Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles

Cyrille Martin

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_/_/_/_/_/_/_/_/_/_/

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique : Systèmes et Logiciels"

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'Ecole Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique"***

présentée et soutenue publiquement

par

Martin Cyrille

le 15 décembre 2003

Titre :

Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles

---

Directeur de thèse : Brigitte Plateau

JURY

M. JACQUES MOSSIÈRE , Président
M. FRANCK CAPPELLO , Rapporteur
M. RAYMOND NAMYST , Rapporteur
MME BRIGITTE PLATEAU , Examinateur (Directeur de thèse)
M. JACQUES BRIAT , Examinateur (Co-encadrant)
MME PASCALE ROSSE , Examinateur

Table des matières

Table des figures

Liste des tableaux

Remerciements

Tout d'abord je tiens à remercier Fabienne pour son soutien, ses encouragements et son aide pour éliminer les coquilles orthographiques sournoisement cachées dans ce document. Je la remercie particulièrement de m'avoir toujours supporté avec le sourire durant la rédaction de cette Thèse.
Je souhaite également remercier Jacques et Pascale pour m'avoir fait confiance et m'avoir accordé une grande liberté sur l'évolution de mes recherches. Je remercie Jacques pour ses précieux et parfois douloureux conseils de rédaction.
Merci à Olivier, Guillaume et Titou pour leur aide dans de nombreuses situations telles que réflexion scientifique, réflexion caféinée et apport substantiel de houblon à la réflexion.
Merci à Jean François pour sa relecture et pour ses conseils d'organisation du document.
Merci à Wilfrid qui a su supporter mes humeurs de développeur et de collaborateur avec le sourire (souvent :-)) et qui a réalisé une version distribuable de notre projet.
Je remercie également Brigitte pour sa grande patience et sa gentillesse.
Merci à Florence pour toujours avoir eu le courage et la persévérance de me confier l'administration de sa machine – souvent qualifiée (à tort) de noms d'oiseau – et pour tous les services qu'elle m'a rendus.
Merci à Corine et Rémi, compagnons de pauses tabagiques et houblonniques.
Merci à Pierre pour son aide dans la modélisation universelle du monde, mieux connue sous le nom de TP-complétude.
Merci à Stephane, célèbre administrateur du Icluster qui a toujours été disponible pour réparer et maintenir en état cette superbe machine.
Je souhaite enfin remercier les thésards du Laboratoire avec lesquels j'ai enduré de grandes joies et de grandes soirées et sans lesquels ces trois années de travail n'aurait pas été aussi agréables.
Je remercie également toutes les personnes du Laboratoire ID pour leur accueil et leur sympathie pendant quatre ans.

Chapitre 1

Introduction

De nos jours, les simulations par modèles mathématiques de phénomènes physiques sont utilisées dans de nombreux domaines. Pour réduire les coûts de développement, de nombreux domaines industriels utilisent des simulations informatiques pour remplacer des tests réels coûteux. Dans des domaines différents comme la génétique ou les statistiques, il s'agit d'analyser de très grandes quantités de données. La qualité des résultats de ces calculs ou traitements dépend de la taille des données traitées. Plus la masse de données est importante plus les résultats obtenus sont fiables et précis. On peut donc dire aujourd'hui que les besoins en calcul scientifique ne font que croître et dépassent largement les possibilités des microprocesseurs actuels.
Le calcul parallèle permet de réduire les temps de calcul de ces problématiques et ainsi d'augmenter le volume de données traitées en un temps donné. Les applications parallèles de calcul hautes performances deviennent de plus en plus répandues.
Les plates-formes de calcul suivent cette tendance et offrent des puissances de calcul toujours plus importantes. Il est possible de distinguer deux types de plates-formes parallèles.
Le premier est constitué de machines parallèles produites et vendues par les constructeurs (BULL, HP, IBM, SGI) comme des environnements de type "clé en main". Ces plates-formes fournissent de très bonnes performances car leur architecture est conçue de manière à répondre aux besoins des clients sur un domaine applicatif précis. Les constructeurs offrent un ensemble de services très complets, mais ces plates-formes propriétaires restent "fermées" et ciblées sur un domaine applicatif. De plus, le coût d'une architecture de ce type est très élevé.
Un deuxième type d'architecture parallèle a pour objectif de fournir une plate-forme modulaire et souple d'utilisation à faible coût. Ce type regroupe les architectures de grappes de stations de travail. Ces plates-formes parallèles sont mises en oeuvre avec des composants issus de la grande distribution qui permettent de réduire fortement leur coût. Le challenge de ces architectures est de fournir un environnement d'utilisation et les performances de plates-formes propriétaires à moindre coût.

Les travaux de recherche présentés dans la suite de ce document seront relatifs à des plates-formes parallèles de type grappe de stations de travail de très grandes tailles.

1.1 Grappes de calcul

Les grappes de calcul sont constituées par l'interconnexion de stations de travail via un réseau. L'objectif de ces architectures est de fournir une plate-forme parallèle haute performance avec du matériel à faible coût.

Les machines utilisées pour construire ces plates-formes sont des machines de bureau plus ou moins complexes. Ce sont soit des machines de type stations de travail, mono-processeur, soit des architectures de type serveur multiprocesseurs (Symetric MultiProcessor). Les grappes composées de machines multiprocesseurs sont alors nommées CLUMP ("CLUster of Multi-Processor").

Les réseaux utilisés pour interconnecter ces machines (noeuds) sont dits rapides, c'est à dire à haut débit et faible latence, comme Quadrics [68], Myrinet [26], SCI [15] ou encore un réseau standard de type LAN 802.

Le rapport coût / performances de ces architectures est très bon car le matériel utilisé dans leur conception est standard (non spécifique à cette architecture) et peu coûteux. Ces architectures offrent un environnement souple et modulaire. Si l'utilisateur désire plus de puissance, il "suffit" d'ajouter des machines sur le réseau d'interconnexion. Du fait de la croissance des besoins en puissance de calcul, la taille de ces architectures ne cesse d'augmenter. La taille moyenne d'une grappe variait entre 10 et 100 machines il y a 5 ans. Aujourd'hui une architecture standard se compose de centaines de machines. Certains projets disposent déjà de grappes de plus de 1000 processeurs (Cf. Top500 Juin 2003).

Le travail de recherche exposé par la suite s'intéresse à certains problèmes de passage à l'échelle engendrés par l'augmentation de la taille de ces architectures. La section suivante rappelle brièvement les problèmes posés par la programmation parallèle et sa mise en oeuvre sur des architectures parallèles à mémoire distribuée.

1.2 Environnements logiciels pour grappes

Tous les programmes de calcul hautes performances doivent exploiter au mieux les capacités de l'architecture sous-jacente. Sur une architecture séquentielle ce travail est effectué par le compilateur. Le programme de calcul est décrit dans un langage abstrait, générique et indépendant de l'architecture. Le compilateur traduit ce code en un langage machine optimisé pour une architecture cible. Ceci permet de rendre un programme portable. Le programme n'est écrit qu'une fois et la phase d'optimisation spécifique est réalisée par le compilateur.

Une autre façon de faire est de traduire (manuellement ou par compilation) le programme en une séquence d'appels de fonctions prédéfinies constituant ce que l'on appelle une machine parallèle virtuelle. Cette machine parallèle virtuelle peut alors être portée sur les machines cibles de façon à exploiter au mieux leurs spécificités

techniques.

La programmation parallèle ajoute plusieurs niveaux de difficultés. Le premier est l'expression du calcul à paralléliser. Il n'y a pas aujourd'hui de consensus sur un "bon langage de programmation" qui permette d'exprimer simplement les tâches de calcul et leur dépendances. Une deuxième approche consiste à utiliser les langages existants (LISP, PROLOG, FORTRAN, etc...) et d'en extraire le parallélisme. Une autre approche est d'étendre des langages existants par des opérateurs d'expression du parallélisme, de la synchronisation et des communications (Java, Ada, C, C++, etc...). Dans ce sens, le standard OpenMP [33] est orienté vers un parallélisme de données.
Le second problème provient de la difficulté de compiler efficacement ces langages sur une architecture CLUMP. Actuellement, on ne sait traiter ce problème que sur des architectures SMP.

C'est pourquoi aujourd'hui la programmation parallèle des grappes est basée sur l'utilisation d'une machine parallèle virtuelle permettant de voir un programme parallèle comme un réseau de processus communicants. Le code du processus est écrit dans un langage séquentiel classique ou dans un langage parallèle compilable sur un SMP. Les processus interagissent via une API (Application Programming Interface) de communication. MPI [43], PVM [56] et PM^2 [16] sont des exemples de telles machines parallèles virtuelles.

La mise en oeuvre de ces machines parallèles virtuelles repose sur une implantation efficace des communications entre processus et un déploiement efficace du réseau de processeurs virtuels (processus de l'application parallèle) sur le réseau de processeurs physiques.
En effet, les performances de ces communications sont critiques pour les performances du programme parallèle. Pour cette raison, elles doivent être optimisées pour exploiter au mieux les capacités physiques du réseau sous-jacent. Pour permettre la portabilité d'un programme parallèle, les environnements de programmation actuels comme MPI, PVM ou PM_2 fournissent une interface de communication abstraite et générique, permettant d'écrire un programme parallèle indépendamment de l'architecture réseau sous-jacente. L'optimisation des communications est réalisée par ces environnements pour les différents réseaux physiques existants.

Une lacune des environnements de ce type concerne l'efficacité du déploiement et, plus particulièrement son passage à l'échelle sur des grappes d'une centaine de noeuds et plus. C'est sur ce point en particulier que porte mon travail de thèse. La section suivante présente plus précisément les objectifs de recherche.

1.3 Les objectifs et contributions de cette thèse

La généralisation des architectures en grappes composées de machines (aussi nommées noeuds) standards du commerce est due à leur très bon rapport prix / performances. Ces plates-formes sont interconnectées par différents types de réseau : un réseau LAN 802, destiné à l'administration et les transferts de fichiers et un réseau haut débit et faible latence (Quadrics [68], Myrinet [26]) pour les besoins d'échange de données entre les différents processeurs virtuels d'une application de calcul parallèle hautes performances. Ces réseaux très performants destinés au calcul parallèle ont mis en évidence les lacunes des bibliothèques de communication par passage de messages portables comme MPI ou PVM par leur incapacité à exploiter conjointement plusieurs types de réseau.

Par ailleurs, les opérations lancement de programme parallèle incorporées à ces environnements ont montré une incapacité à passer l'échelle d'un grand nombre de noeuds. En effet cette opération, ainsi que la plupart des outils d'exploitation de ces plates-formes restaient conçus comme la répétition séquentielle d'une même opération sur chacun des noeuds composant la grappe.
De plus, il n'existait aucune bibliothèque générique et portable permettant le développement d'outils parallèles d'exploitation pour une architecture en grappe.

Les objectifs de ce travail de thèse sont la conception et la réalisation d'un outil de déploiement d'application parallèle sur des grappes de grandes tailles constituées de machines "Unix" (plus particulièrement Linux). Le rôle d'un tel outil est la mise en place des différents processus sur chaque machine cible, l'établissement d'un moyen de communication (liaison logique bipoints) entre ces différents processus ainsi qu'un ensemble de services de contrôle permettant de "piloter" une application parallèle (redirection des E/S, arrêt, signaux, etc...). Cet outil devra posséder les propriétés suivantes :

- **efficacité et passage à l'échelle**. Le parallélisme sera systématiquement exploité ;
- **portabilité**. L'outil devra être indépendant des protocoles de création de processus à distance ;
- **simplicité**. L'outil devra être simple à installer et à utiliser. En particulier, il ne doit pas exiger de droits particuliers du point de vue de l'administration système de la grappe.

Ces travaux ont conduit au développement d'un prototype de "lanceur" d'application parallèle nommé "Taktuk"[1]. Lors de sa conception, nous avons privilégié une architecture flexible, générique et modulaire permettant une bonne extensibilité de cet outil. Des mesures de performances ont permis de vérifier l'efficacité et le passage à l'échelle de cet outil. Son utilisation dans le cadre du laboratoire ID comme dans des contrats avec des partenaires industriels ont permis de le valider. En particulier, les services de déploiement et de contrôle fournis, ont été incorporé au noyau exécutif Inuktitut. Des outils d'administration de grappes de grandes tailles ont été développés

[1] brouillard en Inuit

avec Taktuk et sont distribués dans une suite linux pour grappe (Cf. chapitre 8). Taktuk est également utilisé dans un outil de réservation de ressource (système de "batch") nommé OAR [31] (Cf. chapitre 8).

1.4 Contexte de recherche

Ce travail de thèse a été réalisé au sein du laboratoire Informatique et Distribution (ID UMR 5132 CNRS-INPG-INRIA-UJF) appartenant à la fédération d'Informatique et Mathématique Appliquées de Grenoble (IMAG). Ce travail s'est déroulé en collaboration avec BULL dans le cadre du projet LIPS de l'action DYADE (BULL-ID-INRIA). L'un des thèmes de recherche du laboratoire ID est la conception d'un environnement de programmation parallèle pour le calcul hautes performances sur grappe de SMP. Athapascan exploite la généricité du langage C++ [77] pour offrir un ensemble de types de données permettant d'exprimer simplement des tâches de calcul et leurs dépendances via des objets (données) partagés. L'implantation de ces types de données automatise le placement des tâches de calcul sur les processeurs, leur synchronisation et les mouvements des objets entre les mémoires des différents noeuds. C'est le premier langage exécutable de façon transparente à l'utilisateur sur grappe de SMP.

Cet environnement repose sur un noyau exécutif portable proche d'une librairie de communication pour le calcul parallèle haute performance classique. La première version de ce noyau exécutif s'appelait Athapascan0 [40]. Athapascan0 assure l'interopérabilité d'un sous ensemble du standard MPI [43] et d'un sous ensemble du standard Posix Thread [75][54]. Il permet à des threads (processus légers) placés sur différents noeuds de communiquer via MPI. Il permet d'exploiter le parallélisme intra-noeud des grappes de SMP à l'aide de processus légers (threads) communiquant par mémoire partagée. L'exploitation du parallélisme inter-noeuds est réalisée par passage de messages via MPI. Les limites de cette approche proviennent de la dépendance à MPI et de la perte d'efficacité due aux compromis techniques nécessaires pour rendre l'utilisation de MPI compatible avec des processus légers.

Un nouveau de noyau exécutif appelé Inuktitut a été défini afin de gérer plusieurs environnements de communication pour le calcul hautes performances. Inuktitut doit permettre à Athapascan d'utiliser plusieurs couches de communication comme MPI, Corba[83], PM_2,etc... En plus de fournir un couche de portabilité entre les différents environnements de programmation parallèle existants, Inuktitut doit fournir la possibilité d'exploiter plusieurs de ces environnements simultanément. La raison est d'une part d'utiliser les différents réseaux d'une grappe si ils existent et d'autre part de communiquer entre les différents noeuds hétérogènes d'une grille de grappes via les protocoles Internet appropriés (XDR [74], Corba [83], etc...) L'architecture et l'utilisation d'Inuktitut sont étudiées plus en détail dans la section 8.1.1.

Les travaux de thèse se sont inscrits dans le cadre du déploiement d'un réseau

de processeurs virtuels Inuktitut sur des grappes de grandes tailles (centaines de machines).

1.5 Le plan de cette thèse

Ce document est organisé de la manière suivante. Le chapitre suivant présente le concept de machine parallèle virtuelle sur lequel repose la plupart des environnements de programmation parallèle existants. Une étude de cas présentera la façon dont ces environnements (MPI, PVM et PM^2) déploient leur machine parallèle virtuelle. Les différentes applications d'administration et d'exploitation de grappe existantes seront également décrites. Nous présenterons ensuite notre travail de conception et d'évaluation d'un lanceur portable efficace passant à l'échelle d'un très grand nombre de noeuds. Nous décrirons ensuite l'implantation de ce lanceur et nous étudierons ces performances. Nous présenterons également les différents projets auxquels ce lanceur a été intégré : la machine parallèle virtuelle Inuktitut, base de l'environnement ATHAPASCAN [47], les outils d'administrations (KaTools [62]) intégrés dans la distribution linux grappe, éditée par la société Mandrakesoft [7] et dans un ordonnanceur de travaux de type PBS [13] nommé OAR [31]. Nous conclurons enfin sur le bilan et les perspectives à court et moyen terme ce travail.

Première partie

Machines parallèles virtuelles

Chapitre 2

Concept de machine parallèle virtuelle

Les générations actuelles de grappes de calcul ont vu disparaître la plupart des machines parallèles à mémoire distribuée propriétaires et dédiées à une utilisation. Les dernières machines de ce type restent cantonnées dans un petit nombre de grand centres de recherche spécialisés. La majorité des utilisateurs de calcul scientifique utilisent des grappes construites avec du matériel largement distribué qui permet de réduire le coût d'une telle architecture. Si elles offrent des performances inférieures à des super-calculateurs élaborés à partir de matériel spécifique et optimisé, le rapport prix/performance de ces plates-formes a fortement contribué à leur développement et à leur utilisation très répandue. Les machines utilisées pour construire ces plates-formes vont de la simple station de travail à des serveurs de calcul multiprocesseurs (SMP). Les réseaux d'interconnexions utilisés vont d'un réseau local (LAN 802) à des réseaux plus performants et plus coûteux comme Myrinet [26] ou quadrix [68]. La grande souplesse de ces architectures permet d'adapter ces architectures aux besoins et aux moyens des utilisateurs qui les utilisent. Ceci a participé à la forte démocratisation de ces architectures et à la forte augmentation de leur taille. Il y a cinq ans une grappe de taille moyenne se composait de dizaines de machines. Aujourd'hui une grappe de calcul regroupe des centaines de processeurs et il existe déjà plusieurs projets possédant des plates-formes composées d'un ou plusieurs milliers de processeurs.

Les machines ou "noeuds" composant une grappe sont généralement homogènes car l'exploitation d'une plate-forme hétérogène reste coûteuse et difficile. Leurs caractéristiques techniques (processeur, mémoire, réseau, etc...) et logicielles (système d'exploitation) sont identiques. Cependant, la forte extensibilité de ces architectures peut réduire cette homogénéité au cours du temps. L'évolution technique des machines étant très rapide, le remplacement de noeuds défectueux ou l'ajout de nouvelles ressources de calcul introduit une variabilité de puissance entre les machines.

Tous les environnements de programmation sur grappe proposent des librairies qui implantent une interface de communication par passage de message. Elles permettent de décrire un programme parallèle comme un ensemble de processus

communicant via des opérateurs de communication bipoints ou multipoints. Ces environnements offrent aussi les outils de lancement et de contrôle de ces programmes parallèles sur les noeuds de la grappe. Dans le cadre des super-calculateurs ces bibliothèques de programmation et ces outils étaient directement optimisés par le constructeur sur une architecture matérielle cible. Comme il n'existe pas d'architecture matérielle type pour les grappes de calcul, les environnements de programmation parallèle recherchent un bon compromis efficacité / portabilité. Pour cela, ils définissent une machine parallèle virtuelle qui d'une part permet d'implanter au mieux l'interface applicative souhaitée (par exemple MPI) et d'autre part qui soit portable efficacement sur les différents systèmes d'exploitation et réseaux pouvant composer une grappe. D'une manière générale, une telle machine virtuelle apparaît comme un réseau virtuel de processeurs.

Pour faciliter la programmation des plates-formes à mémoire distribuée (grappes ou super-calculateurs), une partie des activités de recherche sur les environnements de programmation parallèle a porté sur le développement d'une abstraction homogène du matériel : une machine parallèle virtuelle (MPV). Cette abstraction est constituée de flots d'exécutions virtuels (processeurs logiques) et d'une interconnexion virtuelle (réseau logique).
Ceci permet d'abstraire les caractéristiques techniques de tous les noeuds utilisés. Dans cet environnement, le programmeur ne gère que des flots d'exécutions et les communications qui leur permettent d'échanger des données nécessaires à la progression de son calcul parallèle. La partie propre à la plate-forme physique, comme le nombre de noeuds, le réseau physique utilisé ainsi que les caractéristiques de l'architecture d'un noeud de calcul sont gérés par l'environnement implantant la machine parallèle virtuelle.

L'implantation de la machine parallèle virtuelle comprend un certain nombre de problèmes clefs à traiter. La gestion et l'optimisation des communications sur une architecture de type grappe sont très importantes pour les performances obtenues. Comme ces optimisations dépendent fortement du matériel utilisé, l'implantation de la machine parallèle virtuelle doit fournir des supports optimisés et efficaces pour la famille de réseaux physiques communément utilisés dans les grappes (LAN 802, Myrinet[26], SCI[15] et Quadrics[68]). Le portage d'une application parallèle sur une architecture matérielle différente se résume à utiliser une machine parallèle virtuelle configurée et optimisée pour cette architecture.

Préalablement à son exécution le programme parallèle doit être lancé, c'est à dire que l'on doit déployer le réseau de processus sur les noeuds de la grappe. Pour cela, il faut créer un ou plusieurs processus par noeuds selon les indications du programmeur ou d'un ordonnanceur. Cette opération de lancement est coûteuse car l'opération unitaire de création de processus distant l'est. Une mise en oeuvre séquentielle, souvent utilisée, est inefficace et ne passe pas à l'échelle. Enfin, l'environnement doit permettre à l'utilisateur d'interagir avec son programme parallèle comme il le fait avec un programme séquentiel (Entrées / Sorties console, arrêt, etc...).

La section suivante détaille plus précisément ces points critiques de l'implantation d'une machine parallèle virtuelle. Elle insiste sur les mécanismes d'abstraction qui permettent le support de plusieurs réseaux physiques différents et sur les méthodes utilisées pour déployer une machine parallèle virtuelle.

2.1 Réseaux et architecture logicielle de la communication

2.1.1 Les réseaux d'interconnexion

Les réseaux d'interconnexion de grappes utilisés aujourd'hui sont de deux types :

- des réseaux commutés hiérarchiques de type LAN 802 dont les débits sont de 100Mb/s, 1Gb/s et bientôt de 10Gb/s. Les commutateurs actuels se composent de un à deux niveaux. Un commutateur de premier niveau peut commuter 256 ports à 100Mb/s ou 48 ports à 1Gb/s de façon non bloquante. Commuter un plus grand nombre de ports nécessite d'interconnecter les commutateurs entre eux en grille ou via un commutateur de niveau supérieur. L'intérêt d'un tel réseau est qu'il permet d'interconnecter une grappe à un intranet sans problème en utilisant sur ce réseau tous les protocoles de l'internet. Ceci contribue à faciliter le portage de tous les environnements de programmation parallèle. L'inconvénient de ce type de réseau est la forte latence de communication du protocole 802. Un autre inconvénient est la difficulté de pouvoir construire une interconnexion non bloquante (où toutes les liaisons bipoints entre noeuds sont disjointes) de plusieurs milliers de noeuds.
- des réseaux haut débit / faible latence pouvant connecter plusieurs milliers de noeuds via des réseaux de clos. Quadrics [68] ou Myrinet [26] sont représentatifs de tels réseaux. L'exploitation des faibles latences du réseau nécessite la mise en oeuvre de protocoles appropriés incompatibles avec les protocoles de l'internet (TCP/IP).
- des réseaux haut débit / faible latence basés sur un accès distant à la mémoire des différents processeurs comme SCI [15]. Ces réseaux sont interconnectés via des tores 2D ou 3D. Les protocoles de communication de ces réseaux sont basés sur des accès directs en lecture ou en écriture de la mémoire des noeuds distants. Ils sont donc incompatibles avec les protocoles internet standard basés sur TCP/IP.

2.1.2 Architecture logicielle de la communication

Les fonctions de communication sont souvent architecturées en couches. Toutes les couches jusqu'au niveau transport sont intégrées dans le système d'exploitation ou les bibliothèques fournies. L'efficacité de la communication étant critique en calcul parallèle, les laboratoires de recherche et les fabriquants de réseaux de grappes proposent leurs propres protocoles pour optimiser les échanges sur le réseau, BIP [70] et GM [4]

sur Myrinet et QsNet [19] de Quadrics.
Des protocoles alternatifs ont été proposés pour les réseaux LAN 802 avec le même objectif, parmi eux GAMMA [35] et SBP [71]. Aucun de ces protocoles n'a passé l'échelle de plus d'une dizaine de noeuds et n'a pu se substituer au protocole TCP/IP.
Malgré les différences de protocoles, les fonctionnalités de transport sont équivalentes. Elles permettent d'établir un nombre quelconque de liaison bipoints entre les processus des noeuds connectés.

2.2 Communications et machine parallèle virtuelle

2.2.1 Notion de réseau virtuel

Les protocoles de communication pour le calcul parallèle sont les bases des environnements de programmation, par passage de messages (MPI [43] et PVM [56]), par messages actifs (Inuktitut (Chapitre 8), Nexus [45]) ou les appels de procédures à distances (PM^2 [65], Corba [83]). Ce sont des communications bipoints ou multipoints (diffusion, accumulation, réduction) qui peuvent impliquer tous les noeuds de façon plus ou moins structurée.
Une grande partie des programmes parallèles de calcul scientifique organisent les processus communiquant en grille multidimensionnelle. Les fonctionnalités des protocoles de communication sous-jacents permettent d'établir un nombre quelconque de liaisons bipoints entre les processus interconnectés.
Il est alors du rôle de l'implantation de la machine parallèle virtuelle (bibliothèque de communication) d'établir les liaisons implantant les schémas de communication offerts par l'API (Abstract Programming Interface) de communication. Par exemple MPI offre un ensemble de primitives de communication collectives, c'est la bibliothèque MPI qui devra "construire" l'arbre couvrant les noeuds impliqués dans cet échange.

2.2.2 Les grappes multi-réseaux

Du fait de l'usage d'ordinateurs standards du commerce, les noeuds d'une grappe comportent automatiquement une à deux connexions réseau de type LAN 802. Ainsi toute grappe peut comporter deux réseaux : un réseau standard LAN 802 et un réseau haut débit / faible latence (Myrinet, Quadrics ou SCI).
Pour optimiser le débit d'un réseau lorsque la charge de communication est trop importante, il est possible de doubler un réseau en mettant deux interfaces d'un même réseau sur chaque noeud. Dans ce cas on parle souvent de transport multi-rail [38].
Par ailleurs, une grappe peut être obtenue par l'assemblage de deux sous grappes : une grappe Myrinet et une grappe SCI par exemple. Dans le cas de grappe mixte il se pose immédiatement deux problèmes :

- lors d'un échange, quel réseau emprunter lorsque les noeuds impliqués sont connectés à deux réseaux ?
- lors d'un échange entre deux noeuds connectés à des réseaux différents, comment router au mieux les messages de l'un à l'autre ?

Dans le cas d'un réseau multi-rail, la première question se ramène à un problème d'équilibrage de charge. Si les deux réseaux sont différents, il faut tenir compte des propriétés des réseaux relativement à la requête de communication. Par exemple, une requête peut passer en un paquet sur un réseau et en plusieurs sur un autre. Ces points font aujourd'hui l'objet de recherches [65] et seul le réseau de type multi-rail est supporté dans le cache de Quadrics.
Le second point peut être abordé de plusieurs manières. Une première est de considérer que le protocole fédérateur IP est toujours disponible sur l'ensemble des noeuds et que s'il existe un chemin physique entre deux noeuds IP assurera le routage. Cette solution implique le surcoût de IP sur un réseau à faible latence. Une seconde manière est de considérer qu'il existera toujours un réseau fédérateur LAN 802 interconnectant tous les noeuds de la grappe. Ainsi lorsque l'on communique d'une sous grappe SCI à une sous grappe Myrinet, on emprunte le réseau LAN 802 qui connecte tous les noeuds. Enfin une dernière façon de procéder est d'utiliser des noeuds passerelles appartenant aux deux sous grappes. Ces noeuds passerelles assurent le relais de communication entre les deux sous grappes, ainsi que le routage (calcul des routes) et la conversion des protocoles faibles latences utilisés. Cette voie difficile a été explorée par Olivier Aumage [22] dans son travail de thèse. Il a choisi d'insérer ces fonctions de passerelles dans la bibliothèque de communication Madeleine II, c'est à dire au niveau de l'implantation de la machine parallèle virtuelle PM^2.

2.3 Interaction avec l'environnement

Un programme séquentiel classique interagit avec l'utilisateur et accède à des ressources comme des fichiers. Par analogie un programme parallèle possède les mêmes besoins. Pour cette raison la machine parallèle virtuelle fournit un ensemble de fonctionnalités permettant au programme parallèle (distribué sur plusieurs machines) d'interagir avec l'utilisateur et d'accéder des ressources diverses.

2.3.1 Système de fichiers parallèles

Les données nécessaires à un calcul parallèle ne peuvent souvent pas être toutes placées en mémoire et il est nécessaire de les lire ou d'écrire ces données dans des fichiers au cours de l'exécution. Le placement des données sur les disques locaux des noeuds de la grappe, la réalisation et la synchronisation des accès parallèles ont été l'objet de recherches [86][87] et ont conduit à la définition d'un standard MPIIO[39] inclu dans MPI.
L'implantation de la machine parallèle virtuelle est responsable d'une part de l'implantation des fichiers de données nécessaires sur les systèmes de fichiers locaux, systèmes de fichiers distribués [57] ou réseau de stockage [73][72] disponibles sur une grappe. Elle est également responsable du choix des réseaux à utiliser, de l'établissement des liaisons de communication nécessaires à ces échanges de données et des synchronisations des caches des systèmes de fichiers locaux utilisés pour accélérer les

accès aux données.
Une implantation efficace capable de supporter les différents modes d'organisation physique d'un système de fichiers parallèle est encore un domaine de recherche très actif [30][73] et en particulier au laboratoire ID [60].

