"

MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");

int init_module (void)
{
  return 0;
}

void cleanup_module (void)
{
  remove_analyzers ();
}
```

B.3 Prototype du fichier probe_loader.c

Listing B.3 – Prototype du fichier probe_loader.c

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include "request_queue_types.h"
#include "probe.h"

MODULE_AUTHOR("");
MODULE_DESCRIPTION("");
MODULE_LICENSE("GPL");

int init_module (void)
{
        return 0;
}

void cleanup_module (void)
{
        remove_probes ();
}
```

VU :
Le Directeur de Thèse :

VU :
Le Responsable de l'École Doctorale :

VU pour autorisation de soutenance
Rennes, le
Le Président de l'Université de Rennes 1

Bertrand FORTIN

VU après soutenance pour autorisation de publication :
Le Président du Jury,

Résumé

Les grappes de calculateurs sont aujourd'hui souvent utilisées pour exécuter des applications scientifiques. Ces applications peuvent être de natures différentes, séquentielles ou parallèles fondées sur le paradigme de communication par mémoire partagée ou par échange de message. Ainsi, des charges applicatives très diverses peuvent être exécutées sur des grappes. Or, les applications ne peuvent s'exécuter correctement que si elles tirent le meilleur profit des ressources disponibles. La gestion du partage des ressources entre les applications est traditionnellement à la charge de l'ordonnanceur du système. Or, actuellement, aucun système d'exploitation pour grappe n'offre un ordonnanceur global permettant d'exécuter correctement des charges applicatives données.

Les travaux présentés dans cette thèse portent sur la conception et la mise en œuvre d'un ordonnanceur global de processus modulaire et adaptable, dans le cadre du développement d'un système à image unique pour grappe. L'architecture de l'ordonnanceur global permet d'adapter la politique d'ordonnancement à la charge applicative à exécuter grâce à un mécanisme de configuration simple et dynamique. De plus, pour faciliter la mise en œuvre de politiques d'ordonnancement global, un environnement de programmation a été développé.

L'ordonnanceur global s'appuie sur des mécanismes de gestion globale des processus afin de déployer, déplacer, arrêter ou redémarrer les applications en cours d'exécution. Un mécanisme unique de gestion de processus, appelé *processus fantôme*, a été proposé et a permis de mettre en œuvre efficacement des mécanismes de création distante, de migration et de création/restauration de point de reprise. Ces mécanismes ont été développés pour pouvoir tirer profit des autres services du système à image unique pour grappe KERRIGHED notamment celui de gestion globale de la mémoire. Les mécanismes de gestion globale des processus offrent un mécanisme de gestion globale de *threads* au sein d'une grappe. Ce travail a abouti à la mise en œuvre d'une interface *POSIX thread* complète.

Les travaux de thèse présentés dans ce document ont abouti à la mise en œuvre d'un prototype fondé sur le noyau LINUX 2.4.24. Ce prototype a été intégré au sein du système à image unique pour grappe KERRIGHED, diffusé sous licence GPL (http ://www.kerrighed.org/).

L'ensemble de ces travaux a été validé par des expérimentations avec des applications scientifiques fournies par EDF R&D. Le support de *POSIX thread* de KERRIGHED a été validé par l'exécution d'applications OPENMP en utilisant un compilateur OPENMP standard visant l'interface *POSIX thread*.