2.3.2 Contrôle d'une machine parallèle virtuelle

On regroupe sous le nom de fonction de contrôle : le démarrage et l'arrêt du programme (démarrer et arrêter tous les processus de la machine parallèle virtuelle), la redirection des entrées / sorties console du programme parallèle (interaction avec l'utilisateur via affichage et clavier) ainsi que des fonctions propres à une machine parallèle virtuelle comme par exemple la création, l'arrêt ou le retrait de flots d'exécution sur les noeuds.

Toutes ces opérations se ramènent :
- à la diffusion d'une donnée sur l'ensemble des noeuds depuis un point de contrôle (processus console) dans le cas d'une entrée au clavier ou d'un signal de contrôle.
- à l'accumulation de données depuis tous les noeuds dans le cas d'une sortie de données à la console ou l'émission d'un signal.
- à l'émission ou réception d'une donnée entre le point de contrôle et un noeud particulier dans le cas d'une entrée ou sortie de données ou d'un signal de contrôle.

Chaque machine parallèle virtuelle fournit un point de contrôle (processus) et un réseau logique de liaisons permettant à ce point de contrôle de communiquer en bipoints et en multipoints avec les différents processus composant la machine parallèle virtuelle répartis sur les noeuds de la grappe. Ces liaisons utilisent l'un des réseaux physiques disponibles de la grappe. Ces communications de contrôle peuvent être multiplexées avec les communications de calcul sur le réseau à faible latence. Pour éviter de dégrader les performances des communications de calcul , il est plus raisonnable d'utiliser un réseau moins performant comme un réseau LAN 802.

Aujourd'hui les plates-formes interconnectées avec un réseau physique très performant (Myrinet, Quadrics, SCI) possèdent toujours une interconnexion moins performante de type LAN 802 destinée à l'administration et à la maintenance de la plate-forme. Dans ce cas une machine parallèle virtuelle utilise des réseaux physiques différents pour l'échange de données et les communications de contrôle. Dans le cas ou la plate-forme physique ne possède qu'un seul type d'interconnexion soit rapide soit LAN 802, il est intéressant de séparer ces différents types de communication en établissant des réseaux logiques dédiés. Une première raison provient de la différence entre les maillages nécessaires au contrôle et celui nécessaire à un calcul parallèle. L'autre raison est de permettre un partage du réseau basé sur des priorités (communication de calcul plus prioritaires) pour ne pas perturber les communications destinées à l'avancement du calcul.

2.4 Déploiement d'une MPV

L'opération de déploiement d'une machine parallèle virtuelle est l'opération de transition entre un état où un processus sur un noeud de la grappe émet une requête de lancement d'un programme parallèle à l'état où ce programme parallèle commence son exécution sur la grappe. Ce déploiement se déroule en plusieurs phases :

- La première phase consiste à démarrer un processus sur chacun des noeuds. Ce processus va exécuter le prologue d'initialisation de la machine parallèle virtuelle avant l'exécution du code applicatif.
- La seconde phase consiste à établir les liaisons de communications de contrôle et à initialiser les protocoles associés propres à la machine parallèle virtuelle.
- La troisième phase consiste à établir les liaisons de communication de calcul et à initialiser les protocoles de communication propre à la machine parallèle virtuelle. Ces liaisons peuvent être mises en place sur un réseau physique différent du premier. Par exemple, le réseau de contrôle peut être établi au-dessus d'un réseau LAN 802 et le réseau dédié au calcul peut être établi sur un réseau à faible latence de type Myrinet. L'établissement de ces liaisons exige des échanges d'adresse réseau entre les noeuds.
- Une dernière phase peut initialiser l'environnement d'accès parallèle à des fichiers si il existe (MPIIO). Cette dernière étape démarre l'exécution du programme parallèle.

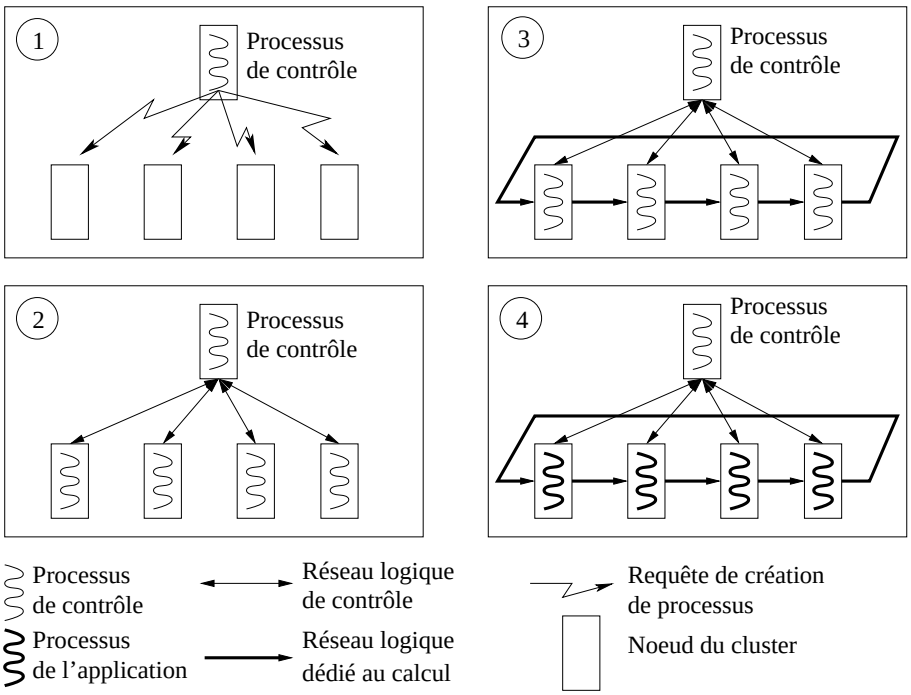


FIG. 2.1 – Exemple de mise en oeuvre d'un ensemble de processus communiquant utilisant des protocoles standards de création de processus distant.

La figure 2.1 présente un exemple de réalisation d'un déploiement sur un réseau physique de type LAN 802. La machine parallèle virtuelle déployée est constituée de

deux réseaux logiques l'un dédié au contrôle de la machine parallèle virtuelle l'autre dédié aux communications engendrées par le calcul d'une application parallèle. Pour des raisons de clarté de la figure le réseau applicatif dédié au calcul a une topologie en anneau. Cette figure illustre les différentes phases énoncées ci-dessus.

Les opérations de base du déploiement sont l'établissement des liaisons de communication (dans la plupart des cas TCP pour le réseau de contrôle) et la création d'un processus à distance (protocole d'exécution distante de type rsh [76] ou ssh [41]). Il est clair que l'implantation séquentielle de cette suite d'étapes croît au mieux linéairement en fonction du nombre de noeuds. La parallélisation de ces opérations est un moyen d'améliorer d'une part les performances de ces opérations et d'autre part d'augmenter le passage à l'échelle du déploiement.

Deux approches sont possibles pour la mise en oeuvre des étapes initiales du déploiement. La première consiste à utiliser les protocoles standards de l'internet pour la création de processus à distance (rsh [76], ssh [41]). Ces protocoles sont relativement coûteux, une part importante du coût de ces protocoles provient d'une part de leur généralité et d'autre part des vérifications et authentifications liées à la sécurité. Ces coûts s'ajoutent au coût incompressible de création de processus.

Une façon de contourner ces coûts inutiles est de substituer aux protocoles standards un protocole dédié au déploiement d'une machine parallèle virtuelle de la façon suivante. Une première phase consiste à établir un réseau de processus communiquant sur l'ensemble des noeuds cibles à l'aide de protocole standard et donc coûteux. Le lancement d'un application parallèle se traduit donc par la diffusion dans ce réseau de la requête de lancement. Chaque processus de ce réseau se duplique ainsi que ces liaisons d'interconnexions afin d'obtenir une nouvelle instance de machine parallèle virtuelle munie de son réseau de contrôle. L'intérêt d'une telle méthode est de factoriser les coûts liés à l'utilisation de protocoles standards dans le déploiement de plusieurs applications parallèles. Le réseau de lancement (sa topologie) peut également être optimisé pour cette seule opération.
Le prix à payer est celui de la fiabilité de ce réseau initial qui doit résister à tout mauvais fonctionnement des applications ou des noeuds.

La figure 2.2 représente un exemple de duplication de machine parallèle virtuelle ayant pour objectif de factoriser les coûts de création par des protocoles standards. Dans l'état initial étiqueté 0, il existe seulement une entité de machine parallèle virtuelle dédiée au lancement de nouvelles instances de machines parallèles virtuelles. Les états 1 et 2 montrent la demande de lancement effectuée sur cette machine parallèle virtuelle par un processus de contrôle. La machine parallèle virtuelle dédiée au lancement s'exécute sur l'ensemble de noeuds de la grappe (4 noeuds dans cet exemple). Le processus de contrôle demande le lancement d'une application parallèle sur 3 noeuds de cette grappe. Cette requête est diffusée sur le réseau de contrôle, l'état 2 de la figure représente la création locale d'un nouveau processus sur l'ensemble des noeuds cibles. Ces nouveaux processus sont ensuite connectés localement à la

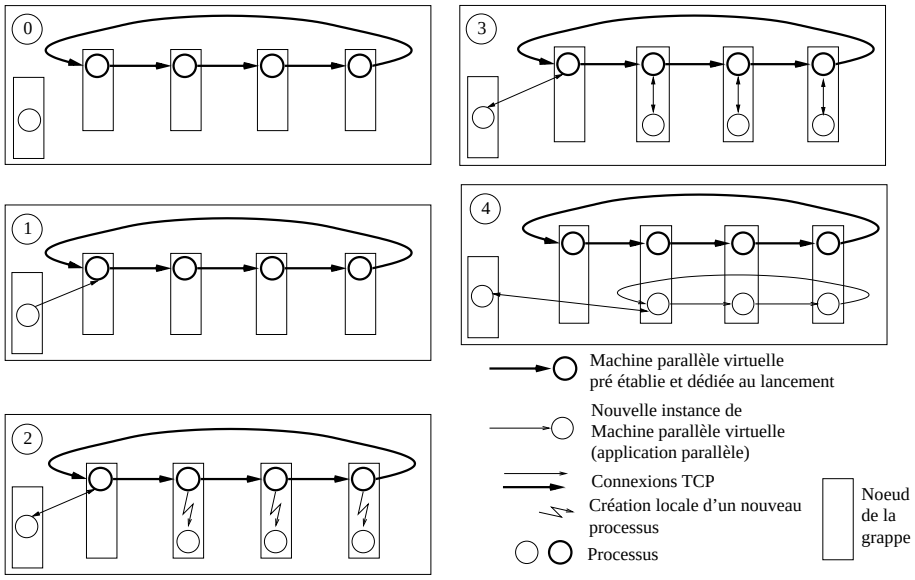


FIG. 2.2 – Exemple de mise en oeuvre d'un ensemble de processus communiquant par duplication d'une machine parallèle virtuelle existante.

machine parallèle virtuelle de lancement (état 3). Cette première connexion est utilisée pour échanger les adresses réseaux nécessaires aux nouveaux processus pour réaliser leur propre interconnexion (avec une topologie en anneau sur l'exemple). L'état 4 présente le résultat de ce lancement, une fois que le nouvel ensemble de processus s'est interconnecté les connexions avec la machine parallèle virtuelle de lancement sont fermées. Cette nouvelle instance est toujours reliée au processus de contrôle. Ainsi, on obtient une nouvelle instance de machine parallèle virtuelle prête à commencer l'exécution du programme parallèle. Dans cet exemple, nous avons présenté un modèle de machine parallèle virtuelle reposant sur un seul réseau physique et interconnectée avec une topologie logique en anneau de manière à simplifier la figure. Il est possible de réaliser des configurations plus complexes (utilisation de plusieurs réseaux de topologies différentes) sur le même principe.

2.5 Outils d'exploitation

Nous avons décrit les principes de mise en oeuvre des machines parallèles virtuelles sous-jacentes aux environnements de programmation parallèle. Les applications scientifiques ne sont pas toutefois les seuls exemples de programmes parallèles exécutés sur une grappe.
En effet, l'administration et l'exploitation d'une grappe nécessitent aussi d'effectuer des actions élémentaires sur tout ou partie des noeuds d'une grappe. Ce sont par exemple les opérations de diffusion / installation de logiciels (compilateurs, bibliothèque de programmation, etc...), de modification de fichiers de configuration ou l'installation de fichiers applicatifs sur l'ensemble des noeuds de la grappe.

Symétriquement, il est nécessaire d'extraire des fichiers utilisateurs du système de chaque noeud de la grappe. Parmi ces opérations, on trouve également toutes les fonctionnalités de diffusion de signaux de contrôle (arrêt, démarrage de service ou destruction de processus) ou acquisition d'événement de surveillance système ("monitoring").

La plupart de ces commandes sont implantées séquentiellement et ne permettent pas un passage à l'échelle satisfaisant sur des plates-formes composées de centaines de noeuds. La seule solution est donc de réimplanter ces commandes avec une version parallèle. Pour cela, il est possible d'utiliser un environnement de programmation parallèle qui utilise les principes de déploiement cités dans la section précédente. Les solutions proposées dans le domaine des outils d'exploitation seront également étudiées dans le chapitre 3 puisque dans la plupart des cas, elles reposent sur un ensemble de processus communiquant (base d'une machine parallèle virtuelle).

2.6 Conclusion

Nous avons vu que tout environnement de programmation parallèle repose sur l'implantation d'une machine parallèle virtuelle fournissant une interface de programmation ("API"). Cette machine parallèle virtuelle offre l'abstraction d'une architecture de grappe de stations de travail SMP comme un réseau logique de processus et de fils d'exécution.
Une activité importante de R&D a été et est encore consacrée à l'implantation efficace d'opérations de communication exploitant au mieux le ou les réseaux d'interconnexion [15][68][26]. Plus récemment, un effort s'est porté sur les systèmes de fichiers parallèles (MPIIO [73][59][39], etc...). Peu d'attention a été portée sur la phase de déploiement d'une machine parallèle virtuelle qui constitue la base de tout environnement de programmation parallèle. Cette phase a pendant longtemps été traitée de manière ad hoc par les environnements de programmation considérés. Cependant son implantation portable passe difficilement l'échelle d'un grand nombre de noeuds et demande des adaptations spécifiques pour les grosses configurations. Il en est de même dans le domaine des outils d'administration et d'exploitation quand ils existent.
Cette thèse montre qu'il est possible de définir un "lanceur" général et portable qui passerait l'échelle par un usage systématique du parallélisme et qui pourrait servir pour le déploiement de toute machine parallèle virtuelle (MPI [43][49][6], Inuktitut [5], etc...) comme pour la réalisation d'outils d'exploitation parallèle.
Un tel lanceur aurait en charge les deux étapes initiales de la phase de déploiement qui sont : (i) la création des processus sur les noeuds de la grappe et (ii) la construction du réseau de contrôle permettant à ces processus d'interagir avec le point de contrôle. Toutefois, il est nécessaire d'étendre les fonctionnalités citées ci-dessus avec des fonctionnalités de communication. En effet, la construction des différents réseaux applicatifs ou le transfert de données ou de fichiers nécessite une interaction entre les processus (acquisition d'adresses réseau, etc...).

Le chapitre suivant fait le point sur la façon dont les environnements de programmation parallèle et les outils d'exploitation existants déploient leur machines parallèles virtuelles.

Chapitre 3

Déploiement de machines parallèles virtuelles : étude de cas

Dans ce chapitre, nous faisons le point sur la façon dont sont déployés les environnements de programmation parallèle les plus utilisés. Nous décrirons également les lanceurs et les outils existant pour l'administration et l'exploitation des grappes de grandes tailles. Cette étude nous permettra de tirer un bilan relatif à nos objectifs de lanceur générique efficace.

3.1 Déploiement d'environnement de programmation parallèle

Cette section présente les méthodes de déploiement utilisées dans les principaux environnements de programmation parallèle. Ils comprennent la mise en oeuvre des machines parallèles virtuelles sur lesquelles reposent ces environnements. Comme les applications parallèles de calcul scientifique ont des temps d'exécution très importants, les efforts de recherche sur l'optimisation de la séquence d'initialisation et de mise en oeuvre de ces environnements ont été négligés. Cependant, la forte augmentation de la taille des architectures de type grappe pose au delà du problème du temps de déploiement, un problème critique de passage à l'échelle.

3.1.1 PVM

L'environnement de programmation et d'exécution de programme parallèle PVM [56] (Parallel Virtual Machine) a été conçu au laboratoire d'Oak Ridge (Tennessee) en 1989. Il fait partit des premières implémentations d'une machine parallèle virtuelle. L'implantation de la machine parallèle fournit par PVM respecte le modèle présenté dans le chapitre 2. Cette implantation repose entièrement sur les protocoles standards

de l'internet (TCP ou UDP sur IP).

La mise en oeuvre de PVM consiste à créer un ensemble de processus communiquant sur les noeuds de la grappe. Ces processus seront utilisés par toutes les futures applications parallèles qui s'exécuteront sur cette machine parallèle virtuelle.
Le déploiement consiste à démarrer un premier processus (processus de contrôle cf. chapitre 2). Ce processus correspond à la console virtuelle de la machine parallèle virtuelle avec laquelle l'utilisateur peut interagir. Le déploiement explicite commence via cette console, ce processus crée alors les différents processus distants sur les noeuds de la grappe spécifiés en utilisant des protocoles de l'internet standard (rsh ou ssh). Une fois ces processus créés, ils s'interconnectent via des connexions UDP pour former un réseau virtuel de processus. Ces processus seront exploités par la suite comme des processeurs virtuels dédiés aux communications par toutes les instances de programme parallèle qui s'exécuteront.

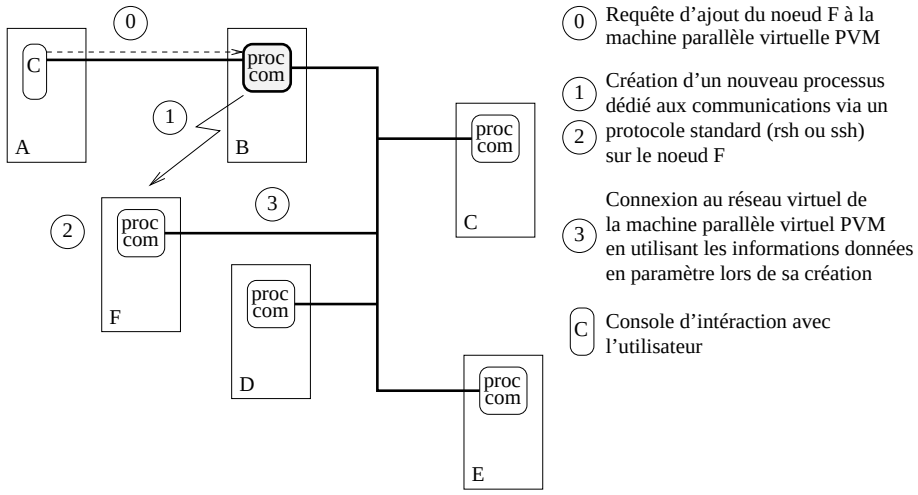


FIG. 3.1 – Ajout d'un processus (processeur virtuel dédié aux communications) à une machine parallèle virtuelle existante. Cette opération est répétée itérativement pour la mise en oeuvre complète de l'environnement PVM.

La figure 3.1 détaille l'ajout d'un processus à la machine parallèle virtuelle. Le déploiement complet de l'environnement est effectué par la répétition séquentielle de cette opération. Pour la création du premier processus cette opération est effectuée plus simplement, directement par le processus de contrôle. Ce premier processus joue le même rôle que les autres mais possède un fonctionnalité supplémentaire : ils traitera toutes les créations ultérieures de processus.

La figure 3.2 montre comment la machine parallèle virtuelle est utilisée. Comme le démarrage de cet environnement et coûteux et linéaire en fonction du nombre de noeuds, cette création de machine parallèle virtuelle est factorisée de manière à permettre le lancement successif ou conjoint (exemple de la figure 3.2) de plusieurs

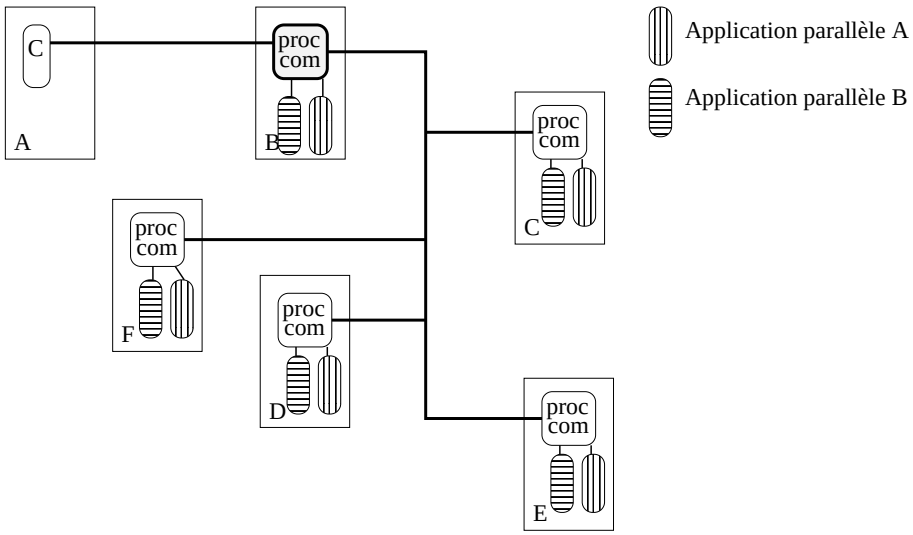


FIG. 3.2 – Utilisation d'un machine parallèle virtuelle PVM par deux programmes parallèles **A** et **B**.

programmes parallèles. Lorsque l'utilisateur demande, via le processus console, le lancement d'une application parallèle, cet ordre de création de processus est diffusé sur le réseau virtuel. Les processeurs virtuels de communication concernés créent localement un nouveau processus pour exécuter le code de l'application parallèle. Toutes les communications[1] de ces applications parallèles passeront par les processeurs virtuels de communication qui garantissent l'interconnexion de ces différents processus applicatifs.

Cette méthode de déploiement souffre d'un coût de mise en oeuvre linéaire en fonction du nombre de noeuds. Ce coût est effectivement mis en facteur par l'utilisation répétée d'une même entité de machine parallèle virtuelle PVM. Cependant, cet unique ensemble de processus communiquant doit garantir une forte fiabilité par rapport aux nombreuses défaillances possibles (de la part des applications parallèles ou des noeuds de la grappe).

3.1.2 MPI

Le standard MPI [43][44] définit une interface de programmation ("API") de haut niveau permettant de réaliser simplement des communications par passage de message. Ce standard a été défini spécifiquement pour exploiter des architectures à mémoire distribuée (grappes). Ce standard fixe uniquement les fonctionnalités fournies par cette interface de programmation. L'implantation de la machine parallèle virtuelle nécessaire au fonctionnement de cet environnement ainsi que les méthodes

[1]Par défaut, toutes les communications sont acheminées par les processeurs de communication. Cependant, PVM fournit également la possibilité de créer des liens de communication directs entre deux noeuds.

de déploiement sont laissées libres.

3.1.2.1 Mpich

Mpich [49] est une implantation du standard MPI réalisée au Laboratoire Argonne à Chicago. Cet environnement de programmation parallèle par passage de message repose sur une machine parallèle virtuelle semblable à celle décrite dans le chapitre 2. L'implantation de cet environnement inclu deux mécanismes de déploiement d'application parallèle sur des grappes. Ces mécanismes sont décrits de manière précise dans la section 2.4 car ils sont utilisés dans de nombreux autres environnements.
Ces deux méthodes consistent à démarrer un processus qui sert de lanceur d'application parallèle nommé "mpirun". Ce processus sera utilisé par la suite, une fois le déploiement terminé, comme console de contrôle de l'application parallèle démarrée. Lors de la phase de déploiement ce processus est chargé de créer le réseau de processus communiquant sur les noeuds concernés de la grappe. Pour cela, il utilise soit les protocoles standard de création de processus distant comme rsh [76] ou ssh [41], soit une machine parallèle virtuelle préexistante est dédiée au lancement.

Dans le premier cas, (figure 2.1 du chapitre 2) la création des processus distants est effectuée séquentiellement par le processus "mpirun" en utilisant des protocoles standards. Lors de leur création, les processus reçoivent en argument l'adresse réseau sur laquelle ils pourront se connecter au processus mpirun. Le réseau de contrôle ainsi formé est utilisé pour diffuser aux processus les informations nécessaires à l'interconnexion du réseau applicatif (complètement maillé).

Dans le second cas, (figure 2.2 du chapitre 2) la création des différents processus est effectuée à l'aide d'une machine parallèle virtuelle existante. Cette machine virtuelle est dédiée aux opérations de lancement de programme parallèle (nouvelles instances de machines parallèles virtuelles). Cette machine parallèle virtuelle dispose d'une interconnexion virtuelle simple basée sur une topologie en anneau. Elle est mise en oeuvre une seule fois par la méthode énoncée ci-dessus. Dans ce cas, le processus "mpirun" se connecte à cette machine parallèle virtuelle et diffuse un ordre de création de processus sur les noeuds concernés. Un nouveau processus est créé sur chaque noeud cible. Ce processus est interconnecté aux autres via la machine parallèle virtuelle dédiée au lancement. A ce stade, seul l'ensemble des processus a été dupliqué, la duplication de l'interconnexion est effectuée en utilisant la machine parallèle virtuelle (notée MVP) originelle. Une fois que cette MVP a été dupliquée les liaisons avec cette dernière sont fermées. On obtient donc une nouvelle MVP indépendante de la première.

Une fois que ce réseau de processus est mis en place, chaque processus termine l'initialisation de la machine parallèle virtuelle (création de nouvelle liaisons réseau, etc...). La machine parallèle virtuelle initialisée, les processus commencent l'exécution

du code de l'application parallèle.

Le mécanisme de déploiement reposant sur une machine parallèle dédiée au lancement permet de factoriser le coût linéaire en fonction du nombre de noeuds, dû à l'utilisation de protocoles standards comme rsh et ssh. Cependant, comme la topologie de cette première machine parallèle est basée sur une topologie en anneau, le temps de diffusion de l'ordre de duplication reste linéaire en fonction du nombre de noeud.

3.1.2.2 MPI LAM

L'université de l'Ohio aux Etats Unis propose également une implantation du standard MPI nommé MPI LAM [6]. La machine parallèle virtuelle sur laquelle est basée cet environnement a un fonctionnement proche de celle de l'environnement PVM. La mise en oeuvre de cet environnement consiste à créer une première instance de machine parallèle virtuelle en utilisant de manière itérative des protocoles de création de processus distants standards. Cette opération coûteuse consiste donc à démarrer une machine parallèle virtuelle dédiée aux communications de la même manière que PVM. Les processus qui la composent sont dédiés aux communications et au contrôle de l'application parallèle.

Le déploiement d'une application parallèle s'effectue ensuite par la diffusion d'un ordre de lancement sur cette machine parallèle virtuelle. Chaque processus de contrôle crée localement un nouveau processus qui exécutera le code de l'application parallèle. Comme dans l'environnement PVM, toutes les communications des processus de l'application parallèle seront traitées par les processus de communication de la machine parallèle et acheminées par son réseau virtuel.

L'objectif de cette méthode de déploiement est de factoriser l'utilisation coûteuse des protocoles standards d'exécution distante en permettant d'exécuter successivement plusieurs programmes parallèles sur une même machine parallèle virtuelle. Contrairement à l'environnement PVM cette machine parallèle virtuelle ne supporte qu'une instance de programme parallèle simultanément. Si l'on souhaite exécuter un autre programme parallèle il faudra au préalable démarrer une nouvelle machine parallèle virtuelle.

Le coût de la création de la machine parallèle virtuelle de cet environnement est linéaire en fonction du nombre de noeuds. Cependant, l'interconnexion de son réseau de processus est de type complètement maillé, ce qui autorise une diffusion de l'ordre de lancement d'un programme parallèle logarithmique en fonction du nombre de noeuds. Sa conception proche de PVM permet à MPILAM de supporter une gestion dynamique du nombre de noeuds composant la machine parallèle virtuelle (Standard MPI 2 [44]).

3.1.3 Score

Le projet Score [55] est développé par le "PC cluster consortium" au Japon. Ce projet fournit une implantation de MPICH [17] et de PVM utilisant une machine parallèle virtuelle commune nommée SCore-D. Cet environnement est utilisé pour le contrôle et l'initialisation des applications parallèles qui utilisent par la suite une bibliothèque de communication nommée PM[78] basée sur un réseau Myrinet.
La mise en oeuvre de cet environnement est effectuée par l'utilisation d'une machine parallèle virtuelle existante. Cette première instance de machine parallèle virtuelle est démarrée séquentiellement de manière centralisée par l'utilitaire "scout"[2]. Ce premier déploiement initialise un ensemble de processus interconnectés en anneau (Cf. figure2.2).
Le déploiement d'une application parallèle est effectué par un processus de contrôle ("scrun") qui diffuse une requête de création de processus en utilisant l'environnement préexistant. Une nouvelle interconnexion au dessus du réseau Myrinet est créée par l'initialisation de la bibliothèque PM. Cette interconnexion sera dédiée au calcul.

Cet environnement d'exécution de programme parallèle propose des services de contrôle avancés sur les applications parallèles en cours d'exécution. Ces services permettent de réaliser un "gang scheduling" [51] entre plusieurs applications et un mécanisme de "checkpoint" permettant d'interrompre et de reprendre l'exécution d'une application parallèle.

Cet environnement est exploité sur des grappes de grandes tailles (1040 Processeurs), cependant le coût du déploiement de cet environnement est linéaire en fonction du nombre de noeud. Ceci justifie le choix d'utiliser un protocole d'exécution distante peu coûteux comme "rsh". La diffusion d'une requête de création ou de contrôle ("gang scheduling" et "checkpoint") sur la machine parallèle virtuelle reste également linéaire du nombre de noeuds (topologie en anneau).

3.1.4 PM_2

L'environnement de programmation PM_2 [65] a été développé au Laboratoire LIP à Lyon et à l'Université de Lille. La machine parallèle virtuelle sur laquelle il repose est nommée "Madeleine" [22]. Cette machine parallèle a la particularité de fonctionner dans un mode multi-réseaux (Cf. section 2.2.2). Le déploiement de cet environnement est effectué par le lanceur de PM_2 nommé "Léonie". Ce lanceur utilise de manière séquentielle des protocoles standards de création de processus à distance pour initialiser un ensemble de processus communiquant. Ce principe a déjà été présenté dans les projets précédents et ne sera pas présenté a nouveau dans cette section. Cependant, le mécanisme de déploiement de cet environnement possède une caractéristique qui n'est présente que dans peu d'environnements. Comme PM_2 est un environnement

[2]Cette opération de déploiement utilise le protocole standard "rsh".

multi-réseaux, le mécanisme de déploiement est d'une part responsable de l'établissement d'un réseau de processus et d'autre part de fournir à ces différents processus une description précise de la topologie des réseaux physiques sous-jacents. Pour cela, "Léonie" utilise un langage de description qui permet de définir précisément les différents types de réseaux disponibles sur chaque noeud de la grappe.

3.2 Déploiement efficace d'outils d'exploitation de grappe de grandes tailles

Dans la section précédente, nous avons présenté des outils de déploiement dédié aux applications parallèles. Cependant la problématique d'un déploiement efficace se retrouve dans le domaine des outils d'exploitation de grappe. Actuellement il n'existe pas d'outils génériques et fédérateurs pour exploiter ou administrer une grappe de grande taille. Pour cette raison, les principaux projets [80] [46] [18] de grappe de grande taille (100 $\sim$ 1000 noeuds) ont développé des suites d'outils adaptés à la taille de ces plates-formes. Ces projets ont été des précurseurs sur la problématique du déploiement efficace d'une application sur une grappe de grande taille. Les sections suivantes présentent brièvement ces projets de recherches basés sur des grappes de très grandes tailles et insistent plus particulièrement sur les outils de déploiement qu'ils inclus.

3.2.1 Le projet Storm

Le projet Storm [46] rassemble un ensemble d'outils d'exploitation de grappes pour des architectures de très grandes tailles composées de milliers de noeuds. Ces outils inclus un mécanisme de surveillance système ("monitoring") capable de remonter une vue cohérente de l'état (par exemple l'évaluation de la charge système) de chaque noeud composant la grappe, un mécanisme de réservation de ressources et d'ordonnancement de travaux et un mécanisme de lancement de programme parallèle et de diffusion de fichier.

Ces outils reposent sur une machine parallèle virtuelle implantée spécialement pour ces outils et exploitant les caractéristiques d'un réseau à faible latence de type Quadrics [68][19]. Cette machine parallèle virtuelle est démarrée en même temps que le système d'exploitation de chaque noeud et possède la même durée de vie. Elle utilise de manière intensive les primitives de diffusion et de réduction optimisées fournies par l'environnement matériel de Quadrics. Ces primitives de communication fonctionnent dans un mode non connecté et confèrent une très bonne résistance aux pannes par cet environnement.

Ces outils d'exploitation sont implantés de manière à passer l'échelle sur une architecture composée de millier de noeuds. Les estimations théoriques sur le temps

de lancement d'une application parallèle sur 4096 noeuds, indiquent des résultats de l'ordre d'une dizaine de millisecondes.
Cependant cette implantation de machine parallèle virtuelle efficace est très dépendante du matériel sous-jacent. Seul le réseau Quadrics permet d'atteindre ces performances et une implantation de Storm sur des réseaux de type LAN 802 laissent envisager des résultats nettement moins performants.

Pour conclure, cette approche utilise les caractéristiques et les services de communication du réseau Quadrics de manière à supprimer les problèmes de fiabilité rencontrés par les approches utilisant une machine parallèle virtuelle préétablie. De plus sur des réseaux moins performants (LAN 802) les mécanismes de diffusion efficaces en fonction de la taille des données restent un problème difficile lorsque l'on souhaite effectuer une implantation générique de ces derniers. Pour ces raisons le transfert de cette technologie vers des réseaux moins performants semble difficile.

3.2.2 Yod du projet CPlant

Le projet CPlant ("Computational Plant") [1] est développé dans le laboratoire national Sandia (SNL)[14]. L'objectif de ce projet est de réaliser une grappe composée de 10.000 noeuds interconnectés par un réseau Myrinet. Ce projet inclus une suite complète d'outils permettant l'exploitation de telles plates-formes. Cette suite d'outils se compose d'un système d'évaluation de l'état de chaque noeud composant la grappe. Ce module de surveillance est utilisé conjointement avec un module de réservation de ressource et d'ordonnancement de travaux de manière à effectuer une répartition de la charge de travail équilibrée sur l'ensemble des noeuds de la grappe. Ces modules utilisent un lanceur d'application parallèle nommé Yod pour réaliser le déploiement d'une nouvelle instance de machine parallèle virtuelle propre à cette application.
Ces outils reposent sur une machine parallèle préétablie fournit par l'environnement de programmation PORTAL [28]. Cependant son fonctionnement et sa mise en oeuvre sont différents de ceux présentés dans le chapitre 2.
En effet, la machine parallèle virtuelle dédiée au lancement d'application parallèle fonctionne dans un mode déconnecté. C'est à dire que les différents processus qui la composent ne sont pas connectés avec les autres par des liaisons réseaux permanentes. Le fonctionnement en mode déconnecté garantie une bonne fiabilité car les différentes pannes possibles (réseau, processus de la MPV ou noeud de la grappe) ne sont plus traitées comme la détection d'une panne suivit de son contournement, mais comme l'adaptation à un échec de connexion.
Lorsqu'un ordre de lancement d'application parallèle doit être diffusé, les processus qui disposent des informations nécessaires réalisent une interconnexion pour réaliser cette diffusion de données. Une deuxième différence se situe sur la topologie de l'interconnexion réalisée lors d'une demande d'exécution. Cette interconnexion est hiérarchique, c'est à dire qu'elle utilise un arbre couvrant l'ensemble des noeuds cibles.

Ces spécificités confèrent à cet environnement un fort potentiel de passage à l'échelle dû à l'utilisation d'une topologie réseau hiérarchique et une forte fiabilité dûe à un fonctionnement en mode déconnecté.

3.2.3 Rexec et Gexec

Le projet Rexec [37] fournit un lanceur d'application pour grappe de station de travail. Cet outil ne s'appuie pas sur un environnement de programmation parallèle basé sur une machine parallèle virtuelle. Cependant, son fonctionnement repose sur un ensemble de processus communiquant et permet d'exécuter un programme en parallèle sur tous les noeuds d'une grappe. Le lancement de l'exécution d'un programme sur N noeuds de la grappe est réalisé par trois types de processus préétablis.
La figure 3.3 détaille les interactions de ces différents processus pour réaliser le lancement d'un programme **P**. Dans un premier temps le processus de contrôle (noté Rexec) permettant d'interagir avec les différents processus du programme **P** est démarré par l'utilisateur. Ce processus interroge un des processus de surveillance (noté Vexecd) pour obtenir la liste des noeuds correspondant le mieux aux critères de l'utilisateur, dans cet exemple le taux d'occupation (charge système) le plus faible. Une fois que cette liste de noeuds lui est fournie, il contacte directement les processus responsables de la création de nouveaux processus (noté Rexecd). Ces derniers créent un nouveau processus chargé d'exécuter le programme **P**.

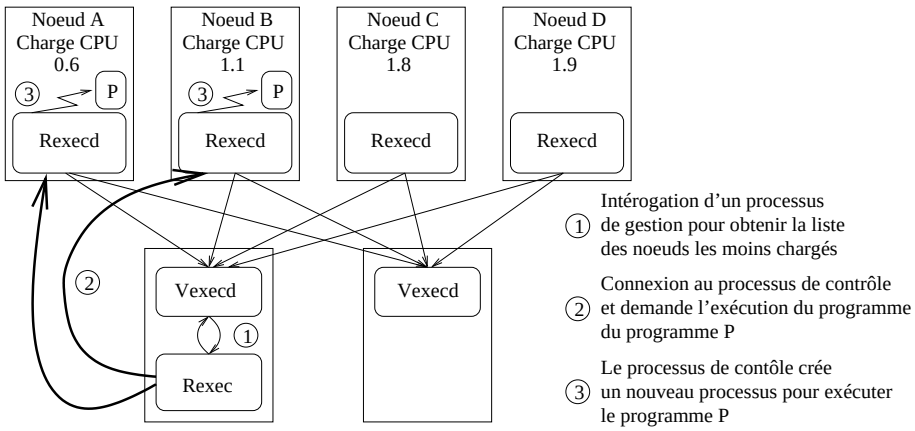


FIG. 3.3 – Fonctionnement et organisation du lanceur parallèle Rexec

Le fonctionnement de cet outil est proche d'une machine parallèle virtuelle fonctionnant en mode déconnecté. Les connections aux différents processus communiquant sont effectuées lors de la diffusion d'une demande de création de processus. Pour diminuer le coût de l'utilisation de ces protocoles, Rexec a fait le choix d'implanter un nouveau protocole plus léger, basé sur une couche de communication sécurisée (ssl [82]), fournissant le même service. Cette approche permet à cet outil d'obtenir de bonne performance sur des grappes de taille moyenne. En effet même si

les requêtes de création de processus distant sont effectuées en concurrence, le coût de cette demande d'exécution parallèle reste linéaire en fonction du nombre de noeuds.

Pour cette raison, les concepteurs de ce projet sont à l'origine de l'outil Gexec [36][3] dont le fonctionnement de base est similaire à celui de Rexec. Cependant, les demandes de connexion ne sont plus effectuées en totalité par la commande Gexec (équivalente à Rexec). En effet dans cette nouvelle version, l'accent est porté sur le passage à l'échelle de cet outil. Pour cela, les différentes requêtes de création de processus sous effectuées récursivement. La topologie de l'interconnexion de la machine virtuelle parallèle repose sur un arbre N-aire équilibré (par défaut binaire). De cette manière le coût de la diffusion de l'ordre de création de nouveaux processus n'est plus linéaire mais logarithmique en fonction du nombre de noeuds.

3.2.4 SCUPP basé sur MPICH

L'exploitation et surtout l'administration de grappes de grandes tailles nécessite la parallélisation de commandes simples classiques. Le projet SCUPP [61] suit cette voie en proposant des versions parallèles des principales commandes d'administration Unix. L'implantation de ces commandes repose entièrement sur l'environnement de programmation parallèle Mpich. Une première version de ces outils a été implantée au dessus du mécanisme linéaire de création de machine parallèle virtuelle. Cependant cette voie a rapidement été abandonné au profit d'une version utilisant une machine parallèle virtuelle dédiée au lancement. Ce changement rapide de solution vient du fait que les commandes d'administration parallèle doivent passer à l'échelle d'une grappe de grande taille, mais doivent également fournir une certaine capacité d'interaction avec l'utilisateur. Contrairement au domaine du calcul scientifique le temps de calcul des processus démarrés peut être largement négligeable par rapport au temps de lancement de ces commandes.

3.2.5 C3 du projet OSCAR

Le projet C3 [29][12]("Cluster Command & Control") fournit également un ensemble de commandes Unix parallèles. Ces commandes parallèles d'administration sont implantées au dessus des protocoles d'exécution distante standard (rsh et ssh). La parallélisation de ces commandes est souvent simple puisqu'il suffit de réaliser l'exécution de la commande Unix existante sur l'ensemble des noeuds cibles. Cette diffusion de demande d'exécution est effectuée par le processus de contrôle (démarré par l'utilisateur) qui est également responsable de rediriger et de multiplexer les entrées / sorties des commandes séquentielles Unix vers l'utilisateur.
Afin de paralléliser l'envoi des requêtes d'exécution distante via les protocoles standards (rsh et ssh), la diffusion de ces requêtes est effectuée en concurrence de manière à obtenir un recouvrement entre les parties d'exécution locale et distante de ces protocoles.

La figure 3.4 représente un exemple d'implantation de la commande parallèle "cexec" qui déclenche l'exécution d'une commande Unix (passée en paramètre) sur plusieurs noeuds d'une grappe. La diffusion concurrente des requêtes d'exécution distante est réalisée en utilisant un processus client du protocole standard par noeud cible. Cette méthode permet de recouvrir la partie de l'exécution du protocole traitée par le noeud distant. Les différentes entrées / sorties effectuées par les commandes Unix sont acheminées par des liaisons réseaux (TCP) établies par les processus (client et parfois serveur) qui implantent le protocole d'exécution distante (rsh ou ssh).

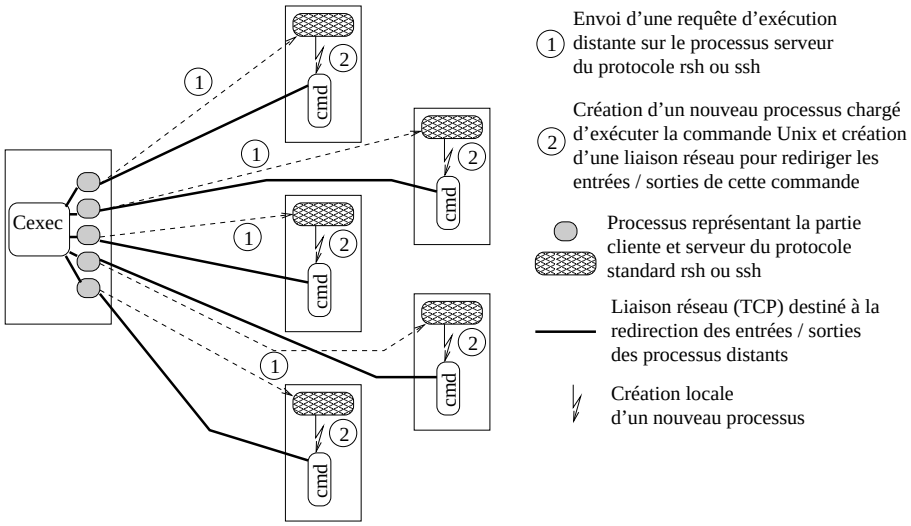


FIG. 3.4 – Fonctionnement détaillé de la commande "cexec". Cette commande déclenche l'exécution d'une commande Unix sur un ensemble de noeuds de la grappe et redirige les entrées/sorties de chaque instance de ces commandes vers l'utilisateur.

Ce projet utilise une parallélisation de type "exécution concurrente" pour augmenter les performances de la diffusion de la requête de création de processus distant. Cependant, ce mécanisme ne traite en parallèle que la partie du protocole qui s'exécute sur la machine distante. La partie cliente du protocole n'est pas recouvrable, ce qui implique un coût de lancement linéaire en fonction du nombre de noeuds et du coût de la partie cliente d'un protocole standard (rsh ou ssh).

3.2.6 Ksix

Ksix [21] constitue un environnement de programmation dédié à l'implantation d'outils d'exploitation et d'administration pour grappes de grandes tailles. Il fait partie du projet SCE ("Scalable Cluster Environnement") [81][80] développé par l'Université de Bangkok en Thaïlande. Les différents outils développés au sein de ce projet repose sur un environnement de programmation, Ksix, qui implante une machine parallèle virtuelle fournissant l'ensemble des services de bases nécessaires à ces outils. Ces

fonctionnalités de base se composent essentiellement de primitives de communication multi-port permettant de réaliser des opérations de diffusion et d'agglomération efficaces. Les liaisons réseaux utilisées pour l'implantation de cette machine parallèle virtuelle fonctionnent en mode déconnecté (protocole internet UDP/IP). La topologie du réseau virtuel des processus composant cette machine parallèle virtuelle suit une topologie en arbre permettant d'obtenir de bonne performance sur les opérations de diffusion et d'agglomération de données.
La mise en oeuvre de cet environnement est effectuée au démarrage de chaque noeud de la grappe. La topologie en arbre du réseau virtuel de processus garantie un coût de diffusion de requête (par exemple de création de processus) logarithmique en fonction du nombre de noeuds.

3.3 Bilan sur l'existant

Ce tour d'horizon sur les projets existants qui sont confrontés à la problématique du déploiement d'un ensemble de processus communiquant permet d'effectuer un bilan assez complet sur les différentes méthodes disponibles pour fournir un mécanisme de déploiement efficace pouvant passer à l'échelle sur de grappes de grandes tailles.
Les projets comme Mpich [49], MPILAM [6], PM_2 [65], Rexec [37], C3 [29] utilisent un mécanisme de mise en oeuvre centralisé. Ce principe implique qu'un processus va effectuer la diffusion d'une requête de création de processus distants sur tous les noeuds cibles de la grappe. Les différentes optimisations possibles de ce principe (diffusion des requêtes concurrente comme C3 ou utilisation de protocole dédié et à faible coût) permettent d'obtenir de bonnes performances seulement sur des grappes de petites tailles (dizaine de noeuds). Le coût linéaire en fonction du nombre de noeuds de ces approches interdit leur passage à l'échelle sur des plates-formes composées de centaines ou de milliers de noeuds.
Pour contourner ce problème d'efficacité de nombreux projets (Mpich [49], MPILAM [6], Storm [46], Yod [27], KSIX [21]) utilisent une machine parallèle virtuelle préétablie permettant de réaliser le déploiement efficace d'un ensemble de processus communiquant. Dans ce cas, les caractéristiques d'un déploiement d'un nouvel ensemble de processus communiquant sont communes aux caractéristiques du mécanisme de diffusion de ces environnements. En effet la création d'un ensemble de processus se réduit à la diffusion d'une requête de création de processus sur les noeuds concernés. Cette technique offre de bonnes performances, mais il est cependant important d'utiliser une topologie d'interconnexion de cette machine parallèle virtuelle adéquate. En effet, le projet Mpich utilise une topologie en forme d'anneau qui induit un coût de diffusion linéaire en fonction du nombre de noeuds.

L'utilisation d'une machine parallèle virtuelle pour réaliser le déploiement contribue à factoriser le coût d'une approche basée sur l'utilisation de protocole standard de l'internet (rsh ou ssh) pour réaliser la création des différents processus. Cependant, cette factorisation pour être réellement efficace oblige ces environnements à fournir

une très bonne fiabilité. Une panne de noeud physique, de réseau ou de l'application parallèle qui s'exécute ne doit ni perturber, ni dégrader les performances de ces environnements.

Dans l'existant, ce problème de fiabilité est contourné de deux méthodes différentes. La première consiste à dédier la machine parallèle virtuelle préexistante au lancement (Mpich). Cette solution ne garantie pas une résistance aux pannes physiques de noeuds ou du réseau, mais consiste à isoler l'environnement de lancement de l'environnement d'exécution du ou des programmes parallèles de l'utilisateur. Pour cela la machine parallèle virtuelle est entièrement dupliquée (processus et interconnexion virtuelle), ainsi une erreur d'un ou plusieurs processus de l'application parallèle dégrade seulement la MPV attachée à ce programme. Pour cette raison, les environnements KSIX, Storm et Yod utilisent une approche différente. Elle consiste à faire fonctionner un ensemble de processus communiquant en mode déconnecté, soit en utilisant les propriétés de communication de réseau efficace comme Myrinet ou Quadrics, soit en utilisant des protocoles communication en mode déconnecté de l'internet standard (UDP). Ce fonctionnement permet de traiter plus facilement les différentes pannes.
Par extension, il est possible de voir les projets (C3 et Gexec), qui utilisent des protocoles de création de processus distants (dédié ou standard), comme un ensemble de processus communiquant dans un mode déconnecté. En effet, l'implantation de ces protocoles nécessitent l'exécution d'un processus sur chaque noeud de la grappe (démon associé au protocole). Cet ensemble de processus peut être vu comme une machine parallèle virtuelle disposant d'une interconnexion fonctionnant en mode déconnecté. Plus précisément, une connexion est établie à chaque envoi de requête. Ce fonctionnement est théoriquement proche de celui exposer dans le paragraphe ci-dessus.

Outre les problèmes de passage à l'échelle et de performance, le projet PM_2 insiste sur l'importance de pouvoir décrire précisément l'architecture cible sur laquelle un environnement doit être déployé. Ces informations, sur la puissance de chaque noeud ainsi que sur les différents types de réseaux physiques auxquels ils sont connectés, peuvent être exploitées pour initialiser correctement l'environnement de communication. Par extension, il pourrait également être pris en compte pour déterminer quelle méthode de déploiement est la plus efficace pour un certain type de configuration de grappe (par exemple multi-réseaux).

Pour conclure, il apparaît indispensable à un ensemble de processus communiquant dans un mode connecté ou déconnecté de posséder une interconnexion virtuelle hiérarchique. Cette topologie d'interconnexion garantie un coût de diffusion d'une requête de création de processus logarithmique en fonction du nombre de noeuds. Ceci impliquant automatiquement un bon potentiel sur le passage à l'échelle de ces environnements. D'autre part il peut être intéressant d'exploiter en plus un niveau de parallélisme supplémentaire en réalisant une diffusion de cette requête sur un modèle de communication multi-port. Ceci peut être réalisé en exploitant les primitives de com-

munication fournies par un réseau physique, ou en réalisant des appels concurrents aux services standards d'exécution distante comme ssh ou rsh.

3.4 Conclusion

Le développement très rapide de la taille moyenne des architectures de type grappe a conduit les activités de recherches de ce domaine à travailler sur des outils permettant d'exploiter leur puissance de calcul. Depuis que les grappes standards se composent de centaines de noeuds, des efforts de recherche se concentrent sur le développement d'outils de déploiement efficaces. Cette problématique touche le domaine du calcul haute performance pour réaliser le déploiement d'applications de calcul parallèles sur des grappes de très grandes tailles, ainsi que le domaine de l'administration et de la maintenance de ces architectures qui souhaite utiliser des outils efficaces permettant d'atteindre une certaine interactivité.

Dans la suite de ce document, nous présenterons nos travaux sur le déploiement d'application parallèle sur des grappes formées de centaines ou de milliers de noeuds. L'objectif du mécanisme de déploiement présenté par la suite est de permettre la création d'un nouvel ensemble de processus communicants de manière efficace. Pour cela, nous avons choisi d'exploiter le parallélisme et la concurrence de manière systématique.
L'un des objectifs de ces recherches est également de passer d'un modèle où la configuration des machines de la grappe est modifiée pour implémenter les services nécessaires au mécanisme de déploiement, à un modèle ou le mécanisme de déploiement s'adapte à la configuration des différentes machines composants la grappe cible. Pour cela, notre outil de déploiement doit être indépendant des protocoles d'exécution distantes utilisés. Il doit également être capable de s'adapter à une certaine hétérogénéité de configuration des machines cibles, en permettant l'utilisation de plusieurs protocoles d'exécution distante pour effectuer un même déploiement.
Comme cet outil doit s'adapter à la configuration de la plate-forme cible, sa mise en oeuvre (son installation) ne doit nécessiter aucun privilège particulier. Un utilisateur doit pouvoir utiliser cet outil sans droit d'administration particulier.
Comme la plupart des projets existants qui utilisent des protocoles d'exécution distantes, les notions de sécurité intrinsèques aux services fournis ne seront pas traitées par le mécanisme de déploiement. Tous les aspects de sécurité seront reportés sur les différents protocoles utilisés pour réaliser le déploiement.
Notre attention durant le développement de cet outil de déploiement s'est focalisée sur des caractéristiques de généricité, de flexibilité, d'adaptation et de passage à l'échelle.

Deuxième partie

Déploiement efficace d’une machine parallèle virtuelle

Chapitre 4

Outils et méthodes pour le déploiement parallèle

Comme nous l'avons vu, la plupart des couches de communication pour le calcul parallèle existantes sont assimilables à des "machines parallèles virtuelles". Leur initialisation consiste à démarrer un processus sur chaque noeud cible de la grappe et à créer une interconnexion logique initiale leur permettant de communiquer.
Dans ce chapitre nous concentrons notre discours sur le déploiement de l'exécution et l'interconnexion initiale d'un programme parallèle sur une grappe de grande taille. Le bilan sur l'état de l'art de cette problématique propose deux niveaux de parallélisation. Le premier consiste à effectuer des appels concurrents d'exécution distante. Le deuxième propose d'utiliser une diffusion hiérarchique (récursive) de la requête de création de processus.

La première section présente les outils de base qui permettent de créer un nouvel ensemble de processus communicants. L'étude des protocoles standards de création de processus distant nous permettra de dégager une abstraction de ces protocoles. La deuxième section présente l'utilisation de ces connecteurs pour réaliser un déploiement séquentiel. Son coût est linéaire du nombre de noeuds. La troisième section présente les deux types de parallélisation permettant de réaliser le déploiement avec un coût logarithmique du nombre de noeuds.

4.1 Protocoles standard d'exécution distante

Pour démarrer un programme sur machine distante, il existe des protocoles standards de type client / serveur. Ces protocoles sont surtout utilisés dans le domaine de l'administration et de l'exploitation des systèmes répartis sur un réseau. Ils permettent de déclencher l'exécution d'un programme sur une machine distante en redirigeant ses entrées sorties vers la machine cliente. Les plus utilisés sont **rsh** [76] et **ssh** [41]. Ils sont disponibles sur la plupart des systèmes d'exploitation (Unix,Windows,MacOs).
Du point de vue de l'utilisateur ces différents protocoles fournissent les mêmes fonctionnalités. Elles consistent en : (i) la création d'un processus sur la machine distante,

(ii) l'établissement d'une liaison TCP permettant d'interagir avec le processus distant et (iii) la mise en place sur la partie cliente d'un processus de contrôle (redirection des entrées / sorties et des signaux de contrôle du processus distant). L'implantation de ces protocoles [75] se présente sous la forme d'un processus serveur (démon) et d'un processus client (console de contrôle). Les implantations de ces protocoles fournissent généralement une partie cliente disponible sous la forme d'une bibliothèque de programmation. Un programme peut alors utiliser une primitive cliente d'un protocole et demander la création d'un processus distant. Cette primitive retourne généralement une liaison TCP permettant d'interagir et de contrôler le processus distant. Dans ce cas, le processus client de contrôle n'existe pas et ce traitement est à la charge du programme utilisant une primitive d'exécution distante.
Dans la suite de ce document, une partie cliente sous forme d'un processus de contrôle sera nommée client externe (externe à un programme qui l'utilise), et une partie cliente disponible sous la forme d'un primitive fourni au travers d'une bibliothèque sera nommée client interne (interne au programme qui l'utilise). La figure 4.1 illustre le fonctionnement générique de ces protocoles lors de l'utilisation d'un client interne par un programme (par exemple un lanceur d'application) et d'un client externe par l'utilisateur.

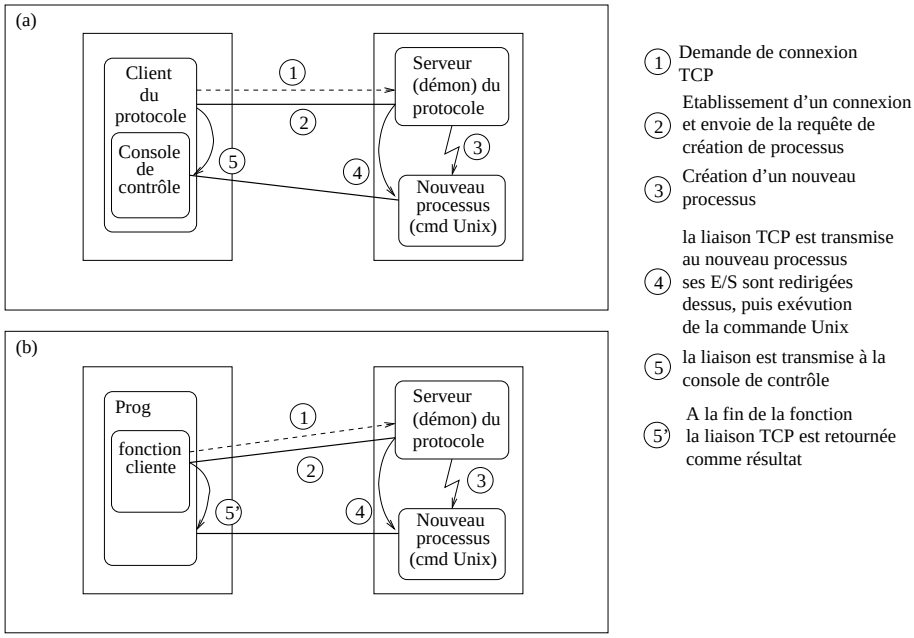


FIG. 4.1 – Fonctionnement générique d'un protocole de création de processus distant, (a) en utilisant un client externe ou (b) un client interne

Les protocoles standards rsh et ssh diffèrent selon le niveau de contrôle et d'authentification effectué préalablement au lancement du programme sur la machine distante. Avant de détailler le fonctionnement de ces protocoles nous allons effectuer un bref rappel sur la gestion de la sécurité sur une machine possédant un système d'exploitation multi-utilisateurs.

4.1.1 Authentification et sécurité

Dans les systèmes actuels plusieurs personnes (utilisateurs) peuvent travailler simultanément sur une même machine. Pour garantir l'intégrité des données et des programmes de chaque utilisateur, ces derniers possèdent des droits d'accès uniquement sur leurs ressources. L'utilisateur λ ne peut pas accéder aux données d'un utilisateur α. Pour assurer cette sécurité, toutes les ressources d'un système d'exploitation sont étiquetées avec le nom de leur propriétaire. Ainsi chaque utilisateur se voit attribuer un espace de stockage et d'exécution qui lui est propre : le compte utilisateur. Pour qu'un utilisateur ne puisse pas usurper l'identité d'un autre, les accès aux comptes sont protégés par mot de passe. Lorsqu'un utilisateur souhaite utiliser une machine multi-utilisateurs, il doit s'identifier en fournissant son nom et un mot de passe.
Cette identification doit être effectuée lors d'une demande distante d'exécution de programme pour assurer la sécurité du système distant relativement aux accès d'un autre système.

Il existe deux niveaux d'authentification : un niveau machine (la machine A a le droit de communiquer avec la machine B) et un niveau utilisateur (l'utilisateur λ a le droit d'exécuter un programme sur la machine B en tant qu'utilisateur $\lambda^{'}$). Dans un protocole sécurisé, les deux parties du protocole (client et serveur) identifient de manière sûre leur interlocuteur par le moyen classique de "nom d'utilisateur" (login) et mot de passe (password).

Dans les deux parties suivantes nous étudierons le fonctionnement des protocoles **rsh** et **ssh**. **rsh** effectue une authentification peu sécurisée mais rapide et **ssh** est considéré comme le protocole plus sûr actuellement.

4.1.2 Le protocole "rsh/rshd"

L'authentification de la machine cliente par la machine serveur est simplement effectuée en vérifiant le couple <adresse réseau, nom de machine> fourni par le client avec le couple enregistré sur le serveur ou fourni par le mécanisme de nommage DNS (Domaine Name Service). Ceci permet d'éviter qu'une machine usurpe l'identité d'une autre.

L'authentification de l'utilisateur est ensuite effectuée de manière standard avec demande du mot de passe correspondant. Ce mot de passe est alors envoyé sur le réseau à la partie distante sans être crypté. Ce mot de passe peut alors être intercepté par toutes personnes connectées sur le même réseau. Pour contourner cette faille de sécurité, il est possible de réaliser une authentification sans envoi de mot de passe. La machine serveur possède une liste de machines associée à une liste d'utilisateurs autorisés à se connecter sans donner de mot de passe.

Ce mécanisme de liste évite de faire circuler des mots de passe en clair sur le réseau. Ce protocole est entièrement fondé sur une relation de confiance entre la partie cliente et la partie serveur. Dans le cas où la partie cliente ment sur l'identité distante de l'utilisateur, elle peut facilement se connecter à une machine en envoyant une demande de connexion de la part d'un couple machine / utilisateur valide sur cette cible. Pour limiter ce problème la partie serveur vérifie que la partie cliente s'exécute avec des droits privilégiés. Ceci permet de limiter les possibilités de connexion non autorisées à une situation où la machine cliente a été compromise[1].

Comme nous pouvons le constater, cette solution n'est valide que dans un milieu clos (sécurisé) ou toutes les machines se font "confiance". L'intérêt de ce protocole est d'être très léger et peu coûteux en temps de connexion distante. Le client se contente d'envoyer ses informations et d'attendre la réponse du serveur. L'authentification du coté serveur est tout aussi simple puisqu'elle consiste à vérifier les informations reçues avec celles enregistrées localement. Le temps de communication est lui aussi très faible car il suffit de transmettre le nom d'utilisateur, le nom de la machine cliente et le nom du programme à exécuter.

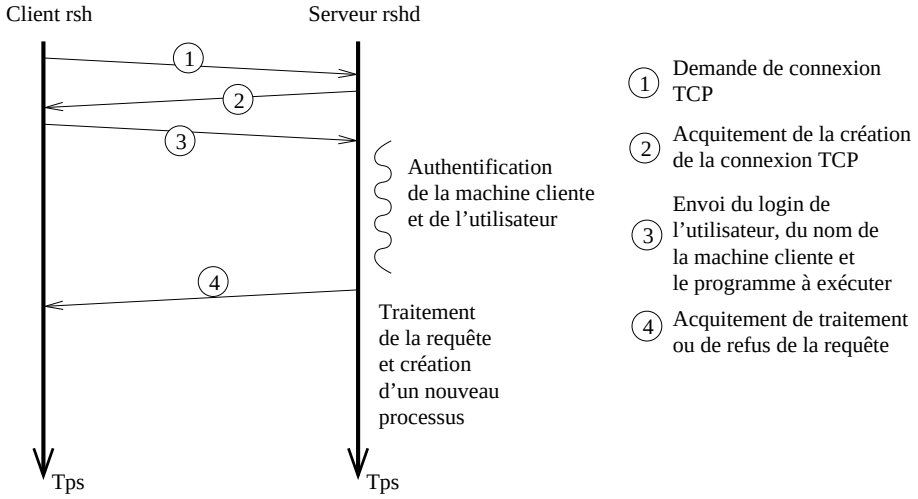


FIG. 4.2 – Graphe temporel de l'envoi d'une requête d'exécution distante en utilisant le protocole **rsh**

Le schéma 4.2 représente le déroulement temporel d'une session rsh. Il inclut : la demande de connexion TCP, l'envoi des données nécessaires au protocole (login, nom de machine et programme), l'authentification réalisée par la partie serveur et l'envoi d'un acquittement (traitement ou refus de la requête).

Le protocole **rsh/rshd** propose un service d'exécution distante simple et rapide. En contrepartie le niveau de sécurité fournit par ce protocole reste faible. Son utilisation est donc essentiellement réservée à un cadre de confiance mutuelle entre deux machines. Dans le cas d'une grappe, on considère que les authentifications nécessaires

[1]Un utilisateur a réussi à passer en mode "root"

ont été réalisées lors de l'accès à des noeuds de la grappe depuis une machine externe. Par la suite, à l'intérieur de cette grappe il est suffisant de se servir de protocole d'exécution distante sécurisé. C'est dans ce cadre d'utilisation que le protocole **rsh** reste très utilisé.

4.1.3 Le protocole "ssh/sshd/ssh-agent"

Ssh est un protocole d'exécution distante fortement sécurisé. Il a été créé pour permettre l'exportation de services critiques sur des réseaux largement ouverts (internet). Dans ce protocole la notion de "confiance" entre deux machines ou deux utilisateurs n'existe pas. L'authentification des machines et des utilisateurs est basée sur des algorithmes de chiffrement symétrique et asymétrique dont la présentation permettra de mieux appréhender les surcoûts de ce protocole, ainsi que leurs localisations.

4.1.3.1 Principe de fonctionnement du protocole de chiffrement de ssh

Le protocole ssh permet d'établir une liaison réseau cryptée entre deux tiers. L'encryptage des données qui circule sur le réseau garantit la confidentialité des communications entre les deux individus. Dans le cas de la mise à disposition d'un service d'exécution distante ces algorithmes sont utilisés pour une authentification de l'utilisateur qui souhaite exécuter une commande sur la machine cible. Le protocole **ssh** utilise deux types d'algorithmes de chiffrement pour garantir la confidentialité des messages échangés et pour limiter le surcoût du cryptage des données.

Algorithme symétrique Les algorithmes symétriques sont aussi nommés algorithmes à clé secrète. Ils utilisent une seule clé pour réaliser les opérations de chiffrement et de déchiffrement. Cette clé est dite secrète car ce protocole de chiffrement repose sur le fait que seuls l'expéditeur et le destinataire connaissent cette clé. L'expéditeur chiffre ses données avec cette clé et le destinataire les déchiffre en utilisant cette même clé. Il existe divers algorithmes de chiffrement symétriques [41], le plus répandu reste DES ("Data Encryption Standard" 1974) introduit par IBM. Le chiffrement y est très rapide. La gestion de ces clés reste un problème. Comme une clé doit être connue par le destinataire et l'expéditeur, ils doivent se l'échanger de manière sûre.

Algorithme asymétrique Les algorithmes asymétriques ont été introduit pour résoudre le problème de l'échange de la clé secrète des algorithmes symétriques. Ces algorithmes utilisent une clé de chiffrement différente de celle de déchiffrement. Ces clés fonctionnent par paire et il est impossible de calculer l'une des clés à partir de l'autre. Ce couple de clés est utilisé sous la forme d'une clé secrète utilisée pour déchiffrer un message, tandis que la clé publique est utilisée pour chiffrer un message.

La clé publique est diffusée à tous ceux qui souhaite la connaître. De cette manière si deux personnes A et B souhaitent s'échanger des messages chiffrés, A chiffre son message avec la clé publique de B qui peut la décrypter avec sa clé secrète, et réciproquement. Ce mécanisme de chiffrement est très coûteux et alourdit considérablement le temps d'un échange de message entre deux personnes. Pour cette raison **ssh** utilise des clés de session.

Clés de session Pour factoriser le surcoût du chiffrement par clés asymétriques, **ssh** utilise les deux algorithmes cités ci-dessus. Le client et le serveur établissent une première liaison cryptée en utilisant un algorithme asymétrique et coûteux. Cette liaison est ensuite utilisée pour faire circuler une clé secrète dite clé de session (valide que pour une seule session **ssh**) utilisant l'algorithme symétrique. Cette clé symétrique sera utilisée pour chiffrer tous les échanges de données entre les deux entités. L'utilisation de ce type de clé garantit un faible surcoût de chiffrement.

4.1.3.2 description du protocole

La figure 4.3 montre l'utilisation de ces différents algorithmes lors de l'établissement d'une connexion TCP cryptée entre la partie cliente et la partie serveur du protocole ssh. Elle présente également une des techniques disponibles utilisée pour authentifier l'utilisateur distant pour fournir un schéma temporel complet de l'établissement d'une connexion ssh. Dans cette exemple nous ne nous intéressons à la version 1[41] de ce protocole, la version 2 [41] possède un fonctionnement similaire. La création d'une liaison TCP cryptée entre deux tiers se déroule de la manière suivante :

1. Une liaison TCP classique est établie entre le processus client et le processus serveur.
2. La partie cliente et la partie serveur du protocole échange respectivement leur clé publique asymétrique.
3. Le processus serveur génère une clé symétrique de session.
4. Cette clé de session est envoyée au client, pour que la suite des communications soit chiffrée avec cette clé symétrique privée. Cet envoi est réalisé en chiffrant cette clé de session avec la clé publique asymétrique du processus client.
5. Le client envoie au serveur sa requête de création de processus, en précisant le nom de l'utilisateur et le programme demandé.
6. Le serveur doit authentifier cet utilisateur. Pour cela, il réalise un challenge qui consiste à chiffrer des données générées aléatoirement avec la clé publique de l'utilisateur (présente sur son compte local) et les envoyer à la partie cliente. Le challenge est réussi si le client réussi à déchiffrer ces données et à les retourner au serveur.
7. Le client réalise le challenge et déchiffre les données et les renvoie déchiffrées sur le serveur

8. Le serveur vérifie la réussite du challenge en comparant les données envoyées aux données reçues. Il renvoie un acquittement à la partie cliente signifiant le traitement ou le refus de sa requête de création de processus.

Dans cet exemple les deux machines se connectent pour la première fois l'une à l'autre. Si ce n'est pas le cas, l'échange des clés publiques n'est pas réalisé, car chaque partie possède déjà la clé publique de l'autre. Dans ce cas, les deux parties réalisent l'authentification de l'autre en réalisant un challenge du même type que celui effectué pour l'authentification de l'utilisateur énoncé ci-dessus.

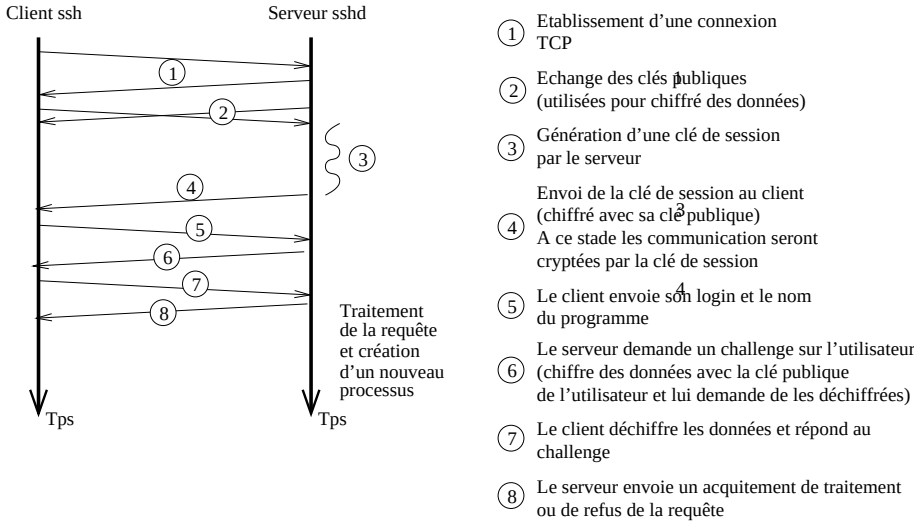


FIG. 4.3 – Graphe temporel de l'envoie d'une requête d'exécution distante en utilisant le protocole **ssh**

La figure 4.3 présente une technique d'authentification de l'utilisateur distant basée sur un challenge en utilisant un couple de clés asymétriques propre à un utilisateur. Cependant, il existe d'autres méthodes plus ou moins sûres et plus ou moins souples pour authentifier l'utilisateur. Dans la première, l'utilisateur s'identifie de manière classique en donnant son nom et son mot de passe. Cette solution est sécurisée car les communications sont cryptées. La deuxième solution combine le principe de clés publiques / privées avec un mot de passe (pass-phrase) propre au protocole **ssh** (différent du système). Lorsque cette solution est retenue, l'utilisateur doit donner ce mot de passe à chaque ouverture de session. Ce mécanisme permet de ne pas faire circuler le mot de passe du système d'exploitation sur le réseau. De plus cela permet de vérifier que le compte de l'utilisateur n'a pas été compromis[2] sur la machine distante. Dans ce cas la clé publique ne peut déchiffrer un message que si elle est associée à ce mot de passe. L'utilisateur doit donc le fournir à chaque authentification par challenge. Pour une utilisation plus souple de cette dernière configuration, il est possible de démarrer un **agent ssh** qui ne demandera qu'une seule fois le mot de passe et le fournira à toute nouvelle demande de connexion

[2]L'utilisateur c'est fait "emprunter" son identité par un tiers qui potentiellement connaît son mot de passe."

distante (appel du client **ssh**). Cet agent peut gérer plusieurs machines distantes (et plusieurs mot de passe) et permet de factoriser les différentes demandes de mot de passe à venir. Dans le cas d'appels récursifs d'exécution distante (A $\rightarrow$ B $\rightarrow$ C) la demande d'autorisation est toujours remontée vers l'agent de la machine initiale (A). Seul l'agent lancé par l'utilisateur connaît tous les mots de passe et les fournit à toutes les parties clientes du protocole qui en ont besoin, même si le client n'est pas directement sur la machine sur laquelle s'exécute l'agent.

En contrepartie d'une très bonne sécurité, l'établissement d'une connexion **ssh** est très coûteuse par l'utilisation d'un algorithme de chiffrement asymétrique et la génération des clés de sessions. De plus, à la différence du protocole **rsh**, les deux parties (cliente et serveur) effectuent l'authentification de l'autre partie. Une fois que ces machines se sont identifiées, la machine serveur authentifie l'utilisateur distant suivant le même principe. Les communications nécessaires à l'établissement de ces différentes authentifications sont relativement importantes. De plus dans le cas de l'utilisation d'un **agent ssh**, l'authentification de l'utilisateur sur les différentes machines cibles est centralisée. Ce protocole paraît donc mal indiqué pour les grappes de calcul qui peuvent être considérées comme des milieux clos et protégés. Cependant, ce protocole est le standard sur toute machine fournissant un service d'exécution distante dans un cadre ouvert. Pour cette raison, il parait indispensable de le considérer, car il permet d'accéder à des noeuds d'une grappe depuis des machines extérieures à la grappe.

4.1.4 Concept de connecteur

Dans cette partie, nous allons proposer une abstraction des protocoles d'exécution distantes. Le but est de définir une brique de base qui nous permettra de réaliser un déploiement efficace indépendamment des protocoles utilisés. Nous appellerons cette abstraction de protocole un **connecteur**. Après avoir défini les fonctionnalités d'un connecteur, nous donnerons un modèle de fonctionnement approprié à la parallélisation du déploiement d'une application parallèle.

4.1.4.1 Fonctionnalités d'un connecteur

Un connecteur correspond à l'abstraction d'un client de protocole d'exécution distante (rsh, ssh, ou autre). Les fonctionnalités de base sont : la création d'un processus distant et l'établissement d'une liaison réseau TCP avec le processus distant. Ces fonctionnalités de base sont simples et permettent d'abstraire tous les protocoles standards existant, ainsi que de nombreux protocoles dédiés à un environnement de type grappe.
La figure 4.4 présente les tâches qui sont à la charge d'un connecteur. Comme l'objectif de cette abstraction est de fournir une brique de base à un outil de déploiement, les connecteurs seront présentés comme un appel de fonction disponible via une bibliothèque de programmation. L'appel d'un connecteur déclenche l'exécution d'un nouveau processus sur une machine distante. Le résultat de cet appel de fonction est

une liaison TCP avec le processus distant.

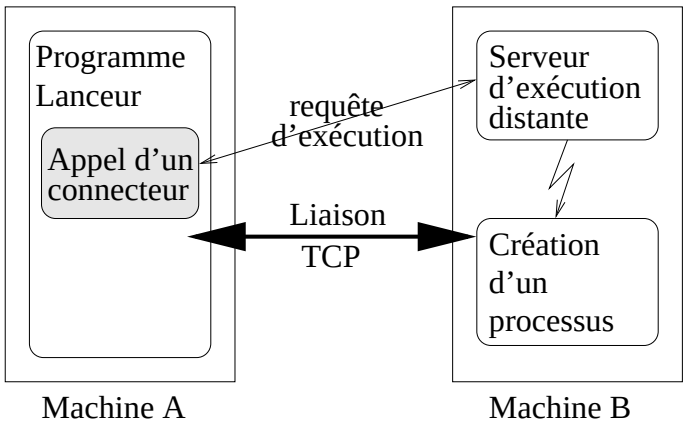


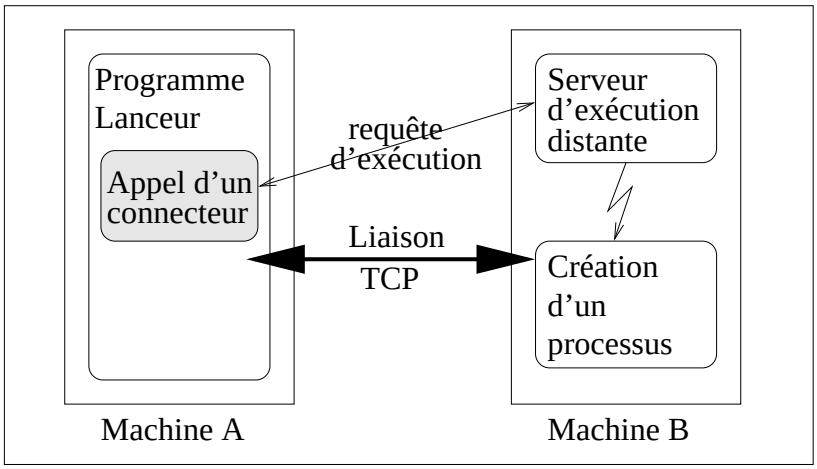
FIG. 4.4 – Fonctionnement d'un connecteur (abstraction de protocole d'exécution distante). Cette figure décrit les tâches qui sont à la charge d'un connecteur.

Tout connecteur doit offrir ces deux fonctionnalités de base. Cette abstraction peut inclure d'autres fonctionnalités. Par exemple, l'envoi de la requête de création de processus peut inclure la diffusion du programme devant être exécuté sur le site distant.

4.1.4.2 Mise en oeuvre d'un connecteur

Les clients disponibles pour les différents protocoles d'exécution distante existants se présentent sous deux formes : un programme (commande Unix) ou une bibliothèque de programmation. Ces deux types de client ont été définit comme des clients de type, respectivement externe et interne.
L'objectif de l'abstraction de connecteur est d'homogénéiser les fonctionnalités et l'utilisation de ces différents protocoles. Pour cela, l'utilisation d'un connecteur se fera toujours en utilisant une fonction de programmation (un appel de connecteur).
La figure 4.5 présente la mise en oeuvre d'un client interne et d'un client externe sous cette forme. Lorsque l'on utilise un client interne le processus qui utilise cette primitive exécute lui même la partie cliente du protocole. Dans le cas d'un client externe, le processus qui utilise l'abstraction d'un connecteur crée localement un autre processus responsable de l'exécution de la partie cliente du protocole associé. Ce processus "client" aura la charge de réaliser l'envoi de la requête de création et l'établissement de la connexion avec le processus distant. La différence entre ces deux méthodes est que le processus "client" créé dans ce dernier cas possède la même durée de vie que le processus distant. En effet, la liaison de contrôle créée par le protocole d'exécution distante est établie entre ces deux processus. Le processus client joue alors le rôle d'un représentant local du processus distant et retransmet toutes les communications du processus "lanceur" au processus distant et réciproquement. Cette différence de fonctionnement est importante en terme de performance. Pour cette raison, les connecteurs représentant des clients externes seront également étiquetés externes, même si les fonctionnalités fournies et leur utilisation restent homogènes.

Abtraction de connecteur utilisant un client interne (bibliothèque)



Abtraction de connecteur utilisant un client externe (commande Unix)

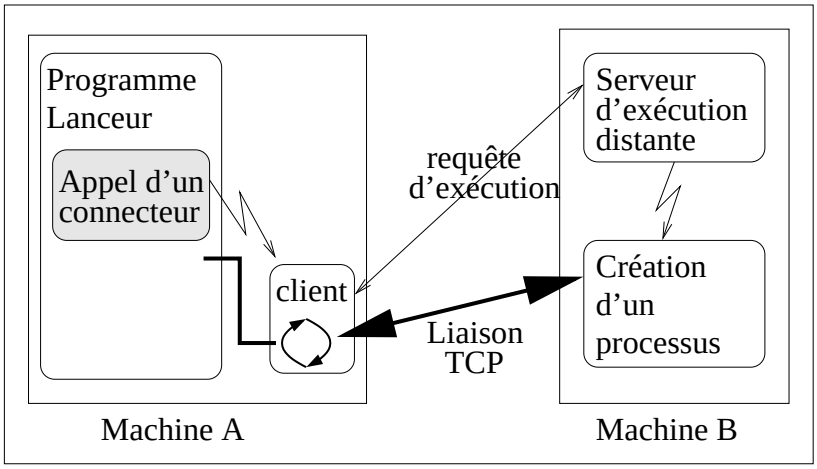


FIG. 4.5 – Mise en oeuvre d'un connecteur en utilisant un client dit "interne" (en haut) et un client dit "externe" (en bas).

Cette approche est a priori plus coûteuse, car elle utilise un nouveau processus pour chaque demande d'exécution distante. Cependant, elle possède l'avantage de pouvoir utiliser les protocoles existants de manière générique en utilisant directement la commande (Unix) cliente d'un protocole comme une "boite noire". Ceci permet une totale indépendance du protocole d'exécution distante utilisé. Il est possible d'utiliser tous les protocoles existants (standards ou propriétaires). Ces différents protocoles doivent fournir les services de base d'un connecteur (la création d'un processus sur une machine distante et la création d'une liaison réseau permettant de le contrôler). Ces fonctionnalités de base représentent un minimum, elles peuvent être étendues de manière à prendre en compte la diffusion des fichiers nécessaires à l'exécution ou la réservation d'un ensemble de noeuds.

4.1.4.3 Modèle de coût

Comme nous l'avons défini, tous les connecteurs correspondent à des protocoles d'exécution distante. Cependant, ces protocoles ont des comportements différents suivant le niveau de sécurité qu'ils fournissent. Pour abstraire ces différences de comportement, nous allons les regrouper en trois parties. Le travail effectué par le client, le travail effectué par le serveur et enfin le taux de communication. Ceci nous permet de dégager un schéma temporel simple sans prendre en compte les différents entrelacements possibles de ces trois parties. Dans ce modèle de coût, le temps total d'un appel d'exécution distante est égal à la somme des trois temps : le temps client, le temps réseau et le temps serveur.

Cette abstraction simple rend le modèle de connecteur extensible et très flexible. Il permet de prendre en compte un protocole d'exécution distante qui permet de diffuser le programme que l'on souhaite exécuter du site client vers le site serveur. Dans ce cas l'indice de communication sera prépondérant. Ces trois indices de comportement nous permettrons de réaliser une parallélisation performante du déploiement. Ces indices permettent d'estimer la quantité de calcul distant qui peut être recouverte durant l'établissement de la connexion distante.

4.2 Déploiement d'une application parallèle

L'objectif de ce travail est de déployer efficacement une application parallèle sur un grand nombre de machines. Le déploiement implique de démarrer un programme sur tous les noeuds et d'établir un réseau virtuel d'interconnexion. Ce réseau sera par la suite utilisé pour contrôler l'application parallèle et pour fournir un premier moyen de communication entre les noeuds.
A ce stade, nous aurons initialisé une machine parallèle virtuelle sur laquelle il sera alors possible de démarrer le calcul de l'application. Cette étape devra être complétée par l'établissement d'une interconnexion virtuelle permettant une communication efficace de l'environnement de programmation parallèle (Cf. chapitre 2). Le réseau de contrôle sera utilisé pour construire cette interconnexion.
Dans cette section nous insisterons sur le déploiement initial, c'est à dire le lancement d'un ensemble de processus et l'établissement d'un réseau virtuel initial. Nous étudierons tout d'abord une approche séquentielle de déploiement, couramment utilisée par les projets existants et les limitations de cette technique. Dans un deuxième temps, nous exposerons les principes de parallélisation permettant de contourner ces limitations. Nous montrerons comment il est possible de réaliser un déploiement logarithmique en fonction du nombre de noeuds par des connecteurs standards.

4.2.1 Approche séquentielle

Comme nous l'avons dit dans les parties précédentes, le déploiement consiste à démarrer un processus sur chaque noeud cible de la grappe et à réaliser l'inter-

connexion de ces derniers. La plupart des machines parallèles virtuelles existantes réalisent le déploiement en deux phases. Les processus sont démarrés sur l'ensemble de noeuds à l'aide de protocoles d'exécution distante comme **ssh** ou **rsh**. Ensuite ces processus s'interconnectent de manière à fournir un réseau virtuel de communication. Pour réaliser cette interconnexion, les informations nécessaires sont centralisées sur le noeud racine effectuant le lancement. Elles sont ensuite diffusées sur tous les noeuds pour permettre la création de nouvelles connexions pour créer un maillage complet du réseau virtuel. Nous ne nous intéresserons qu'à l'étape initiale du déploiement.

4.2.1.1 Création du réseau de contrôle

La solution directe adoptée par de nombreuses machines virtuelles existantes consiste à démarrer tous les processus distants de manière séquentielle. Dans un premier temps nous allons considérer cette approche simple pour étudier comment il est possible d'utiliser les connecteurs pour réaliser en une seule opération le démarrage d'un processus sur une machine distante et l'interconnexion de ce dernier au processus distant qui a demandé son exécution.

Tous les connecteurs créent une liaison logique (connexion TCP) entre la partie cliente et la partie distante du protocole. Le réseau de contrôle ainsi construit est un réseau "étoile" centré sur le processus lanceur (Cf. figure 4.6).

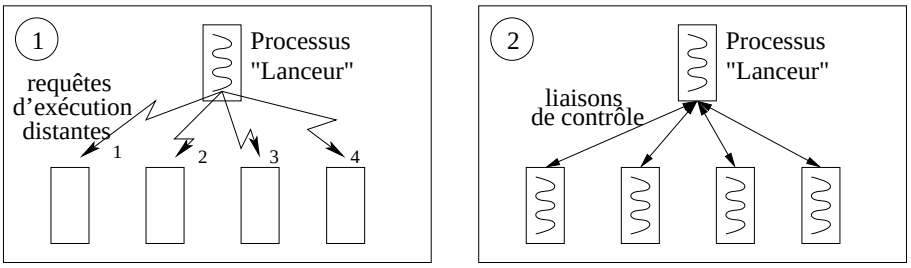


FIG. 4.6 – Présentation de la topologie du réseau de contrôle obtenu à partir d'un lancement séquentiel et centralisé.

La durée d'exécution du déploiement initial est la somme des temps de lancement et d'établissement des connexions élémentaires, c'est à dire des coûts de connexion de connecteurs utilisés.

4.2.1.2 Incidence des connecteurs

Les connecteurs constituent la brique de base d'un outil de déploiement, aussi le coût de ce déploiement est directement lié au coût unitaire d'un connecteur. La section précédente montre que le coût d'un lancement séquentiel centralisé est la

somme des temps unitaires de chaque connecteur utilisé. Outre les différences de coût dûes aux différences des protocoles représentés par les connecteurs. Il est important de distinguer les surcoûts ajoutés à ces protocoles. Comme nous l'avons dit dans la section 4.1.4.2, l'utilisation d'un connecteur "externe" est beaucoup plus coûteuse qu'un connecteur "interne".

Lorsque le client d'un protocole d'exécution distante se présente sous la forme d'un connecteur externe. Il nécessite la création d'un nouveau processus pour exécuter la partie cliente du protocole associé (Cf. section 4.1.4.2). Cela signifie que la création d'une machine virtuelle composée de 20 noeuds nécessite 21 processus (20 clients et le processus lanceur) sur le noeud racine. Ceci limite fortement les capacités de passage à l'échelle d'une telle approche.

4.2.2 Bilan

Cette approche simplifiée effectue un lancement séquentiel centralisé avec un coût linéaire en fonction du nombre de noeuds cibles. De plus, lors de l'utilisation de connecteurs externes ce coût peut "écrouler" le système du fait d'un trop grand nombre de processus. A chaque démarrage d'un processus distant, un nouveau processus local est créé, conduisant à une surcharge du noeud racine, ce qui augmente le temps de chaque lancement distant suivant. Cette approche passe très mal à l'échelle. En plus de son coût, la centralisation de tous les lancements distants sur un même noeud oblige ce dernier à gérer autant de liaisons logiques que de noeuds distants.

Ceci induit un coût des communications de contrôle linéaire du nombre de noeuds. Une autre limitation provient du fait que de nombreux systèmes d'exploitations limitent les ressources offertes à un utilisateur pour ne pas "écrouler" les performances globales du système. Ces limites bornent donc le passage à l'échelle de cette approche au nombre de connexions distantes autorisées (1024 sous Linux) ou, dans le cas de connecteurs externes, au nombre de processus maximum pour un utilisateur (254 sous Linux).
Pour ces raisons, la section suivante présente des solutions de parallélisation du déploiement ayant un coût inférieur et contournant les limites induites par le système d'exploitation.

4.3 Parallélisation du déploiement

Le coût de la création séquentielle d'une machine parallèle virtuelle est très important car il est linéaire du coût des protocoles d'exécution distante standards. Idéalement le lancement en parallèle du programme sur tous les noeuds ramènerait le coût total au coût unitaire d'un connecteur. Cependant la modélisation d'un connecteur par un coût client, un coût réseau et un coût serveur montre qu'un tel résultat

ne pourrait être atteint qu'avec une exécution totalement parallèle des parties clientes et des parties réseaux des connecteurs utilisés[3]. Une telle parallélisation exigerait autant de processeurs que de noeuds destinataires ainsi que le même nombre de liaisons réseaux physiques. Il nous faut donc proposer une méthode qui puisse exploiter au mieux les possibilités de parallélisation offertes par la machine initiatrice et l'interconnexion physique.

La première méthode repose sur la multiprogrammation des parties clientes. La seconde repose sur l'exploitation des liaisons indépendantes du réseau ou d'une capacité d'acheminement supérieure au débit nécessaire à un connecteur.

4.3.1 Multiprogrammation des parties clientes

Dans la section 4.1.4, nous définissons un connecteur comme l'abstraction d'un protocole d'exécution distante. Cette abstraction distingue trois phases dans l'exécution d'un appel d'exécution distante : une phase de traitement sur la partie cliente, une phase de communication et une phase de traitement sur la partie serveur.

Ces phases peuvent s'entrelacer selon le protocole utilisé. La multiprogrammation des parties clientes permet de déclencher les exécutions des parties serveur au plus vite en exploitant les possibilités de parallélisme côté client : plusieurs processeurs, plusieurs ports réseau et recouvrement des latences de transfert entre client et serveur. La conséquence est une parallélisation plus ou moins importante (suivant le fonctionnement d'un protocole) des exécutions des parties serveurs.

La parallélisation locale par multiprogrammation d'une séquence d'appels distants peut être implantée par des mécanismes plus ou moins coûteux qui induisent des taux de concurrence différents. La section 4.1.4.2 montre que les clients des protocoles d'exécution distante (connecteur) peuvent être disponibles sous deux formes : sous la forme d'un exécutable (connecteur externe) ou sous la forme d'une fonction de programmation (connecteur interne). Suivant le type de connecteur (interne ou externe) et la complexité du protocole, il existe trois possibilités d'implémentation :
Dans le cas d'un connecteur externe, chaque appel d'exécution distante est réalisé par un nouveau processus. Ce mécanisme est coûteux en ressources systèmes car il utilise un processus par connexion distante mais autorise une très forte flexibilité.
Sur le même principe, il est possible d'utiliser des processus légers pour paralléliser plusieurs instances de parties clientes de connecteurs internes.
La solution la plus efficace est la parallélisation de la partie cliente du protocole. Cette solution très performante nécessite le développement d'un client parallèle pour un protocole donné. Cette parallélisation consiste à traiter de façon asynchrone les différentes requêtes distantes de manière à recouvrir le temps d'authentification distante par l'envoi de nouvelles requêtes. Elle utilise le nombre de processus légers ("threads")

[3]Les parties serveurs s'exécutent sur chacun des noeuds distants en parallèle

correspondant au nombre de processeurs disponibles et non au nombre de noeuds distants. Cependant, cette approche reste lourde en développement et peu flexible. Pour ces raisons, elle n'est utilisable que pour des protocoles simples comme **rsh**.

4.3.2 Arbre de lancement

Le déploiement de la machine parallèle virtuelle consiste à démarrer un ensemble de processus et à réaliser l'interconnexion logique de ces derniers. La section précédente a montré les limites d'un déploiement effectué de manière centralisée. Dans ce cas, le mécanisme de déploiement réalise tous les appels d'exécution distante nécessaires au démarrage de l'ensemble de ces processus.
Une solution simple permettant de contourner ces limites consiste à répartir les appels distants à effectuer sur plusieurs noeuds. Le mécanisme de déploiement est distribué et s'effectue de manière récursive en parallèle. L'objectif est de faire participer les noeuds de la machine parallèle virtuelle à son propre déploiement. Le premier noeud composant la machine parallèle virtuelle est lancé par l'utilisateur et constitue la base de la récursion de cet algorithme. A ce stade la machine parallèle virtuelle n'est composée que d'un seul noeud ou processeur virtuel. Ce noeud est alors responsable de poursuivre le déploiement en effectuant des appels d'exécution distante pour ajouter de nouveaux noeuds à la machine parallèle virtuelle. Une fois ces nouveaux noeuds démarrés, la machine parallèle virtuelle possède plusieurs processeurs virtuels capables de poursuivre le déploiement. Cette méthode exploite d'une part le parallélisme entre noeuds et le parallélisme entre liaisons réseaux indépendantes. Ce démarrage récursif des différents noeuds de la machine parallèle virtuelle aboutit à la formation d'un arbre de lancement (appel d'exécution distante) couvrant l'ensemble des machines cibles (cf. figure 4.7).

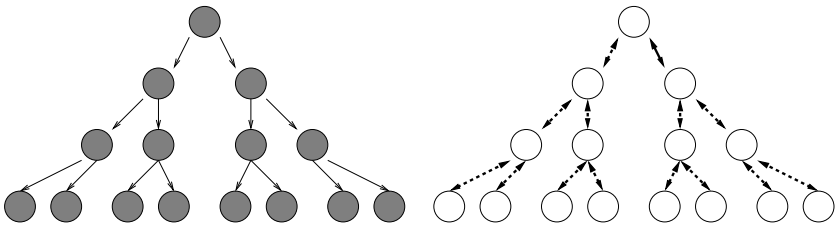


FIG. 4.7 – Distribution des appels d'exécution distante (rsh, ssh, etc...) sur plusieurs noeuds. L'arbre de lancement formé par ces appels distants récursifs est ensuite utilisé comme premier réseau d'interconnexion.

L'efficacité sera d'autant meilleure que l'arbre de lancement sera un arbre couvrant du réseau des liaisons physiques. Dans ce cas le coût du déploiement devient logarithmique en fonction du nombre de noeuds. Lors de l'utilisation d'une topologie de lancement basée sur des arbres équilibrés N-aire, le coût du déploiement est de $\bigcirc(\lambda A log_A(N))$ avec respectivement λ le temps d'un appel d'exécution distante, A représentant l'arité de l'arbre et N le nombre de noeuds cibles.

Cette méthode réduit le nombre de liaisons logiques nécessaires sur chaque et permet ainsi de s'affranchir des limites du système d'exploitation.

4.3.3 Bilan

La parallélisation de la partie cliente a pour objectif d'utiliser toutes les ressources disponibles sur le noeud initiateur (processeur, port réseau) pour exécuter le maximum d'instance de connecteur en recouvrant les latences de transit et les parties serveurs des protocoles correspondants. Le coût de lancement global reste cependant linéaire du nombre de noeuds lancés, bien que le coût de lancement unitaire soit diminué en fonction du taux de recouvrement des patries clientes.
Le lancement parallèle qui consiste à répartir les appels d'exécution distante sur plusieurs noeuds exploite le parallélisme entre les noeuds et entre les liaisons réseaux physiques. Le coût du lancement global peut alors devenir logarithmique du nombre de noeud, multiplié par le coût unitaire d'un connecteur.
Il est clair que l'efficacité relative de ces deux méthodes dépend de l'architecture et des paramètres de coût des connecteurs (protocoles d'exécution distante) utilisés.

4.4 Conclusion

Il est clair qu'une méthode combinant les deux approches (appels distants concurrents et répartition des appels distants) permettrait, en s'adaptant aux propriétés architecturales de la grappe, de réduire au mieux le coût du déploiement d'un réseau virtuel de processus.
Le chapitre suivant est dévolu à l'étude et à la conception d'une telle méthode.

Chapitre 5

Nouvelles approches de déploiement parallèle

Le chapitre précédent propose deux méthodes pour paralléliser le lancement d'une application parallèle. L'une d'elles consiste à distribuer les différents appels d'exécution distante sur plusieurs machines. Ce lancement récursif forme un arbre couvrant l'ensemble des noeuds cibles. Cet arbre peut être considéré comme un graphe de dépendance des différents appels d'exécution distante nécessaires au déploiement. La deuxième méthode proposée suggère d'exploiter une parallélisation locale en effectuant plusieurs appels d'exécution distante en concurrence.
Ce chapitre aborde l'étude du déploiement comme un problème d'ordonnancement. L'objectif est de construire un arbre de lancement ou graphe de dépendance permettant de réaliser un déploiement très efficace sur une grappe de grande taille. Pour cela, la section suivante étudie plus précisément le comportement des appels d'exécution distante concurrents. Cette étude permettra de modéliser les deux méthodes de parallélisation énoncées dans le chapitre précédent en vue de proposer, dans une deuxième section, un ordonnancement optimal dans un contexte homogène. Un contexte homogène implique un comportement des noeuds et des liaisons réseaux constant et identique. La troisième section de ce chapitre apporte une solution d'ordonnancement dans un contexte hétérogène, en proposant une approche dynamique et adaptative. Cette approche exploite un ordonnancement dynamique de type "glouton" par "vol de travail". Une quatrième section présente un langage de description simple permettant de décrire un arbre de lancement complexe. Cette description sera utilisée pour hiérarchiser une grappe hétérogène ou dans la description d'une grille légère (agglomérat de grappes homogènes). Enfin, une dernière section conclut ce chapitre en rappelant les résultats théoriques obtenus.

5.1 Étude et modélisation de connecteurs concurrents

Pour réaliser un ordonnancement efficace des appels d'exécution distante nommé connecteur, cette section revient sur la définition proposée dans 4.1.4. Dans un

premier temps, nous étudierons le comportement d'un connecteur du point de vue du recouvrement de ses phases d'exécution par des exécutions concurrentes. Puis nous proposerons une modélisation du coût d'un connecteur concurrent.

5.1.1 Étude du fonctionnement d'un connecteur

La section 4.1.4 présente le concept de connecteur. Un connecteur représente un protocole d'exécution distante de type client / serveur. La section 4.3.1 introduit la notion d'appel concurrents de connecteur. Elle présente également les différentes techniques utilisées pour exécuter des appels concurrents de connecteur.
L'exécution d'un connecteur se décompose en trois phases : la partie de calcul cliente s'exécute sur le noeud client et une partie serveur s'exécutant sur la machine cible (côté serveur du protocole d'exécution distante). Ces deux phases coopèrent par des communications réseau.
Le coût de la complexité de l'entrelacement de ces trois phases dépendent du protocole associé à un connecteur. Elle dépend également des techniques utilisées pour effectuer ces appels concurrents (processus, threads, appels asynchrones). Les protocoles d'exécution distante standards comme *rsh* ou *ssh* donnent la possibilité d'exécuter un programme présent sur une machine de manière distante. Il est possible d'envisager qu'un connecteur englobe plusieurs opérations de connexion distante : une première pour diffuser le programme à exécuter sur le noeud distant et une deuxième connexion demandant l'exécution de ce programme.

	calcul local	communication	calcul distant
rsh			$\surd$
ssh	$\surd$		$\surd$
diffusion + exécution		$\surd$	

TAB. 5.1 – Récapitulatif du comportement de protocole d'exécution distante standard indiquant les phases ayant un coût prédominant

Le tableau 5.1 présente les phases les plus coûteuses de plusieurs protocoles standard d'exécution distante. Soit tu le temps total d'exécution d'un appel distant pour un connecteur donné : tu = temps local client + temps de communication + temps distant serveur. Pour deux noeuds les calculs distants sont parallélisables. Par contre, la parallélisation des calculs locaux clients dépend du nombre de processeurs disponibles sur le noeud client[1] ou la possibilité de recouvrir le temps de communication d'un appel par le calcul client d'un autre. De même la parallélisation des communications dépend du nombre de liaisons indépendantes, du débit de ces liaisons et du délai de transit (latence réseau).
La section suivante propose un modèle de coût de la parallélisation des appels distants

[1]Un calcul client désigne un connecteur et un noeud client désigne le noeud de la grappe sur lequel un connecteur s'exécute.

concurrents prenant en compte de façon transparente les caractéristiques architecturales.

5.1.2 Modèle de coût

L'exécution d'un connecteur se décompose en deux parties, une partie "locale" non recouvrable et une partie "non locale" qui peut être recouverte par l'exécution d'une autre instance du même connecteur. L'entrelacement de ces deux parties peut être important suivant le protocole d'exécution distante considéré. Il existe deux scénarii possibles : soit l'exécution concurrente de connecteurs forme un pipeline, soit elle conduit à un entrelacement fin où tous les connecteurs concurrents terminent en un temps $T = tu + \alpha(N)$. Dans ce dernier cas, $\alpha(N)$ représente le surcoût de la concurrence de N appels distants. Dans le cas d'un modèle de type pipeline le temps d'exécution d'un connecteur tu se décompose sous la forme $tu = t + \delta$, où t représente le temps à partir duquel il est possible de démarrer un nouvel appel distant (la partie fixe) et δ le temps de l'exécution du protocole sur la partie distante (la partie recouvrable). Le temps total de N appels concurrents est alors $T = t(N) + \delta$, où La figure 5.1 résume ces différents scénarii de comportements sous la forme de diagrammes de Gant.

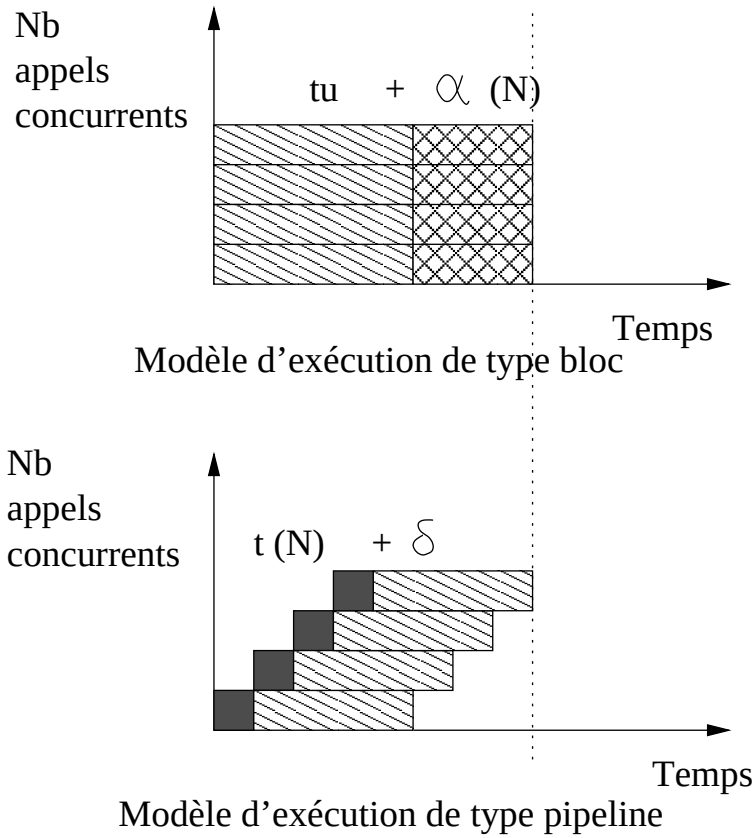


FIG. 5.1 – Comparaison d'un modèle d'entrelacement fin (par blocs) et d'un modèle de type pipeline

Pour choisir parmi ces deux scénari de comportement possible et ainsi choisir un modèle de coût adapté à un comportement réel, la section suivante propose de déterminer expérimentalement le comportement des protocoles rsh et ssh.

5.1.2.1 Caractérisation expérimentale de connecteurs concurrents

L'objectif des expérimentations suivantes est de conclure sur le type de modèle le plus adapté sur le comportement réel des appels concurrents. Pour cela, une première expérimentation consiste à créer un pipeline artificiel en introduisant un retard tr entre chaque création de nouvelle instance d'un connecteur. L'évolution du temps total T en fonction du retard tr, nous indiquera soit que le comportement réel du début de l'exécution des instances de connecteurs suit un modèle de type pipeline où $T = \delta + t(N)$ (dans le cas où il n'y pas de dégradation) soit un modèle où $T = tu + \alpha(N)$.

Les courbes 5.2 exposent les résultats obtenus sur l'exécution concurrente de 97 instances d'un même connecteur (respectivement associé au protocole rsh et ssh). Ces expériences ont été réalisées sur une grappe de 100 machines Pentium III cadencées à 733 MHz interconnectées par un réseau commuté éthernet 100 Mb/s. Ces courbes représentent le temps d'exécution total (T) des 97 connecteurs concurrents en fonction d'un retard tr en millisecondes. Ce temps tr représente la partie fixe du protocole dans un contexte de pipeline Cf. fig5.1. Ces résultats ont été mesurés avec les protocoles rsh et ssh, les appels concurrents ont été implantés en utilisant un nouveau processus pour chaque connecteur.

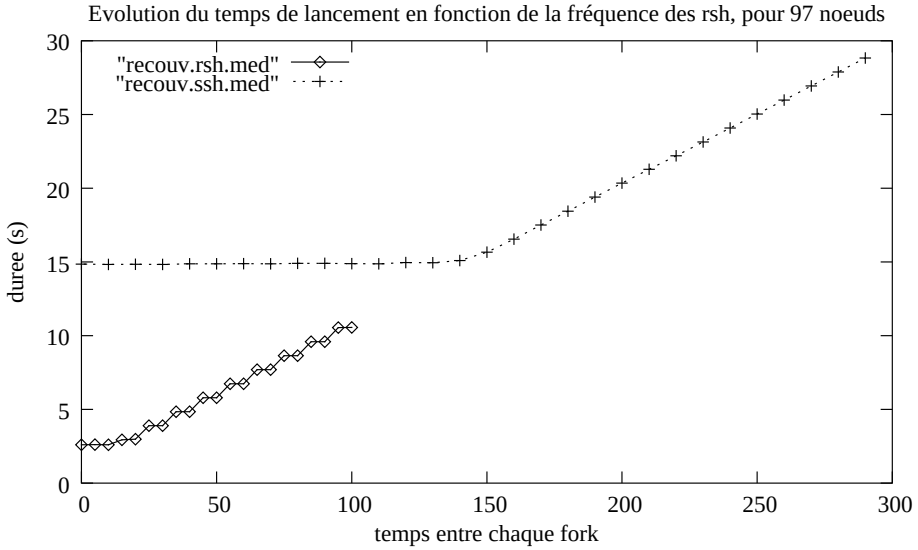


FIG. 5.2 – Expérimentation sur le comportement d'appels d'exécution distante concurrents avec rsh et ssh

Ces premiers résultats montrent que l'ordonnanceur du système d'exploitation (Linux 2.4.20) réalise un ordonnancement de type pipeline sur les différents processus

client. L'ordonnancement réalisé est équivalent à un pipeline "artificiel" composé de tâches de temps unitaire $tu = t + \delta$ dans lequel t représente le temps de la partie fixe du protocole. Ceci confirme donc l'hypothèse suivant laquelle le début de l'exécution des instances concurrentes d'un connecteur forme un "escalier" représentatif d'un modèle d'exécution de type pipeline.
Le seuil à partir duquel l'introduction d'un retard tr trop important provoque l'augmentation du temps total T, représente une estimation du temps t de la partie non recouvrable de ce connecteur. Pour le protocole ssh, respectivement rsh, les expérimentations évaluent ce temps t à respectivement 140 ms pour ssh et 10 ms pour rsh. Le temps d'exécution d'un appel d'exécution distante pour ces deux protocoles est respectivement 330ms pour ssh et 85ms pour rsh.
La figure 5.3 résume le comportement observé pour le protocole ssh. Ces pre-

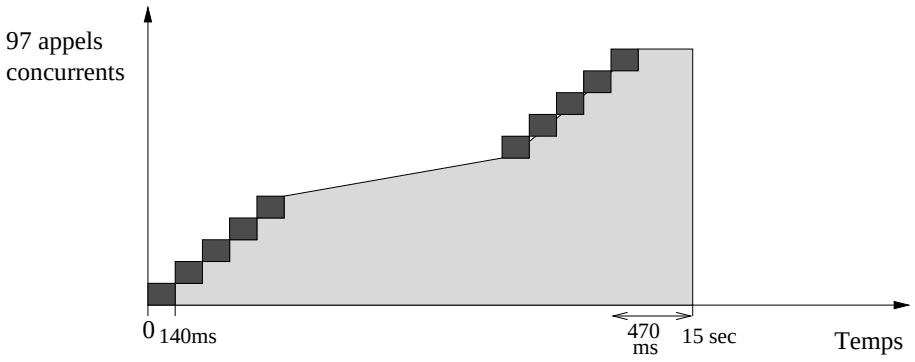


FIG. 5.3 – Schématisation des résultats obtenus pour le lancement de 97 ssh concurrents en fonction de la fréquence de démarrage

mières mesures permettent de réaliser la première partie de l'encadrement sur le comportement de l'exécution concurrente de plusieurs instances d'un connecteur. Elles montrent que le début de l'exécution de ces instances forment un "escalier" qui conforte l'hypothèse d'un comportement pipeliné.
L'ordonnancement réalisé par le système donne un temps d'exécution égale à un modèle pipeline où un délai de 140 ms est introduit entre chaque création de nouveau processus. Ceci valide la décomposition d'un connecteur de la forme $tu = t + \delta$. Cette expérience valide également une première partie du modèle de type pipeline. L'exécution des différents appels de connecteurs sur un modèle de type pipeline donne les mêmes résultats que l'ordonnancement du système. Au delà de 140 ms, le temps total d'exécution augmente de manière linéaire puisque l'on ajoute un temps supplémentaire entre la création de deux processus à l'intérieur du pipeline.

Les expérimentations suivantes ont pour objectif de compléter l'encadrement sur le comportement de l'exécution concurrente d'instances d'un connecteur. Pour cela, les courbes 5.4 donnent la date de terminaison de chaque appel d'exécution distante avec le protocole ssh et rsh, pour une valeur de tr nul ou les valeurs t mesurées pour les protocoles donnés (Cf. les résultats expérimentaux des courbes 5.2).

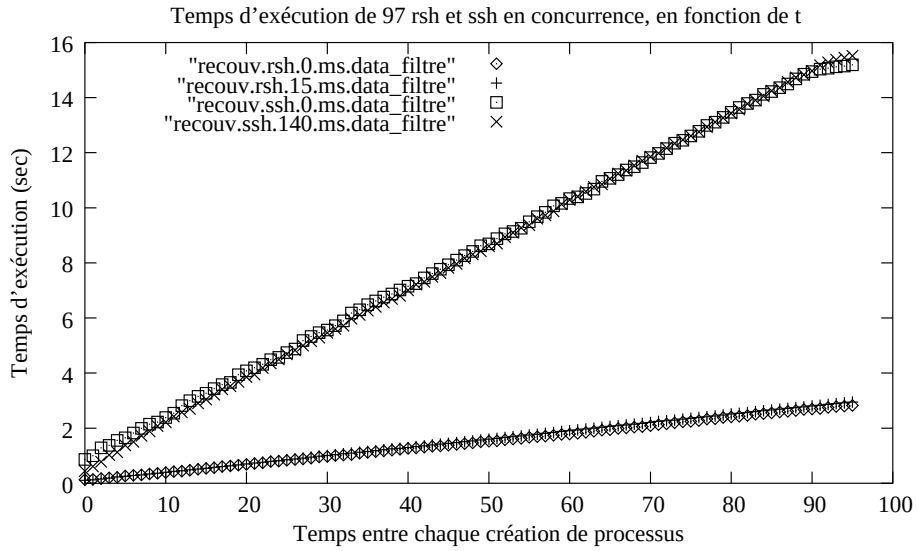


FIG. 5.4 – Date de terminaison de chaque appel d'exécution distante lors de l'exécution concurrente de 97 connecteurs de type rsh et ssh

Ces mesures montrent que le phénomène de pipeline, observé au début de l'exécution, se retrouve à la terminaison de chaque appel d'exécution distante. Ceci est vrai dans le cas où le pipeline est généré par l'ordonnanceur du système (tr nul) ou généré systématiquement par un délai d'attente ($tr = 140ms$ pour ssh et $tr = 15ms$ pour rsh). L'aspect linéaire de ces résultats confirme également que le temps de chaque appel d'exécution distante reste constant malgré le fort taux de concurrence.

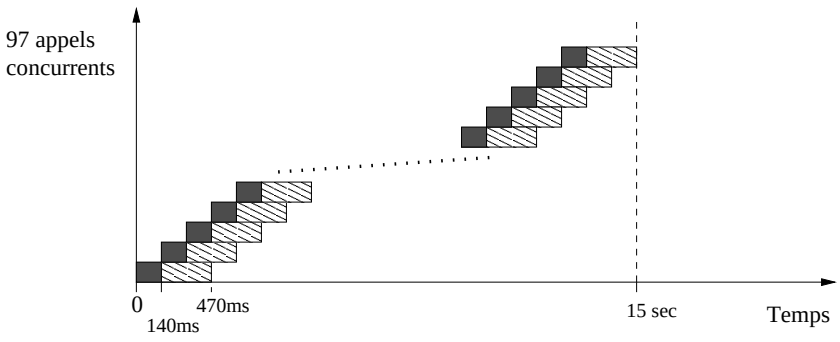


FIG. 5.5 – Schématisation des résultats obtenus pour le lancement de 97 ssh concurrents en fonction de la fréquence de démarrage

Le schéma 5.5 récapitule les résultats exposés ci-dessus et complète l'encadrement commencé dans le schéma 5.3. Le temps d'exécution d'un connecteur se décompose sous la forme $tu = t + \delta$, avec t le temps fixe (partie locale du protocole) formant le pipeline. Ces expérimentations ont également permis de mesurer le temps t de la partie non recouvrable des protocoles rsh et ssh.

5.1.3 Taux de concurrence

Cette section propose une modélisation de l'exécution concurrente de plusieurs instances d'un même connecteur. Le modèle proposé est de type pipeline. La validation expérimentale montre qu'un appel d'exécution distante peut se décomposer en une partie locale t et une partie recouvrable δ. La partie t séquentielle induit un fonctionnement en pipeline des appels distants concurrents. Ce fonctionnement en pipeline peut être obtenu de façon explicite par un retard des appels distants ou implicite par la multiprogrammation du système. Le pipeline implicite (réalisé par le système) possède des limites.
Les mesures montrent que jusqu'à une centaine d'appels concurrents le pipeline se forme correctement. Cependant, la plupart des systèmes d'exploitation limitent le nombre de processus pouvant être exécutés simultanément par un utilisateur, afin d'éviter une surcharge du système qui dégraderait significativement les performances. L'utilisation d'un pipeline explicite (ajout d'une attente de $t$ms) permet de contourner cette limitation. Le nombre de processus actifs simultanément correspond au nombre d'étages du pipeline en régime permanent. Le nombre d'étages est tu/t et correspond au taux de concurrence d'un connecteur.

5.1.4 Bilan

La modélisation d'un connecteur en un temps total d'exécution tu et un temps recouvrable t a permis de modéliser l'exécution concurrente de plusieurs instances d'un même connecteur comme un pipeline dont le nombre d'étages (taux de concurrence) est calculable à partir de mesures expérimentales.
La section suivante va montrer comment cette modélisation peut être utilisée pour optimiser un déploiement distribué en arbre.

5.2 Déploiement dans un cadre homogène et statique

Cette section introduit une modélisation de la problématique du démarrage des processus composant une machine parallèle virtuelle. Cette modélisation s'appuie sur la définition du comportement d'un appel de connecteur de la section précédente. L'objectif de cette modélisation est de réaliser un ordonnancement efficace des différents appels d'exécution distante nécessaires au déploiement d'un programme parallèle. Dans cette section, on se restreint à un ensemble de noeuds homogènes et statiques. C'est à dire qu'un seul type de connecteur est utilisé avec un coût (ainsi que la décomposition de ce coût en parties recouvrables ou non) identique pour tous les noeuds et un nombre fixe et connu de noeuds. La définition de ce modèle permet par la suite d'utiliser les résultats théoriques obtenus pour la problématique de la diffusion de données dans un réseau logique complètement maillé.

5.2.1 Déploiement parallèle

Le déploiement efficace d'une machine parallèle virtuelle peut s'étudier comme un problème d'ordonnancement de tâches devant s'exécuter sur un ensemble de ressources de calcul (processeurs). Dans le cas d'un déploiement récursif, le chapitre 4 montre que la machine parallèle virtuelle peut utiliser ses processeurs virtuels actifs pour poursuivre son déploiement. A un instant τ, il est alors possible de considérer les appels d'exécution distante restant comme des tâches de calcul devant être traitées par les processeurs virtuels actifs. Une tâche représente un appel d'exécution distante. Le traitement d'une tâche crée un processeur virtuel "actif" capable de traiter des tâches à son tour. Un processeur virtuel ne peut traiter des tâches que si il est dans l'état actif. Toutes les tâches et tous les processeurs virtuels actifs sont homogènes. Les temps d'exécution des tâches sont identiques et se décomposent sous la forme $tu = t + \delta$ (Cf. section 5.1.2). Les processeurs virtuels peuvent donc pipeliner l'exécution de tu/t tâches. Si l'unité de temps utilisée est t, cette modélisation introduit une "latence" notée λ égale au rapport tu/t (nombre d'étages du pipeline ou taux de concurrence du connecteur).
Le déploiement efficace d'une application parallèle sur N noeuds se ramène à un problème d'ordonnancement efficace des communications bipoints pour réaliser une diffusion.

5.2.1.1 Modèle Postal

La description du déploiement du paragraphe précédent est identique au modèle postal introduit par Amotz Bar-Noy et Sholmo Kipnis [24] dans le domaine de la diffusion de message. Le modèle postal considère un ensemble de processeurs interconnectés par un graphe de liaisons bipoints complètement maillé.
Chaque processeur peut envoyer un message de type point à point vers tout autre processeur. Un processeur p peut envoyer simultanément un message au processeur q et recevoir un message du processeur r. Dans ce modèle le terme message désigne une donnée atomique échangée entre deux processeurs. L'unité de temps de ce modèle est égale au temps passé par l'émetteur pour envoyer un message. Une communication prend un temps λ qui inclut le temps d'émission, le temps de transit et de réception d'un message. A l'instant i, le processeur p envoie un message M au processeur q. Le processeur p est occupé par l'envoi de ce message durant l'intervalle $[i, i+1]$ et le processeur q est occupé par la réception de ce message durant l'intervalle $[i+\lambda-1, i+\lambda]$. Cette latence de communication inclut donc les surcoûts logiciels et matériels d'une communication. λ est égale au rapport entre (i) le temps passé entre le début de l'émission et la fin de la réception d'un message et (ii) le temps passé par l'émetteur pour envoyer ce message.

5.2.1.2 Déploiement optimal

Le problème du déploiement optimal d'une application parallèle est identique à celui de la diffusion d'un message unique dans le modèle Postal. Il faut calculer un

ordonnancement permettant de minimiser le temps de déploiement. Si $t_o(i)$ représente le temps au bout duquel le processeur virtuel i devient actif avec l'ordonnancement o. Le temps de déploiement de cet ordonnancement sur P processeurs virtuels est le temps au bout duquel le dernier processeur virtuel (i où $1 \leq i \leq P$) devient actif, soit $t_o = max_{1 \leq i \leq P}(t_o(i))$. L'ordonnancement optimal est donc l'ordonnancement parmi tous les ordonnancements o fournissant une borne inférieure du temps de déploiement, soit $t_{opt} = min_o(t_o)$.

La stratégie d'ordonnancement optimale dans le modèle postal [24] est un ordonnancement de type "au plus tôt". Dès qu'un processeur virtuel devient actif, il initie un appel de connecteur à chaque unité de temps (t dans notre cas). Amotz Bar-Noy et Sholmo Kipnis proposent un algorithme d'ordonnancement basé sur cette stratégie dans le cadre des communications collectives et fournissent la preuve de son optimalité.

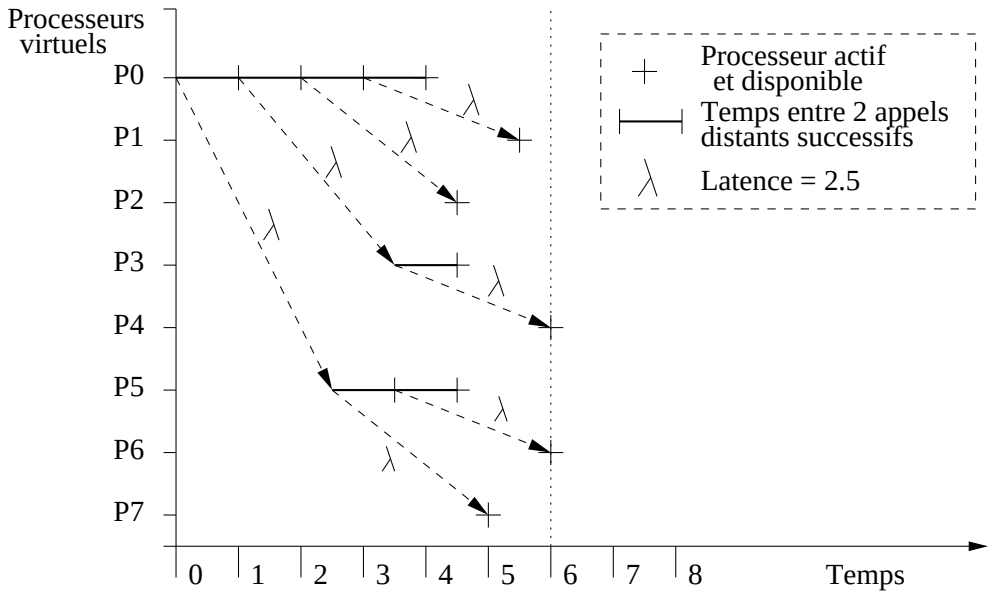


FIG. 5.6 – Diagramme de Gantt de l'ordonnancement optimal des appels de connecteur dans le cadre du modèle postal

La figure 5.6 présente un exemple d'arbre de lancement optimal créé avec un ordonnancement de type "au plus tôt" dans le modèle postal. Ce diagramme de Gantt montre le déroulement temporel des différents appels de connecteur. Cet exemple effectue l'activation de 7 processeurs virtuels avec un connecteur de latence $\lambda = 2.5$. Le temps de ce lancement optimal est égal à la date où tous les processeurs virtuels sont actifs soit 6 unités de temps, soit $6t$.

L'ordonnancement "au plus tôt" donne l'arbre de lancement optimal. La section suivante présente l'algorithme utilisé pour construire cet arbre en fonction des paramètres t et tu qui déterminent λ (taux de concurrence du connecteur).

5.2.2 Construction centralisée d'un ordonnancement optimal selon le modèle postal

Amotz Bar-Noy et Sholmo Kipnis [24] proposent un algorithme optimal pour la diffusion d'une donnée sur un ensemble de noeuds dans le modèle postal. Cet algorithme construit l'arbre de diffusion optimal de manière distribuée. Chaque processeur calcule la liste des processeurs auxquels il doit diffuser le message. Cet algorithme est basé sur une suite de Fibonacci et est présenté dans [24]. Bien que le déploiement se modélise par un modèle postal, les écarts de performance entre un envoi de message et un appel d'exécution distante sont très importants (facteur de 100). Pour cette raison, le calcul réparti de l'ordonnancement n'est pas un point de critique pour les performances du déploiement. De ce fait, nous proposons dans un premier temps un algorithme 1 centralisé construisant entièrement l'arbre de lancement. Cet algorithme implémente un ordonnancement de type "au plus tôt" et permet de calculer le temps théorique optimal de déploiement, ainsi que l'arbre de diffusion associé.

Algorithme 1 Algorithme de construction centralisé de l'arbre de lancement optimal en fonction de t, tu et du nombre de noeuds cibles N.

$tps\_courant = 0$
$nb\_noeuds = 1$
$prochaine\_tache[0] = tu$
$prochain\_tps = tu$
while $(nb\_noeuds < N)$ **do**
  $tps\_courant = prochain_tps$
  $prochain\_tps+ = t$
  for $(i = 0; i < nb\_noeuds; i++)$ **do**
    if $(prochaine\_tache[i] == tps\_courant)$ **and** $(nb\_noeuds < N)$ **then**
      $prochaine\_tache[nb\_noeuds] = tps\_courant + tu$
      $prochaine\_tache[i] = tps\_courant + t$
      $< fils[i] > += nb\_noeuds$
      $nb\_noeuds++$
    end if
    if $prochain\_tps > prochaine\_tache[i]$ **then**
      $prochain\_tps = prochaine\_tache[i]$
    else
      $prochain\_tps = prochain\_tps$
    end if
  end for
end while
$Temps\_total = tps\_courant$

Cet algorithme utilise une unité de temps égale à tu. La variable $tps\_courant$ représente la date courante de chaque itération, nb_noeuds le nombre de noeuds déjà traités. Le tableau $prochaine\_tache$ contient la date à laquelle un noeud i (indice de ce tableau) pourra à nouveau traiter une tâche. La variable $prochain\_tps$ est utilisée

pour calculer la date de la fin d'une tâche sur un noeud, c'est à dire la date à laquelle un noeud pourra traiter une nouvelle tâche. Enfin, le tableau $< fils[i] >$ contient la description de l'arbre de diffusion en fournissant la liste des fils (dans l'arbre) de chaque noeud i.

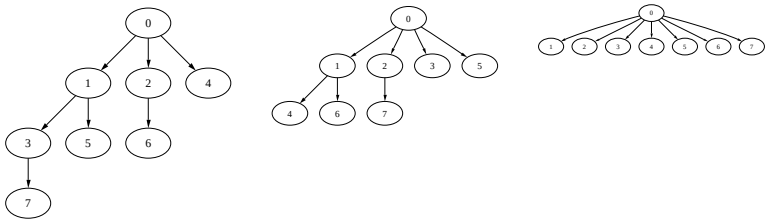


FIG. 5.7 – Exemples d'arbre de lancement optimal pour $\lambda = 1$, $\lambda = 2$ et $\lambda = 10$

La figure 5.7 présente les arbres de lancement générés par les ordonnancements calculés pour différentes valeurs de λ. Suivant le taux de concurrence λ du connecteur utilisé, l'arbre de lancement optimal est soit un arbre binomial (pas de concurrence), soit un arbre de Fibonnaci, soit un arbre plat lorsque la concurrence est très importante.

5.2.2.1 Algorithme de lancement

L'algorithme de lancement récursif est alors le suivant :

if Noeud initiateur **then**
  // appel de l'algorithme 1 pour calculer l'arbre optimal dont il est la racine
  $< fils[racine] >= Calcul\_Arbre(tu, t, < liste\_de\_noeuds >)$
  // lancement des fils du noeud initiateur
  for all i **in** $< fils[racine] >$ **do**
    $Connecteur\_concurrents(i, < fils[i] >)$
  end for
else
  // Chaque noeud reçoit en argument le sous arbre dont il est la racine sous la forme d'une liste $< fils[] >$
  for all i **in** $< fils[racine] >$ **do**
    $Connecteur\_concurrents(i, < fils[i] >)$
  end for
end if

Le noeud initiateur calcule l'arbre couvrant optimal, cet arbre est stocké dans un tableau de liste nommé $< fils[i] >$. Ce tableau contient les sous arbres devant être déployés par chaque fils du noeud initiateur. Un noeud (non initiateur) reçoit en paramètre un tableau $< fils[i] >$ décrivant le sous arbre qu'il doit déployer à son tour et dont il est la racine. Cet algorithme se termine lorsque tous les noeuds ont terminé de traiter (démarrer) la liste de noeuds qui leur a été attribué.

5.2.2.2 Prédiction du temps de déploiement

L'algorithme centralisé (Cf. Algo 1) permet de calculer le temps de déploiement optimal des arbres pour toutes valeurs de $\lambda = tu/t$ et du nombre de noeuds cibles. Les courbes de la figure 5.8 montrent le temps théorique de lancement optimal atteignable pour les connecteurs rsh et ssh en fonction du nombre de noeuds cibles. Ces courbes prévisionnelles ont été établies à partir des mesures expérimentales de tu et t réalisées dans la section 5.1.2.1. Elles représentent une borne inférieure sur les performances d'un déploiement utilisant les protocoles d'exécution distante rsh et ssh. Ces courbes sont des courbes affines (surtout visible pour ssh), la hauteur des marches indique le temps consommé par les différents pipelines (parallélisme de concurrence). La largeur des marches donne le nombre d'appels exécutés en parallèle (parallélisme de distribution). Les valeurs utilisées de t pour chaque protocole dans le modèle $tu = t + \delta$ sont respectivement 140 ms pour ssh et 10 ms pour rsh. Ces valeurs ont été mesurées expérimentalement dans la section 5.1.2.1. Le temps total d'un appel d'exécution distante tu a également été mesuré expérimentalement pour ces deux protocoles, il est respectivement de 330 ms pour ssh et 85 ms pour rsh. Ces valeurs représentent la moyenne sur 100 mesures consécutives avec une erreur de 1ms (ssh) à 5 ms (rsh) sur un interval de confiance à 95, ces mesures ont été réalisées dans les mêmes conditions expérimentales décrites dans la section 5.1.2.1.

Ces résultats théoriques prévisionnels considèrent uniquement le coût des protocoles de création de processus distants (rsh et ssh) et négligent totalement les coûts introduits par l'implantation d'un lanceur suivant cette stratégie. Ils fournissent une première estimation (borne inférieure) du comportement et des performances qu'il est possible d'atteindre en utilisant des protocoles standards réputés comme lourd et coûteux. Ils permettent également de constater que cette stratégie possède un fort potentiel de passage à l'échelle. Les expérimentations du chapitre 7 permettront de comparer les résultats réels d'une implantation de cette stratégie avec ces résultats théoriques.

5.2.3 Bilan

Cette étude a montré qu'il est théoriquement possible d'atteindre de très bonnes performances avec des connecteurs standards (coûteux) comme rsh et ssh. Un déploiement théorique optimal est obtenu par l'utilisation d'un ordonnancement "au plus tôt"conforme au modèle postal de Amotz Bar-Noy et Sholmo Kipnis [24].
Ce résultat repose sur des hypothèses fortes d'homogénéité des connecteurs et sur une connaissance des machines composant la grappe cible. Ceci impose une homogénéité parfaite de la plate-forme d'un point de vue matériel et logiciel. Ce modèle ignore les éventuels points de contention dûs à l'utilisation de services centralisés comme un système d'authentification ou un système de fichiers. De plus, l'ordonnancement réalisé est effectué a priori et ne prend pas en compte les possibilités d'indisponibilité des machines (pannes, arrêt). Si une machine ne répond pas, tout le sous-arbre pré-calculé qu'elle devait lancer sera perdu.

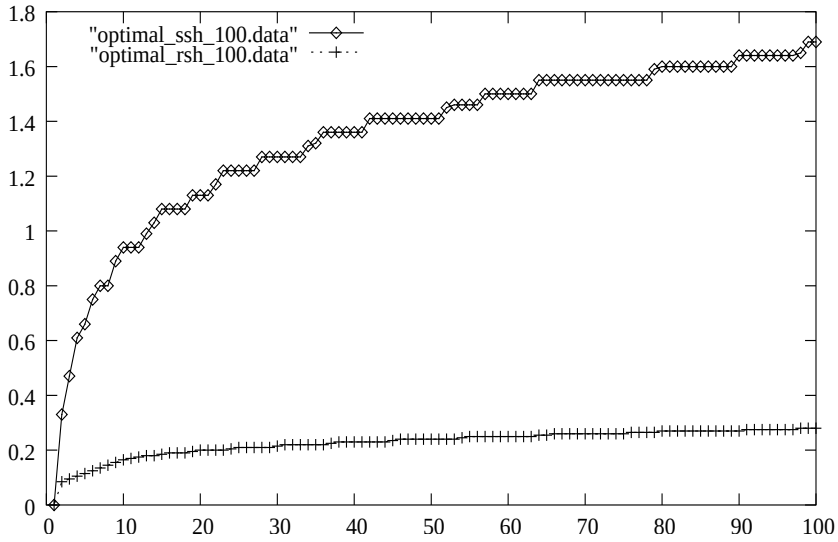


FIG. 5.8 – Résultats théoriques prévisionnels obtenus en utilisant l'algorithme 1 et les mesures expérimentales de t et de tu pour le protocole rsh ($\lambda = 85/10 = 8.5$) et ssh ($\lambda = 330/140 = 2.35$).

Pour éliminer cette contrainte forte sur l'homogénéité et la connaissance a priori de la plate-forme, la section suivante propose une approche permettant de réaliser un déploiement adaptatif sur une plate-forme hétérogène.

5.3 Déploiement dans un cadre hétérogène

Cette section propose une approche basée sur un ordonnancement dynamique des appels de connecteur. L'intérêt d'un ordonnancement dynamique est de pouvoir s'adapter à une plate-forme hétérogène. Dans notre cas, l'hétérogénéité introduit potentiellement différents connecteurs et surtout une différence de coût de chaque appel de connecteurs. Elle rend compte du cas où des machines possèdent des architectures différentes et une interconnexion physique hétérogène. Cette hétérogénéité de comportement peut également être provoquée par le système et les services systèmes utilisés sur la grappe. L'utilisation de services centralisés comme NFS, NIS, DNS ou encore un agent ssh provoque des goulots d'étranglement qui engendrent des comportements irréguliers de la part des connecteurs. De même, dans une exploitation non dédiée, les machines chargées sont moins réactives.

Un ordonnancement statique centralisé comme celui présenté dans la section précédente se prête mal à cette situation. L'objectif est d'utiliser une stratégie d'ordonnancement adaptative permettant d'atteindre des résultats proches de ceux obtenus dans les sections précédentes. Pour cela, cette stratégie devra évaluer dynamiquement les indices caractéristiques des différents connecteurs qu'elle utilise.

La section suivante introduit un ordonnancement "glouton" de "vol de travail" [25][85][32]. Cet algorithme, de type "au plus tôt", doit permettre d'atteindre de bonnes performances. Comme dans le cadre homogène, nous exploiterons le parallélisme de multiprogrammation en utilisant des connecteurs concurrents. La deuxième section présentera une version distribuée et hiérarchique de cet ordonnancement permettant d'exploiter le parallélisme induit par la répartition des appels distants sur plusieurs noeuds. Une troisième section étudiera le coût de cette approche.

5.3.1 Méthode d'ordonnancement de type "glouton"

Une stratégie de type "glouton" permet de maximiser le taux d'occupation d'un processeur. Dans notre cas, cette méthode sera utilisée pour maximiser le taux d'occupation d'un processeur virtuel de la machine parallèle virtuelle. Cette stratégie consiste à fournir une liste de tâches à un processeur. Ce dernier traite ces tâches de manière séquentielle jusqu'à l'épuisement de la liste. Pour exploiter le parallélisme de multi programmation il est possible qu'un processeur puisse traiter plusieurs tâches en concurrence.
L'approche statique dans un cadre homogène, présentée dans la section 5.2, exploite le taux de concurrence d'un type de connecteur tu/t, fonction des paramètres fixes tu et t caractérisant un connecteur (une tâche), pour calculer un ordonnancement optimal du déploiement. Dans une stratégie adaptative, ce taux de concurrence est important pour permettre à un processeur virtuel de traiter un nombre de tâches concurrentes suffisant pour exploiter au mieux la multiprogrammation, sans dégrader les performances du système par une surcharge. Les techniques qui permettent d'évaluer ce taux de concurrence seront décrites dans le chapitre 6.

5.3.2 Stratégie : "glouton" et "vol de travail"

Dans un cadre parfaitement homogène l'ordonnancement optimal des appels d'exécution distante est de type "au plus tôt". Dans ce cadre, il est possible de mesurer expérimentalement les paramètres de cet ordonnancement. Dans un cadre hétérogène cette approche n'est pas réalisable car le comportement d'un appel d'exécution distante peut varier dans le temps et l'espace. Suivant la machine cible le temps d'exécution d'un appel distant peut être différent si cette machine est en cours d'utilisation ou si ces capacités de calcul (puissance CPU) sont différentes. Pour ces raisons il est difficile de réaliser un ordonnancement statique des appels d'exécution distante dans un cadre hétérogène. Pour réaliser un ordonnancement dynamique respectant la contrainte du "au plus tôt", il est possible d'utiliser un ordonnancement dynamique de type "glouton" par "vol de travail" [25][85][32].

Un ordonnancement "glouton" par "vol de travail" correspond à la version distribuée de la stratégie présentée dans la section précédente. Chaque processeur virtuel consomme sa liste de tâches (en exploitant la multiprogrammation). Lorsque cette liste

locale au processeur virtuel est vide, le processeur envoie une requête réseau à un voisin proche (sur le graphe d'interconnexion virtuelle) pour demander une nouvelle liste de tâches. Dans un graphe de connexion réseau de type arbre couvrant, il est possible de distinguer un cas particulier de cette méthode nommée "vol de tâche hiérarchique". Dans ce cas, le noeud auquel cette requête est adressée est systématiquement le père du noeud dans l'arbre émettant cette requête. La requête peut alors circuler dans l'arbre uniquement en direction de la racine.
La taille de la liste retournée par ce voisin détermine la capacité d'adaptation de cette stratégie. Plus le nombre de tâche est faible, plus le déploiement est réactif et mieux il s'adapte mieux à la situation réelle.

5.3.3 Évaluation du coût d'une stratégie par "vol de travail"

Cette stratégie construit dynamiquement un arbre de diffusion couvrant l'ensemble des noeuds. La section précédente insiste sur le fait que le nombre de tâches échangées lors d'une requête de demande de travail est directement lié à la réactivité de cette stratégie et donc à sa capacité d'adaptation.
A l'idéal, une adaptation très fine demande à ce que les tâches soit échangées une par une. Cependant, si cette solution est retenue, le nombre de requête réseau échangé devient linéaire du nombre de noeuds. De plus, comme l'interconnexion de contrôle, utilisée pour ces échanges suit la topologie utilisée pour le lancement (arbre), le coût d'une requête est fonction de la taille du chemin emprunté pour son acheminement. Ceci provoque un coût de communication exponentiel en fonction du nombre de noeuds.
A l'inverse un nombre important de tâche induit un coût de communication réduit et une capacité d'adaptation médiocre.

5.3.4 Bilan

L'utilisation d'une stratégie "glouton" par "vol de travail" permet de réaliser un ordonnancement "au plus tôt", par définition elle maximise le taux d'occupation de chaque processeur virtuel. Cette stratégie doit permettre aux noeuds les plus rapides de consommer plus de travail.
Cependant cette stratégie implique un coût de communication suivant la qualité de l'adaptation voulue. Le compromis entre surcoût de communication et bonne capacité d'adaptation parait difficile.
Une stratégie basée sur une méthode de "vol de travail hiérarchique" permet de limiter le nombre de requêtes nécessaires au déploiement. Cependant, comme elle ne permet pas à une requête de vol d'atteindre tous les noeuds, elle conduit à un niveau moyen d'adaptation.
L'implantation d'une telle stratégie dépend également fortement de l'évaluation locale à chaque processeur virtuel du taux de concurrence des différentes instances de connecteurs. Comme ces connecteurs peuvent être différents et que le temps de réponse peut varier suivant les noeuds cibles, le taux de concurrence devra être évalué en fonction d'indices de charge mesurés dynamiquement sur chaque noeud.

5.3.5 Conclusion

Dans un cadre hétérogène, il semble indispensable d'utiliser une stratégie dynamique basée sur des méthodes adaptatives de type "glouton" par "vol de travail" [25][85][32]. Cette stratégie permet de réaliser un ordonnancement "au plus tôt" et doit permettre d'atteindre des résultats proches de ceux obtenus dans un milieu homogène. Les sections précédentes présentent le fonctionnement de cette stratégie. Son fonctionnement est simple puisqu'il consiste à maximiser le taux d'occupation de chaque processeur virtuel. Cependant, la mise en oeuvre de cette stratégie adaptative semble difficile, car elle repose sur un indice de taux de concurrence et sur un compromis entre coût des communications et adaptativité. Ces deux points critiques semblent difficiles à résoudre dans un cadre général.
Le chapitre 6 présentera tout de même une première implantation de cette stratégie, basée sur une méthode de type "glouton" et "vol de travail hiérarchique". Cette implantation combine cette stratégie avec un mécanisme d'évaluation de performance d'un noeud. Ainsi, le nombre de tâches répondues à un noeud est fonction du nombre de requêtes qu'il a déjà émis et donc déjà traité. Ce mécanisme a pour objectif de fournir un système de cache de tâches sur les noeuds racines d'un sous arbre réactif, de manière à réduire le nombre de requête de vol ainsi que leur distance.

5.4 Bilan

Ce chapitre montre que la problématique du déploiement d'application parallèle avec l'utilisation systématique de la parallélisation (multiprogrammation et hiérarchisation) se résume à un problème d'ordonnancement de communication. Le comportement des appels d'exécution distante est identique à des communications classiques (envoi/réception de messages) de type bipoints. Ce point commun permet d'exploiter les résultats existants sur le problème de la diffusion d'un message ("single message broadcast") sur un réseau complètement maillé.
Ces résultats, par l'introduction d'une stratégie optimale de type "au plus tôt", permettent de prédire une borne inférieure sur les performances d'un outil de déploiement en fonction des protocoles utilisés.
Le comportement d'une communication et d'un appel distant sont identiques, mais la différence de coût de ces deux fonctions de communication est très importante (Rapport de 10 à 100). Cette différence de coût permet d'utiliser un algorithme de "vol de travail, glouton" de manière à implanter une stratégie "au plus tôt". Cette méthode "adaptative" peut être utilisée dans le cadre du déploiement grâce au rapport de coût entre une communication et un appel d'exécution distante.

Chapitre 6

Mise en Oeuvre

Ce chapitre présente la mise en oeuvre des principes de déploiement efficace, tout en essayant de respecter les objectifs que nous nous étions fixés. Les mécanismes de déploiement présentés seront implantés sous la forme d'une bibliothèque de programmation nommée Taktuk[1]. Cet environnement orienté sur le déploiement d'applications, pourra être utilisé par la suite pour réaliser le déploiement d'un environnement de programmation parallèle (Inuktitut, MPI, etc) ou d'outils d'exploitation sur des grappes.

Cette bibliothèque implante les mécanismes de déploiement présentés dans les chapitres précédents. Le service de base de cette bibliothèque consiste à démarrer un ensemble de processus interconnectés par un réseau logique de contrôle. La mise en oeuvre complète d'une machine parallèle virtuelle devra être effectuée au dessus de cet ensemble de processus communicants. Cette étape nécessite des communications pour permettre aux différents processus de s'échanger des adresses réseaux qui leur permettront de mettre en place l'interconnexion applicative de la machine parallèle virtuelle cible (Inuktitut, MPI, PM_2, etc...).
Comme le réseau de contrôle mis en place est de type arborescent, il est nécessaire d'étendre ce service de déploiement avec un service de routage logique permettant aux différents processus de communiquer entre eux. Les environnements parallèles visés (ADI3 [50] de Mpich [49], Inuktitut (Cf. chap 8.1.1) et PM_2 [65]) implantent un protocole de messages actifs [84]. Dans ce protocole de communication, la réception d'un message déclenche l'exécution d'une fonction prenant en paramètre le contenu du message.
Pour permettre d'interagir avec les processus composant l'application parallèle ou les outils d'exploitation, cette bibliothèque doit offrir des services de redirection d'entrées / sorties et de signaux. Ces services seront implantés au dessus du protocole de communication offert par le réseau de contrôle. Ces fonctionnalités pourront facilement être étendues par l'utilisateur de cette bibliothèque, suivant qu'il réalise le déploiement d'une application parallèle ou celui d'un outil d'exploitation.

[1] Taktuk signifie brouillard en langage Inuit (Inuktitut).

Ce chapitre présente la mise en oeuvre de cette bibliothèque nommée Taktuk. Une première section décrira la mise en oeuvre des mécanismes de base du déploiement. Une deuxième section présentera précisément l'implantation du mécanisme de déploiement adaptative introduit dans la section 5.3. Une troisième section insistera sur l'implantation d'un langage de description d'arbre de lancement qui permet de séparer les mécanismes de déploiement, du calcul de l'arbre couvrant adéquate à la configuration. Une quatrième section montera un exemple d'utilisation de cette bibliothèque. Enfin, la dernière section conclura ce chapitre.

6.1 Implantation des mécanismes de base

La bibliothèque de déploiement Taktuk fournit trois principaux services qui se retrouvent en tant que modules dans son architecture. Le module "lanceur" est responsable de l'appel de connecteurs pour déclencher la création distante d'un nouveau processus. Il est chargé de la gestion des différents connecteurs (concurrents) en cours de connexion. Lorsqu'une connexion est établie (processus distant démarré et une liaison de contrôle établie), elle est transmise au module "routeur". Ce module fournit un service de routage logiciel permettant de simuler un réseau complètement maillé au dessus du réseau de contrôle arborescent. Ce module est dédié à l'interconnexion logique et implante le protocole de messages actifs [84]. Enfin, un troisième module est chargé de la redirection des entrées / sorties et des signaux de contrôle entre le processus de contrôle (racine de l'arbre) et les différents processus distants. Ces services de contrôle sont implantés au dessus du protocole de communication.
Comme ces trois modules doivent coopérer et réagir de manière asynchrone, chacun est représenté par un processus léger. Les processus légers "IO" et "routeur" sont respectivement en charge des entrées / sorties et des communications et sont actifs durant toute l'utilisation de la bibliothèque par une application parallèle. Le processus léger de gestion de connexion possède une durée de vie plus courte, qui se limite au lancement des processus distants. Une fois la phase de lancement terminée ce processus léger disparaît car il n'est plus utile au contrôle de l'application parallèle.
La figure 6.1 présente les processus légers actifs pendant et après la phase de déploiement. Les sections suivantes présentent de façon détaillée chacun de ces modules et le fonctionnement du ou des processus légers qui leur sont associés.

6.1.1 Lanceur

Le module de lancement est responsable de la création distante de processus et de la création d'une liaison de contrôle avec ces derniers. Pour cela, il utilise une abstraction logicielle de connecteur semblable à celle décrite dans la section 4.1.4. Dans cette implantation la notion de connecteur ne représente pas seulement un protocole d'exécution distante. Cette abstraction inclut un protocole et une technique de multiprogrammation. Il existe donc plusieurs connecteurs pour un même protocole suivant que l'on utilise une multiprogrammation par processus, par processus légers (threads) ou par un client asynchrone.

Dans cette première version de Taktuk, nous avons implanté plusieurs types de connecteurs exploitant de manières différentes la multiprogrammation de clients "internes" ou "externes". Nous revenons ici sur la présentation de la section 4.1.4.2 de manière plus détaillée.

Une première instance de connecteur a été implantée au dessus du client "interne" du protocole "*rsh/rshd*". Cette instance de connecteur exploite une méthode asynchrone pour recouvrir les temps de calcul locaux et distants du protocole *rsh/rshd*. Pour cela, le client interne de ce protocole a été réimplanté de façon parallèle. Ce nouveau client nommé "rsh asynchrone" établit plusieurs demandes d'autorisation d'exécution sur des sites distants de manière asynchrone et attend tous les acquittements (positifs ou négatifs) des machines distantes. Cette parallélisation est assez coûteuse en temps de développement car il est nécessaire de réimplanter un client parallèle pour chaque protocole. Ce travail est délicat car il peut affecter la fiabilité et la sécurité fournie par un protocole d'exécution distante. De plus cette approche ne fournit pas une bonne généricité par rapport au protocole utilisé. Par contre, le taux de recouvrement est très bon et cette solution consomme peu de ressources système (pas de processus supplémentaires).

Une deuxième instance de connecteur a été implantée avec le client interne du protocole *rsh/rshd*. La parallélisation des appels distants a été réalisée en exploitant la multiprogrammation avec un processus léger (thread) pour chaque appel d'exécution distante. La figure 6.1 expose les différents processus légers qui s'exécutent lors d'un déploiement utilisant ce type de connecteur.
Une fois que les processus distants sont démarrés et que les liaisons réseaux logiques qui les interconnectent sont établies, les différents processus légers nécessaires se terminent. Le schéma du bas de la figure 6.1 montre le nombre de processus légers s'exécutant sur chaque noeud une fois que la séquence de déploiement est terminée.

Le déploiement d'une application parallèle avec des connecteurs internes est très léger et peu coûteux. Cependant cette approche offre une faible généricité par une forte dépendance des protocoles d'exécution distante utilisés. L'ajout d'un nouveau protocole consiste principalement à implanter un nouveau connecteur associé.

Une troisième instance de connecteur permet de contourner le problème de généricité des connecteurs de type "interne". Cette instance utilise directement le programme client d'un protocole. Ce connecteur est dit externe, car il crée un nouveau processus à chaque appel de création de processus distants. Dans ce cas, la partie cliente du protocole est exécutée à l'extérieur de l'application de déploiement. Dans ce cas l'exploitation de la multiprogrammation est évidente et transparente. Cette méthode rend possible l'implantation d'un connecteur générique, pouvant utiliser n'importe quel client de protocole d'exécution distante fournissant les fonctionnalités d'un connecteur (Cf. section 4.1.4.1). Le client d'un protocole est alors utilisé comme une "boîte noire" fournissant un service de création de processus distants. Cependant, cette solution est coûteuse en ressources systèmes car elle consomme un processus

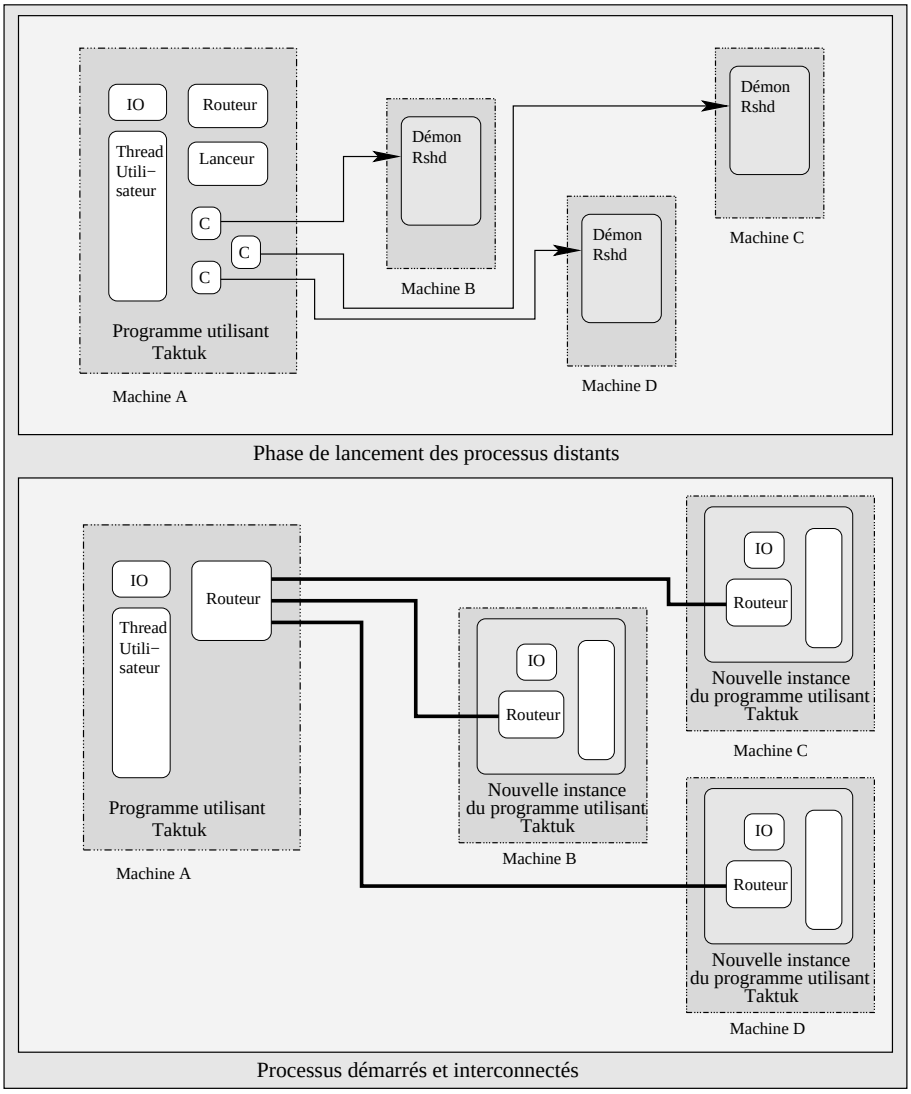


FIG. 6.1 – Illustration des différents processus légers mis en oeuvre lors de l'utilisation d'un connecteur interne.

par appel distant, ce qui limite son utilisation.

Les différents processus (normaux ou légers) mis en oeuvre lors du lancement sont initialisés par le processus léger principal, qui gère l'état d'avancement des différentes demandes d'exécutions distantes. Tous les protocoles réseaux, les protocoles d'exécution distante utilisent un mécanisme d'alarme temporelle ("timeout") pour la détection de panne du noeud distant ou du réseau. Pour permettre une exploitation plus fine des différents protocoles, le "*Lanceur*" implante un mécanisme similaire qui permet de contrôler indépendamment du protocole, le temps de connexion après lequel un noeud distant est considéré comme défaillant. Dans ce cas, l'instance du connecteur concerné est annulée. Lorsque la phase de lancement est terminée, les liaisons réseaux logiques sont enregistrées auprès du processus léger responsable des communications, nommé *Routeur*.

6.1.2 Réseau de contrôle

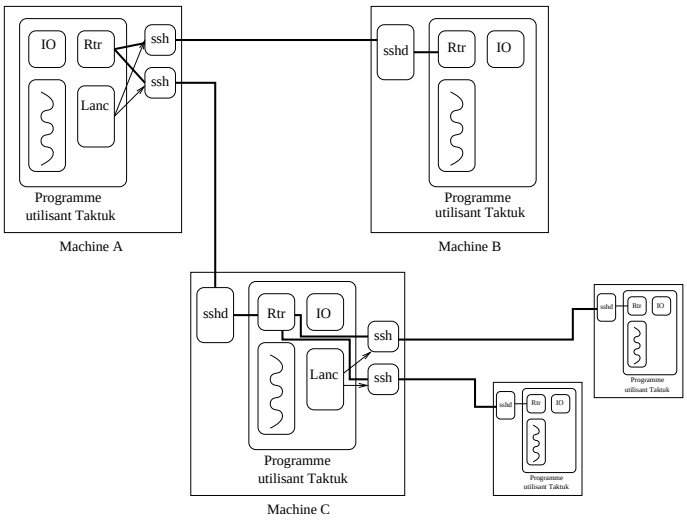


FIG. 6.2 – Illustration des différents processus mis en oeuvre lors de l'utilisation d'un connecteur externe correspondant au protocole ssh

Les figures 6.2 et 6.3 présentent le réseau de contrôle établi par la phase de lancement en utilisant un connecteur "externe", respectivement avec le protocole ssh et rsh. Pour le protocole "*rsh/rshd*", la figure 6.3 insiste sur le fait qu'un nouveau processus est créé pour chaque connexion distante. La durée de vie de ce processus dépasse la phase de lancement, ce processus persiste tant que la connexion est établie. En effet, le client *rsh* est responsable de la liaison réseau logique reliant ce noeud à un noeud distant. A priori, l'impact sera important sur les communications qui seront effectuées par la suite sur cette liaison. Chaque message transitera de l'émetteur au client rsh qui le retransmettra au processus distant. Ce surcoût est d'autant plus important avec le protocole "*ssh/sshd*" illustré dans la figure 6.2. Dans ce cas, les messages échangés entre les machines A et B transitent par le client *ssh* qui s'exécute sur A et par le

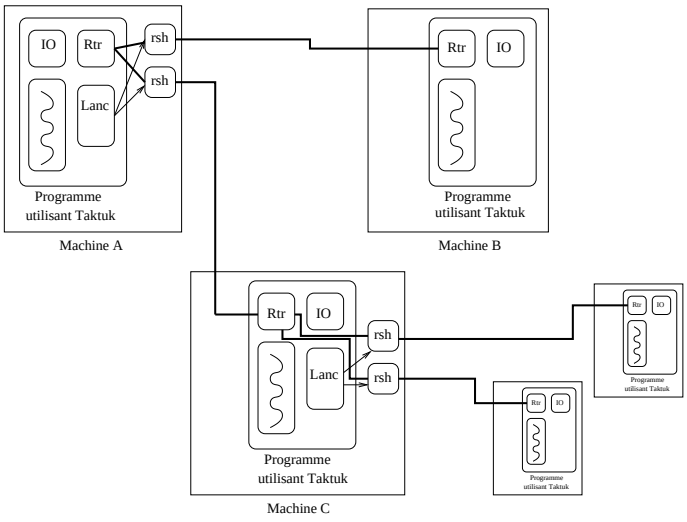


FIG. 6.3 – Illustration des différents processus mis en oeuvre lors de l'utilisation d'un connecteur externe correspondant au protocole rsh

serveur *sshd* qui s'exécute sur la machine B. Pour le protocole "*ssh/sshd*" les parties cliente et serveur restent actives durant toute l'exécution du processus distant de manière à pouvoir crypter par chiffrement toutes les communications (cf.4.1.3). Le chapitre suivant présentera une étude de performances des communications bipoints en fonction des protocoles utilisés dans la section 7.4

Les figures 6.1, 6.2 et 6.3 montrent les différents types d'interconnexion de contrôle obtenus après le lancement. Chaque appel de connecteur déclenche une exécution distante et créé une liaison logique de contrôle permettant d'interagir avec ce nouveau processus. Ces liaisons sont transmises au processus léger responsable des communications (*Routeur*) qui les utilisera pour construire une interconnexion entre les différents processus.

Le processus léger *Routeur* doit fournir deux services différents : un service de routage et un protocole de communication basé sur des messages actifs.

La phase de lancement décrite dans la section précédente construit un ensemble de processus interconnectés par des liaisons de contrôle avec une topologie arborescente. Ces liaisons de contrôle sont établies par les protocoles standards d'exécution distante en vue de rediriger les entrées / sorties standards du processus distant vers l'utilisateur. Dans notre cas, ces liaisons sont détournées pour former un réseau de contrôle dédié à l'acheminement de communications. Pour cette raison, les différents processus ne possèdent plus de moyens d'interaction avec l'utilisateur. Un mécanisme de redirection d'entrées / sorties standards sera fourni et détaillé dans la section suivante.

Pour permettre d'établir une abstraction de réseau logique complètement maillé, chaque noeud doit fournir un service de routage logiciel au travers d'un processus léger (*Routeur*). Ce processus léger gère les liaisons du réseau logique de contrôle de

chaque noeud. Comme le routeur applicatif doit filtrer toutes les communications qui arrivent sur un noeud, il est également responsable de l'implantation du protocole de communication du réseau logique de contrôle.

Ce protocole de communication est de type message actif, c'est à dire que la réception d'un message provoque l'exécution d'un traitement associé. L'entête de chaque message est lu par le *Routeur*, suivant le destinataire du message. Ce message est alors, soit redirigé sur la liaison adéquate en utilisant les tables de routage, soit passé en argument du traitant[2] associé.

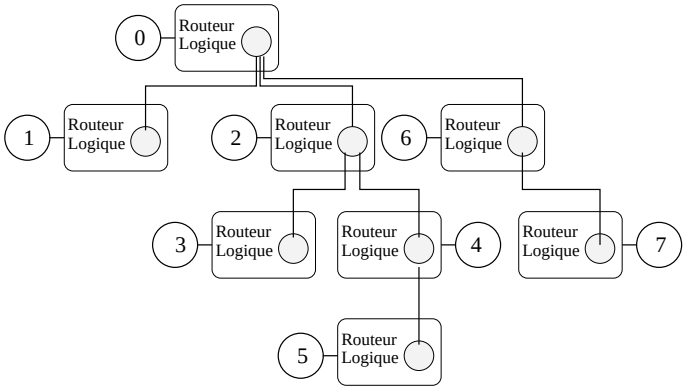


FIG. 6.4 – *Numérotation des noeuds en profondeur sur le réseau de contrôle*

Avant de réaliser un routage logique des différents messages, le processus léger "*Routeur*" attend une barrière de synchronisation de manière à comptabiliser l'ensemble des processeurs virtuels participants. Durant la phase de déploiement certains noeuds de la grappe n'ont peut être pas été atteints, le nombre effectif est alors inférieur au nombre demandé. Cette barrière de synchronisation est également utilisée pour numéroter de manière unique et contigüe l'ensemble des processeurs virtuels participants. Cette numérotation est effectuée en profondeur d'abord sur l'arbre de lancement. La figure 6.4 présente un exemple d'arbre couvrant numéroté de cette manière. Cette numérotation est ensuite utilisée pour construire les tables de routages des processus légers "*Routeur*".

6.1.3 Services de contrôle

Le processus racine de l'arbre de lancement correspond à un processus de contrôle. Ce processus est le représentant auprès de l'utilisateur, des différents processus composant l'application parallèle. Pour cela, il doit permettre aux différents processus d'interagir avec l'utilisateur en fournissant la redirection transparente des sorties standards de chacun d'eux. Les sorties standards de chaque processus sont redirigées sur un processus léger (IO) chargé de les transférer au noeud racine par messages actifs.

[2]L'exécution du traitement associé à chaque message actif est réalisée par le routeur, l'exécution du code de ce traitement peut donc provoquer une dégradation de la réactivité du réseau de contrôle.

Le traitement associé à ces messages actifs consiste à afficher sur la console du processus de contrôle les sorties de chaque processus. Les entrées standards sont redirigées suivant le même principe, par la diffusion des données émises sur l'entrée standard du processus de contrôle à tous les processus composant l'application parallèle. Les signaux de contrôle sont redirigés de la même manière.

6.2 Déploiement adaptatif

Cette section présente le mise en oeuvre d'un déploiement adaptatif respectant la stratégie "au plus tôt" présentée dans le chapitre 5. Cette implantation utilise pour cela un algorithme de vol de travail pour la construction dynamique de l'arbre de déploiement et un indice de performance permettant d'évaluer dynamiquement le taux de concurrence des instances de connecteurs. Ces performances et ces capacités d'adaptation seront présentées dans le chapitre suivant.

6.2.1 Mise en oeuvre du "vol de travail hiérarchique"

La figure 6.5 montre l'architecture multi-threadée utilisée pour implanter l'algorithme "glouton" de "vol de travail" présenté dans la section 5.3. Cette implantation ajoute un module de gestion de liste de tâches sur chaque noeud aux modules déjà présentés dans la section précédente. Le gestionnaire local de tâche envoie et traite les requêtes de vol de travail effectuées localement par le module de lancement ou reçues depuis un noeud voisin.

La section 6.1.2 insiste sur le fait que le traitement associé à un message actif est exécuté par le processus léger *Routeur*. Pour cette raison et afin d'éviter tout risque d'inter-blocage, le traitement associé à une requête de vol de travail est très limité. Il consiste à enregistrer cette demande dans une liste de requêtes de vol. Cette liste sera consommée par le processus léger qui gère la liste locale de tâches disponibles. Les différents services fournis par le gestionnaire de travail sont de répondre à une demande locale ou distante de travail et de transférer cette requête au cas où la liste locale est vide.

Cette implantation est strictement hiérarchique, un processeur virtuel (gestionnaire de travail) ne demande des tâches qu'à son père dans l'arbre de lancement. Cette stratégie simplifie la mise en oeuvre de l'algorithme de vol et limite la distance parcourue par une requête à la profondeur de l'arbre. Ce mécanisme permet de limiter également le phénomène de ping pong de tâches entre deux noeuds. Cependant, l'utilisation d'une méthode hiérarchique implique que lorsqu'une tâche est attribuée à un sous-arbre, elle sera obligatoirement traitée par un noeud de ce sous-arbre, même si un ou plusieurs autres noeuds n'appartenant pas à ce sous-arbre sont inoccupés.

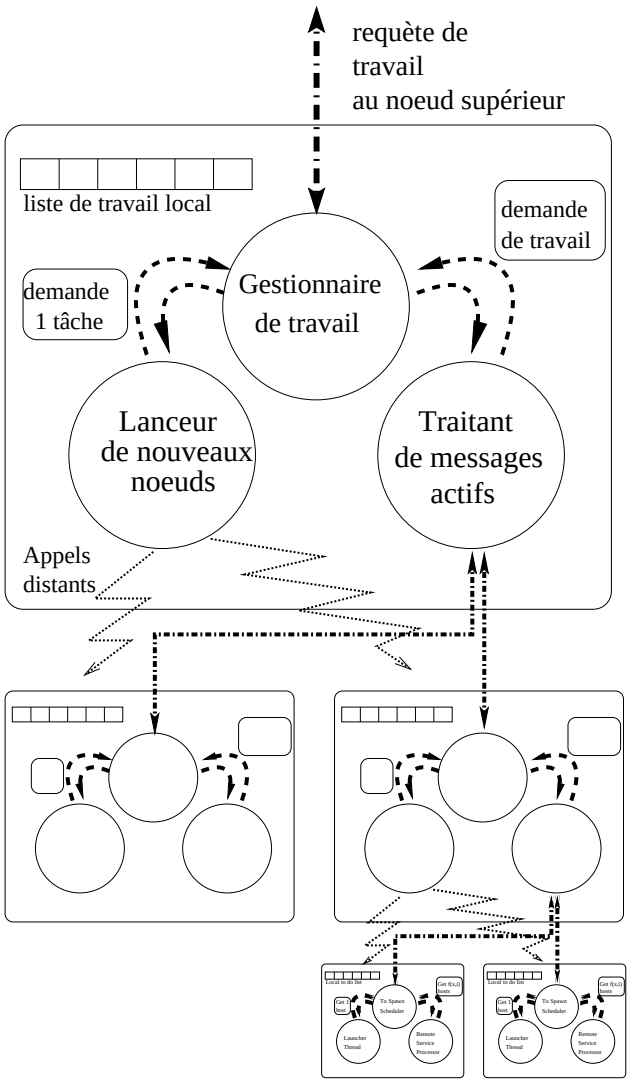


FIG. 6.5 – *Architecture multi-threadée permettant l'implémentation du vol de travail.*

6.2.2 Caches de tâches

Afin de diminuer le nombre de requêtes de vol de travail total, nous utilisons un mécanisme de cache de travail alimenté a priori. Ce mécanisme a pour objectif de factoriser le nombre de requêtes et ainsi de réduire le surcoût des communications de cette stratégie "au plus tôt".
Pour cela, plus un noeud consomme de tâches (localement ou par ses fils) plus le nombre de tâches retournées à chaque requête augmente (cf. Fig 6.6). Le processus léger de gestion de travail comptabilise le nombre de requêtes déjà émises pour chacun de ses fils dans l'arbre. Cet indice de performance est utilisé pour réaliser un cache de tâche sur ce noeud. Plus un noeud ou son sous-arbre est grand (le résultat de chaque tâche ajoute un processeur virtuel), plus la probabilité qu'une ou plusieurs requêtes de vol de tâches arrivent rapidement augmente. Cependant, comme cette méthode dynamique est destinée à des environnements hétérogènes, il est possible, qu'à un certain stade du déploiement la fréquence de traitement d'une tâche change

fortement. Aussi, afin de limiter le risque d'erreur sur l'ordonnancement d'une stratégie "au plus tôt" et pour augmenter la réactivité de cette implantation, le nombre de tâches retournées à chaque requête est borné.

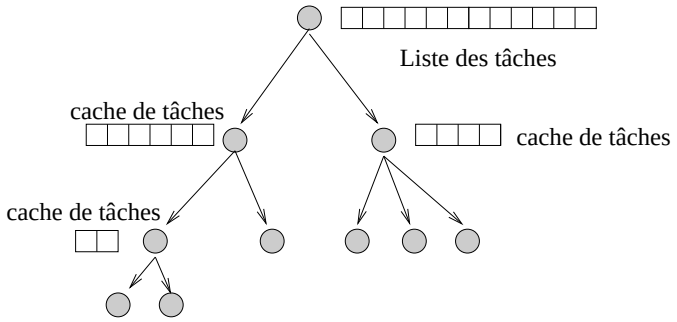


FIG. 6.6 – Stratégie de vol hiérarchique avec cache de taches.

6.2.3 Indice d'estimation de la multiprogrammation

L'implantation d'un mécanisme de déploiement adaptatif repose également sur l'évaluation dynamique du taux de concurrence des différentes instances de connecteur en cours de connexion. Un nombre trop important de processus (légers ou normaux) peut dégrader les performances d'un noeud et annuler les bénéfices de la multiprogrammation. Pour cette raison, il est important d'évaluer si la somme de travail des connecteurs en cours d'exécution est adaptée aux capacités du processeur et de l'interconnexion réseau. Cet indice de charge représente une estimation du taux de concurrence mesuré dans un milieu homogène (Cf. section 5.1.2.1). Dans un milieu hétérogène où les noeuds ont des capacités différentes, ce taux de concurrence ne peut être évalué a priori de manière statique. De plus, dans le cadre des outils d'exploitation les noeuds cibles peuvent être en cours d'utilisation, dans ce cas l'indice du taux de concurrence des connecteurs doit pouvoir prendre en compte les différents processus utilisateurs avec lesquels il partage les ressources d'un noeud (processeur, interface réseau).
Pour cela, l'implantation actuelle repose sur l'évaluation de la charge d'un noeud fourni par le système d'exploitation. Cet indice rend compte des différents processus créés pour effectuer le lancement (instances de connecteurs concurrents) ainsi que des ressources utilisées par d'autres processus en cours d'exécution sur un noeud.

6.3 Langage de description

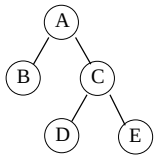
Les sections précédentes de ce chapitre présentent la mise en oeuvre des principes de déploiement décrits dans le chapitre 5. La section 5.2.2.1 introduit l'algorithme récursif de lancement implanté par le module *Lanceur* décrit dans la section 6.1.1. Ce

déploiement récursif prend en paramètre une description de l'arbre de diffusion à déployer. Cette section a pour objectif de préciser comment cette description est implantée dans Taktuk. Nous verrons également les extensions qui ont été apportées à la description de l'arbre de diffusion de manière à traiter les environnements hétérogènes, à décrire une grappe multi-réseaux et à supporter le déploiement d'une application (parallèle ou d'exploitation) sur une plate-forme composée de plusieurs grappes.

6.3.1 Description de l'arbre de diffusion

L'algorithme récursif de la section 5.2.2.1 prend en paramètre la description de l'arbre de diffusion devant être réalisé par chaque noeud. Cette description est passée en paramètre au moment de la création distante d'un nouveau processus. Elle contient la description du sous arbre de diffusion que devra réaliser ce noeud. La topologie de ce sous arbre est implantée par un langage parenthésé interprété par chaque noeud. Ce langage permet de décrire de façon naturelle une topologie arborescente.

$$A(B, C(D, E)) \Longleftrightarrow$$



Il permet également de répartir le traitement de son interprétation, chaque noeud n'interprète que le premier niveau de ce langage qui décrit la liste des fils d'un noeud. Le second niveau de parenthèses délimite le sous arbre de chacun de ses fils. Lorsque le noeud réalise la création d'un processus distant, il passe en paramètre la description du sous arbre associé sans analyser son contenu. De cette manière l'analyse effectuée sur chaque noeud a un coût de traitement proportionnel au nombre de ses fils. De plus, la taille des arguments diminue exponentiellement au fur et à mesure du déploiement.

Cette description précise de l'arbre de diffusion à utiliser permet de séparer l'implantation des mécanismes de déploiement de l'implantation de l'algorithme de calcul de l'arbre de diffusion le plus adapté.
La description de l'arbre peut être complète ou abrégée au bout d'un certain niveau de récursion par l'utilisation de topologies de base prédéfinies et implantées par le mécanisme de lancement. Ces topologies de bases sont : une topologie simple d'arbre N-aire équilibré ou une topologie dynamique et adaptative (Cf. section 5.3), dans ce cas la description se résume à une liste de noeuds.

6.3.2 Environnement hétérogène ou hiérarchique

Pour traiter le cas d'un environnement hétérogène le langage de description parenthésé décrit ci-dessus est enrichi par des paramètres spécifiques à un noeud ou à un groupe de noeuds. L'analyse sémantique de ce langage est effectuée de gauche à droite, la définition d'un paramètre est appliquée à la liste de tous les noeuds suivants

dans le même niveau de parenthèse ou jusqu'à une nouvelle définition de ce paramètre dans ce niveau.
Ces paramètres permettent de spécifier :

- un nom d'utilisateur distant, permettant de réaliser l'authentification de sécurité (Cf. section 4.1.1).
- le type de connecteur devant être employé pour atteindre un groupe de noeuds.
- Un temps limite ("timeout") au bout duquel un noeud sera considéré comme défaillant.
- le type de topologie de base à appliquer sur la liste suivante, arbre N-aire, adaptatif. Par défaut ces noeuds sont les fils directs du noeud du niveau de parenthèse précédent.

Exemple :

$$A(utilisateur1\ B(utilisateur2\ D, E), C(utilisateur3\ F, G))\ H\ utilisateur4\ I\ J$$

La description complète de l'arbre ou de certains niveaux en particulier permet de distinguer une hiérarchisation des noeuds cibles. De cette façon, les noeuds "rapides" (plus performants ou moins chargés) peuvent être placés dans les niveaux supérieurs de l'arbre couvrant. Ces noeuds performants seront alors chargés du déploiement des noeuds moins réactifs. Dans certains cas pathologiques, l'algorithme adaptatif décrit dans la section 5.3 peut être mis en échec par l'engorgement des différents processeurs virtuels actifs par le lancement de noeud "lent". Le mécanisme de description peut alors être utilisé pour orienter cet algorithme et considérer en premier les noeuds connus comme étant réactifs.

6.3.3 Traitement des plates formes multi-réseaux

L'objectif de cette bibliothèque de déploiement est de fournir une base générique permettant la réalisation de nombreux environnements de programmation parallèle. Certains environnements utilisent simultanément plusieurs réseaux physiques à faible latence. Ces environnements, dits multi-réseaux, ont été présentés dans les sections 2.2.2 et 3.1.4. Dans l'hypothèse où il existe un réseau fédérateur de type LAN 802, la mise en oeuvre d'un environnement de ce type peut se faire avec un outil de déploiement comme Taktuk. Pour traiter cette particularité, Taktuk offre la possibilité de transmettre des arguments à l'application parallèle déployée. Pour améliorer les performances et pour éviter de diffuser la description complète de l'architecture sur tous les noeuds. Des arguments spécifiques peuvent être délivrés pour chaque noeud ou groupes de noeuds. Ce mécanisme peut également être utile pour des applications d'exploitation devant effectuer une opération commune avec des arguments différents spécifiques pour certains noeuds cibles.

6.3.4 Traitement des plates-formes multi-grappes

Ce mécanisme de description a été mis en oeuvre avec comme objectif de décrire diverses topologies d'arbres de déploiement. Il permet également de spécifier la topologie du réseau de contrôle établi par le déploiement en suivant l'arbre de diffusion. Suivant son utilisation, la topologie de cet arbre influe directement sur les performances du déploiement ou sur les opérations de communication effectuées sur le réseau de contrôle.
Ce mécanisme permet également de déployer une application sur des plates-formes composées de plusieurs grappes. Ces plates-formes se regroupent sous le terme de grille légère [69] par opposition aux projets de grilles nationaux ou internationaux visant à regrouper des grands centres de calcul. Ces agglomérats de grappes sont à l'initiative d'un utilisateur possédant un compte sur plusieurs ressources de calcul. Pour exploiter ces plates-formes, l'utilisateur doit déployer son ou ses applications parallèles sur l'ensemble des noeuds.
La figure 6.7 illustre un exemple de configuration utilisant deux grappes de calcul et un ensemble de noeuds appartenant à un réseau local interne (intranet).

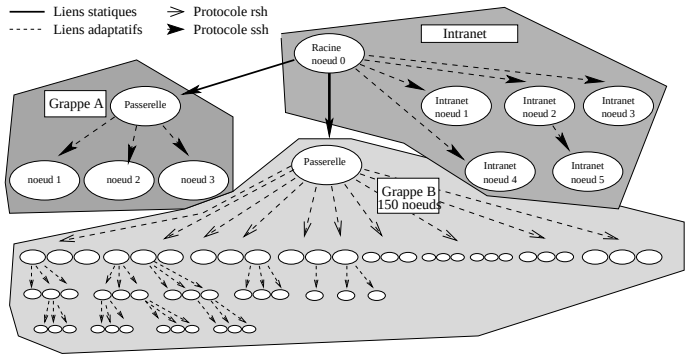


FIG. 6.7 – *Présentation d'un exemple de configuration de grille légère. La topologie du déploiement est décrite par le langage de description.*

Cependant le déploiement d'un environnement de programmation sur des plates-formes de ce type pose souvent des problèmes dûs à des contraintes de sécurité. En effet, comme nous l'avons dit, les grappes représentent un milieu clos souvent isolé par des noeuds spécifiques destinés à autoriser un utilisateur à exploiter les ressources de calcul de la grappe. Ces noeuds sont nommés passerelles. L'accès à un noeud d'une grappe doit être effectué via un noeud de type passerelle. Le déploiement d'une application parallèle sur de telles plates-formes doit donc prendre en compte ces noeuds "imposés" permettant d'atteindre les noeuds d'une grappe. La description de l'arbre de déploiement peut permettre de traiter facilement ce cas. Cependant pour atteindre le premier noeud, racine du déploiement sur une grappe, le mécanisme de description doit être étendu de manière à fournir un chemin d'accès à ce noeud. Le code de l'application parallèle ainsi que celui de la bibliothèque n'est pas présent sur ces noeuds passerelles. La figure 6.8 décrit le mécanisme utilisé pour atteindre le premier noeud de la grappe participant à l'application parallèle. Pour cela, il suffit de donner

en argument à un connecteur générique spécifique le chemin (commande d'exécution distante) permettant d'atteindre le noeud cible.

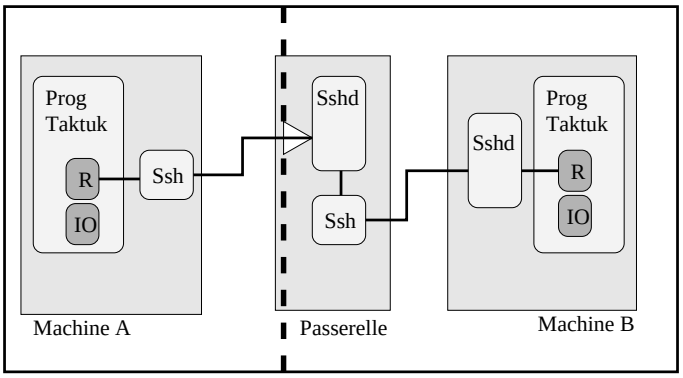


FIG. 6.8 – Méthode utilisée pour atteindre l'un des noeuds d'une grappe protégé par un noeud "passerelle".

6.3.5 Bilan

Le langage de description mis en oeuvre permet de réaliser le déploiement d'un réseau de contrôle suivant toutes les topologies d'arbres possibles. Il est ainsi possible d'adapter la topologie de ce réseau de contrôle à l'utilisation de la bibliothèque Taktuk, suivant que l'on souhaite optimiser le déploiement ou les communications de ce réseau de contrôle. La forte extensibilité de ce langage simple permet de décrire des plates-formes hétérogènes ou des agglomérats de grappes (grilles légères).
L'analyse et la diffusion de la description de la plate-forme cible sont réparties et génèrent un faible surcoût sur le mécanisme de lancement.

6.4 Utilisation

La figure 6.9 montre un exemple d'utilisation de la bibliothèque Taktuk dans le cas d'un programme très simple, effectuant un affichage à l'écran. Toutes les instructions qui sont à l'intérieur du couple "*Taktuk.init(...)*" et "*Taktuk.terminate()*" seront exécutées par tous les processeurs virtuels (processus distants) composant cette application "parallèle". La figure 6.10 montre l'appel de ce programme "parallèle" ainsi que le résultat de son exécution sur 4 machines, les machines A, B et C ainsi que la machine sur laquelle le programme a été démarré.
Cet exemple illustre un programme "Taktuk" simple, mais il peut être complété avec des appels d'exécution de services distants fournit par Taktuk.
L'implantation du déploiement sous forme de bibliothèque permet de le rendre transparent à l'utilisateur. De cette manière, le démarrage d'un programme parallèle est identique au démarrage d'un programme séquentiel avec quelques arguments supplémentaires utilisés et consommés par Taktuk pour réaliser le déploiement

souhaité par l'utilisateur.

```
Taktuk.init(...)
print("bonjour tout le monde")
Taktuk.terminate()
```

FIG. 6.9 – Exemple d'un programme simple de type "bonjour tout le monde", dont le déploiement et l'exécution seront gérés entièrement par la bibliothèque Taktuk

```
>./bonjour -v -cssh -m machineA -m machineB -m machineC
<machine_locale> [rank:0] :->:bonjour tout le monde
<machineA> [rank:2] :->:bonjour tout le monde
<machineB> [rank:1] :->:bonjour tout le monde
<machineC> [rank:3] :->:bonjour tout le monde
```

FIG. 6.10 – Exécution de l'exemple "bonjour" définit par le programme 6.9.

6.5 Conclusion

Ce chapitre a présenté la mise en oeuvre des principes de parallélisation du déploiement sous la forme d'une bibliothèque de programmation nommée Taktuk. Cette bibliothèque doit permettre de développer des outils d'exploitation, d'administration et de déploiement d'application parallèle efficaces et passant l'échelle. Elle a été implantée de manière flexible et extensible. Le chapitre 7 propose la validation expérimentale de cet environnement. Le chapitre 8 présente les différents projets qui utilisent ou reposent sur les fonctionnalités de cette bibliothèque.

Chapitre 7

Validation expérimentale

Le chapitre précédent a présenté l'implantation des principes de déploiement énoncés dans les chapitres 4 et 5 dans la bibliothèque Taktuk.
Ce chapitre a pour objectif de valider cette implantation. Dans une première section, nous présenterons les résultats obtenus en exploitant les mécanismes de parallélisation présentés dans le chapitre 4. Ces premiers résultats, basés sur une approche naïve (arbres de diffusion N-aires équilibrés), nous serviront de références pour les expérimentations suivantes. Ils permettront également d'évaluer le surcoût de l'implantation de ces principes de base. La deuxième section présentera les résultats obtenus avec l'utilisation d'un arbre de diffusion optimal en fonction des caractéristiques des connecteurs utilisés. Les résultats théoriques obtenus dans la section 5.2.2.2 qui représentent une borne inférieure, serviront à estimer le comportement de cette implantation. Une troisième section étudiera le comportement de l'implantation de la stratégie de déploiement adaptative en présentant les performances de déploiement obtenues dans un milieu homogène puis hétérogène. La quatrième section présentera les performances des communications sur le réseau de contrôle mis en place par le déploiement. Ces résultats mettront en évidence l'incidence des connecteurs utilisés pour réaliser le déploiement sur les communications du réseau de contrôle.

Conditions expérimentales

Ces expérimentations ont été conduites sur la plate-forme Icluster. Cette grappe est composée de 100 PCs, interconnectés par de l'éthernet commuté 100 Mbps. Ces machines sont réparties sur 3 commutateurs reliés par un réseau 2 Gbps. Les noeuds sont des Pentium III 733 MHz possédant 256 Mo de mémoire sous Linux 2.4.18. Les protocoles d'exécution distante utilisés pour ces expérimentations sont : *rsh* (*Linux Netkit 0.17*) et *ssh* (Protocole version 2 *OpenSSL 3.1p1*). Toutes les informations nécessaires à l'authentification sont répliquées sur tous les noeuds. De cette manière, les phases d'authentification des deux protocoles utilisés ne nécessitent aucun accès externe (serveur d'authentification NIS ou serveur de nom DNS). Ces protocoles centralisés provoquent des "goulots" d'étranglement et ne permettent pas de considérer des temps

de lancement homogènes.
Les mesures de déploiement présentées dans les sections suivantes ont été réalisées en utilisant la commande "taktukexec" qui déclenche l'exécution d'un programme standard sur un ensemble de noeuds. Les temps mesurés inclus le déploiement et l'interconnexion d'un ensemble de processus "taktukexec", la création d'un nouveau processus pour exécuter la commande spécifiée et une barrière de synchronisation permettant de détecter la terminaison de l'exécution. Pour toutes les mesures nous considérons la médiane sur 10 lancements d'une commande de temps nul.
Malgré ce contexte expérimental, l'homogénéité de comportement d'un ensemble de 100 noeuds est difficile à garantir. Pour cette raison, certaines des courbes suivantes, présentent un "décrochage" aux environs du $55^{ième}$ noeud. Cette perturbation est due à un noeud moins réactif qui perturbe les mesures en provoquant un ralentissement. L'allure générale des courbes reste cependant valide.

7.1 Validation des principes de parallélisation

Cette section présente l'étude de performance des principes de parallélisation présentés dans le chapitre 4. Nous étudierons dans un premier temps les performances obtenues par l'utilisation seule de la multiprogrammation. Puis, nous présenterons les résultats obtenus avec l'utilisation conjointe de la multiprogrammation et d'un arbre de diffusion N-aire équilibré.

7.1.1 Évaluation de la multiprogrammation

Les résultats de la figure 7.1 présentent les performances obtenues lors d'un déploiement d'une application sur 100 noeuds en utilisant deux types de connecteurs "internes" pour le protocole rsh et deux connecteurs "externes", respectivement pour le protocole "rsh" et "ssh". Ces résultats comparent l'implantation de la multiprogrammation en utilisant, soit des processus, soit des processus légers (threads), soit une nouvelle implantation permettant des envois de requêtes de connexion asynchrones. Cette dernière implantation est assez coûteuse en temps de réalisation et n'est envisageable que pour des protocoles simples comme rsh. Cette comparaison montre que les résultats obtenus entre ces deux connecteurs "internes" sont similaires. Le surcoût d'utilisation de processus légers n'est pas suffisant sur 100 noeuds, pour dégrader les performances. Cependant, il est réaliste de penser qu'une approche asynchrone ait de meilleure capacité de passage à l'échelle au delà de 100 noeuds.
L'utilisation de la multiprogrammation a pour objectif de paralléliser la partie recouvrable de chaque appel d'exécution distante. Comme les parties non recouvrables du protocole (ici rsh) sont exécutées séquentiellement, il est normal que l'allure générale de ces courbes soit linéaire en fonction du nombre de noeuds.
La figure 7.1 montre également que l'utilisation de processus pour implanter la multiprogrammation dégrade fortement les performances, en particulier pour un protocole coûteux comme "ssh".

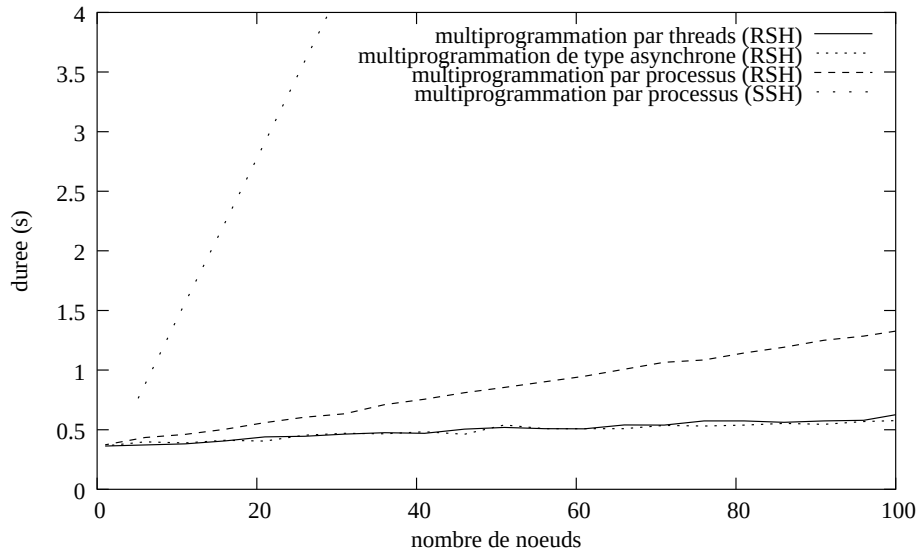


FIG. 7.1 – Comparaison d'un lancement exploitant différentes implantations de multiprogrammation pour les protocoles **rsh** et **ssh**.

Ces résultats montrent qu'une approche essentiellement basée sur la multiprogrammation donne de bons résultats en terme de performances pour des protocoles légers comme "rsh" (environ 600 ms pour effectuer le déploiement). Ils montrent également que les efforts nécessaires à l'implantation d'un client parallèle asynchrone n'est pas réellement "rentable" et qu'une approche basée sur des processus légers fournit de bons résultats. Cependant, les résultats obtenus restent linéaires du nombre de noeuds et limitent les capacités de passage à l'échelle d'une telle approche.
La section suivante présente l'évaluation d'un déploiement exploitant conjointement la multiprogrammation et la distribution des appels distants en utilisant un arbre de diffusion N-aire équilibré.

7.1.2 Évaluation des principes de parallélisation

Les résultats des courbes de la figure 7.2 représentent les résultats obtenus pour des arbres de déploiement N-aires. L'arité de ces arbres représente le nombre d'instances concurrentes de connecteur. Ces résultats correspondent aux meilleures topologies d'arbre de diffusion pour les protocoles ssh et rsh. Ces arités ont été trouvées de manière empirique par comparaison de plusieurs arbres pour chaque protocole, pour faciliter la lisibilité des courbes les arités moins performantes ne sont pas représentées ici. La figure 7.2 ne présente pas les résultats obtenus avec les arbres les plus performants pour les connecteurs internes du protocoles rsh. En fait, le taux de concurrence très important de ces connecteurs suffit à traiter une grappe de 100 noeuds. L'arbre le plus efficace correspondant est donc un arbre "plat" dont les résultats ont été présentés dans la section précédente.

Les résultats obtenus avec cette approche naïve, utilisant des arbres de diffusion N-aires sont assez bons (3.5 sec pour ssh et 1 sec pour rsh sur 100 noeuds). L'aspect

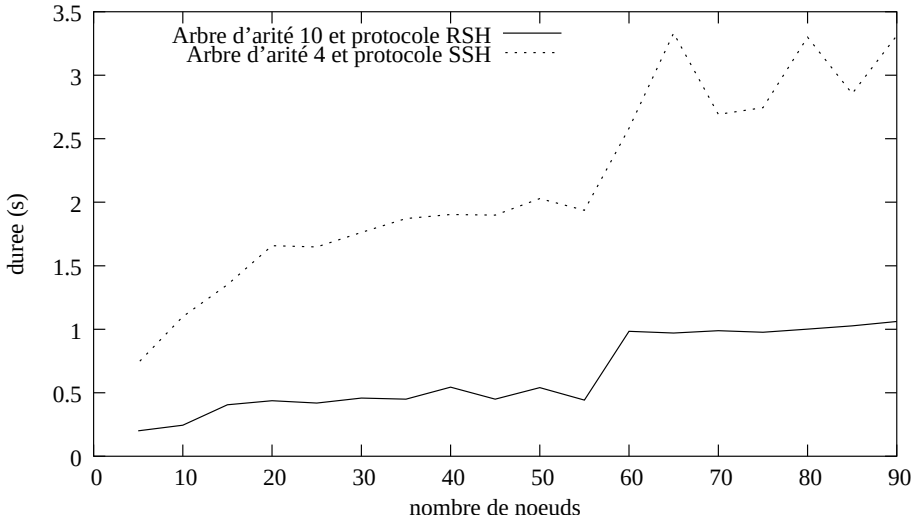


FIG. 7.2 – Meilleurs résultats obtenus avec des arbres de diffusion N-aires, pour les connecteurs externes rsh et ssh.

général des courbes confirme que l'implantation de ces mécanismes produit un coût de déploiement logarithmique du nombre de noeuds. Le "décrochage" des courbes entre les mesures sur 55 et 60 noeuds est dû à l'ajout d'un noeud moins réactif (non homogène malgré les précautions mises en oeuvre). Ces résultats donnent également une évaluation des taux de concurrence les plus performants pour les protocoles rsh et ssh. Ils sont respectivement de 4 pour les connecteurs ssh et de 10 pour les connecteurs rsh. Ces résultats sont différents de ceux mesurés expérimentalement dans la section 5.1.2.1 car ici les mesures incluent le surcoût de l'implantation de la multiprogrammation (connecteurs "externes") et de l'implantation du déploiement récursif.
La section suivante propose de mesurer le surcoût de l'environnement Taktuk sur une demande d'exécution distante entre deux noeuds avec les protocoles rsh et ssh.

7.1.3 Évaluation du surcoût de l'environnement Taktuk

Le tableau 7.1 représente le surcoût de Taktuk utilisant un connecteur "externe" rsh ou ssh par rapport au coût d'un appel d'exécution distante réalisé avec le client de ce protocole. Ces résultats représentent la moyenne sur une série de 50 mesures et l'erreur associée à un interval de confiance à 95%. Ce surcoût s'explique par le fait que dans un appel de client de protocole, le serveur exécute directement la commande spécifiée en argument, ici une commande de temps nul. Tandis que lors de l'utilisation de Taktuk, le serveur d'exécution distante démarre un nouveau processus Taktuk. Ces deux processus "Taktuk" initialisent le réseau de contrôle puis le processus distant exécute la commande de temps négligeable. Cependant, ce surcoût se recouvre très bien par des appels d'exécution distante concurrents. La figure 7.2 montre que le temps de lancement d'une commande de temps négligeable sur 5 et 10 noeuds est comparable au temps de lancement sur 1 noeud (Cf. 7.1). Ce surcoût unitaire est donc subi à chaque étage de l'arbre de diffusion, le surcoût total sur le déploiement est donc logarithmique du nombre de noeuds.

	client rsh	Taktuk + rsh	client ssh	Taktuk + ssh
Tps en ms	55	148	337	422
Erreur IC 95%	61 / 49	155 / 142	344 / 330	438 / 406

TAB. 7.1 – Surcoût de l'utilisation d'un client de protocole via un connecteur par Taktuk

7.2 Évaluation de la stratégie optimale de lancement

Les résultats présentés par la figure 7.3 ont pour objectif de mesurer la qualité de l'implantation du déploiement de la bibliothèque Taktuk. Ces résultats comparent la borne inférieure théorique obtenue pour les protocoles rsh et ssh avec les résultats obtenus expérimentalement.
Pour cela, les performances de déploiement ont été mesurées en utilisant les arbres de diffusion théorique et optimale définis dans la section 5.2.2.2. Les arbres de diffusion optimaux utilisés pour les deux protocoles (rsh et ssh) sont représentés dans la figure 7.4. Ces arbres ont été construits par l'algorithme présenté dans la section 5.2.2 prenant en paramètre les valeurs $tu = t + \delta$ qui déterminent le temps total (tu) d'un appel d'exécution distante et le temps non recouvrable (t) de cet appel. Les mesures de tu et t utilisées pour construire les arbres utilisés durant cette expérimentation correspondent à celles mesurées dans la section 5.1.2.1, soit $tu = 330$ et $t = 140$ ms pour ssh et $tu = 85$ et $t = 10$ ms pour rsh.

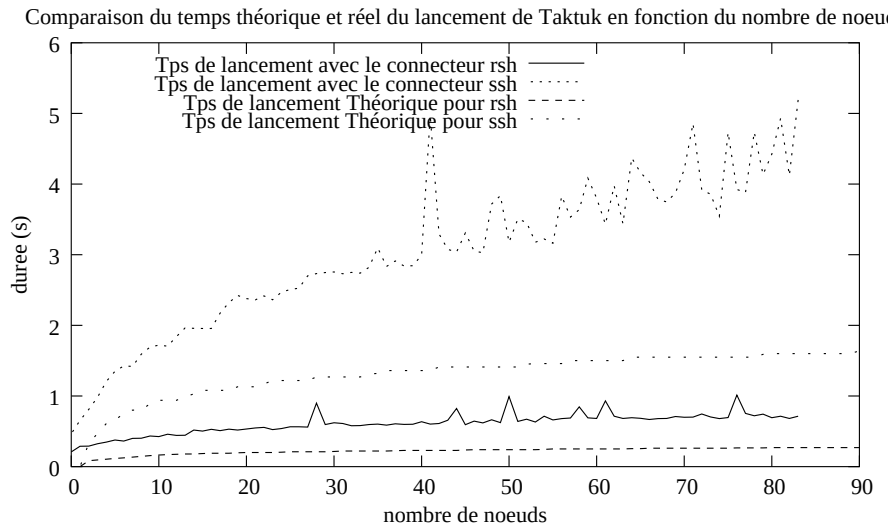


FIG. 7.3 – Comparaison des résultats théoriques prévisionnels vs. résultats.

Les résultats obtenus sont différents des résultats théoriques, ceci provient de plusieurs facteurs. Premièrement, le modèle théorique présenté dans la section5.1.2 ne prend pas en compte le surcoût du mécanisme de déploiement (qui peut être différent suivant la position du noeud dans l'arbre). Ce modèle ne considère que les appels d'exécution distante. C'est pour cela qu'il ne fournit qu'une borne inférieure sur le

temps de déploiement que l'on peut obtenir avec les protocoles ssh et rsh. De plus, l'hypothèse d'un milieu homogène reste forte sur une plate-forme réelle.
Deuxièmement, les paramètres tu et t utilisés pour construire les arbres de diffusion ne correspondent pas à la réalité. En effet, les temps observés ont été mesurés avec les clients des protocoles et non pas avec l'environnement Taktuk qui utilise ces protocoles. La section précédente montre que le surcoût d'implantation introduit par Taktuk n'est pas négligeable. Elle montre également que ce surcoût n'intervient qu'entre les étages de l'arbre de lancement car il est recouvrable par multiprogrammation.

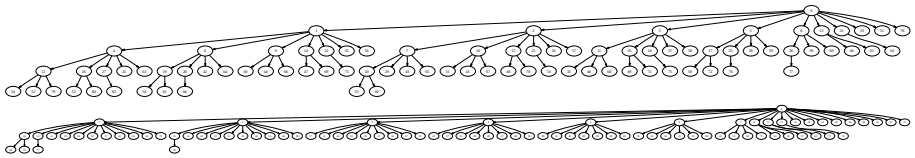


FIG. 7.4 – Arbres de déploiement utilisés comme arbre de diffusion optimale

7.3 Evaluation de la stratégie adaptative

L'un des objectifs d'une approche adaptative est de pouvoir fournir de bonnes performances dans un milieu hétérogène. Cependant, si ces capacités d'adaptation sont bonnes, elle doit également produire un déploiement efficace dans un milieu homogène. Cette section propose donc d'étudier le comportement de l'implantation de l'approche adaptative dans ces deux milieux.

7.3.1 Approche adaptative dans un milieu homogène

Cette section propose d'étudier le comportement de l'approche adaptative dans un milieu homogène. L'objectif est de fournir une approche performante ne nécessitant pas de phase de réglage préliminaire. La première partie valide la construction dynamique de l'arbre de diffusion par l'algorithme de "vol de travail". La seconde valide l'indice de charge utilisé pour déterminer dynamiquement le taux de concurrence de la multiprogrammation.

7.3.1.1 Évaluation de l'implantation de la stratégie : "glouton et vol de travail hiérarchique"

La figure 7.5 compare les meilleurs résultats d'une approche statique basée sur des arbres de diffusion N-aires et ceux obtenus avec une approche basée sur une stratégie de type "glouton et vol de travail hiérarchique". Ces résultats sont très proches et permettent de valider les choix d'implantation de la stratégie adaptative réalisés dans la section 6.2.1.

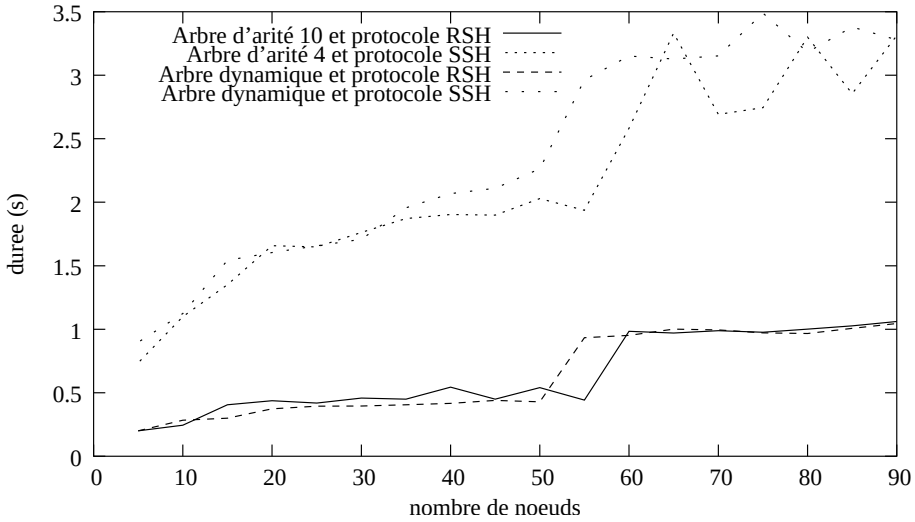


FIG. 7.5 – Comparaison des résultats de l'approche dynamique avec des arbres de diffusion N-aire statique

7.3.1.2 Validation de l'indice dynamique du taux de concurrence

La figure 7.6 a pour objectif de valider l'indice de charge utilisé pour estimer dynamiquement le taux de concurrence de plusieurs instances de connecteurs. Pour cela, elle compare un déploiement où ce taux de concurrence est donné en paramètre et un déploiement réalisant son estimation.
Les valeurs utilisées n'ont pas été calculées ($\tau = tu/t$) en fonction des paramètres tu et t, pour les raisons citées dans la section 7.2. Les taux de concurrence utilisés pour chaques protocoles sont ceux qui ont été estimés en fonction des résultats obtenus sur les expérimentations de déploiement utilisant des arbres de diffusion N-aires qui exploitent la multiprogrammation. Cette estimation correspond à l'arité des arbres obtenant les meilleures performances soit un taux de concurrence $\tau = 4$ pour le protocole ssh et $\tau = 10$ pour rsh.

Les résultats obtenus sont très proches car l'implantation actuelle de la détection de charge (taux de concurrence) fonctionne de la manière suivante : lorsqu'un processus léger de lancement détecte une surcharge potentielle, il attend que tous les connecteurs en cours de connexion terminent. Ceci implique que l'implantation ne suit pas exactement le modèle de pipeline énoncé dans la section 5.1.2.1. Cette implantation introduit donc des délais d'attente supplémentaire et ne suit pas strictement une stratégie "au plus tôt".
Ce mode de fonctionnement est proche de celui réalisé par une approche statique basée sur des arbres N-aires ; pour cette raison les résultats obtenus sont comparables.

7.3.2 Approche adaptative dans un milieu hétérogène

Dans une approche statique, les noeuds et les coûts de communication sont considérés comme complètement homogènes. Lorsque dans une situation réelle, cet

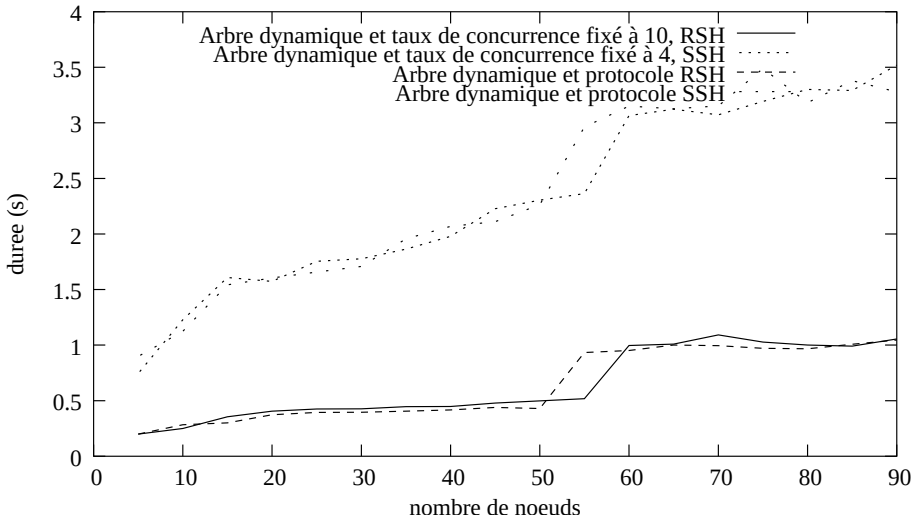


FIG. 7.6 – Validation du taux de concurrence évalué dynamiquement avec une approche dynamique connaissant le taux de concurrence d'un connecteur.

ordonnancement est utilisé, il devient très sensible à cette hétérogénéité. Ceci peut conduire à de fortes dégradations des performances. La figure 7.7 montre les résultats obtenus avec une approche statique et dynamique utilisant le protocole ssh. Chaque point correspond à la moyenne de 21 mesures. Pour mettre en évidence les gains d'une approche adaptative, nous avons "chargé" artificiellement 6 noeuds. Ces noeuds étant très chargés, ils deviennent moins réactifs et le coût d'un appel d'exécution distante augmente.

Les résultats montrent de meilleures performances pour l'approche adaptative. Ceci s'explique en observant les arbres de lancement obtenus (figure 7.7), respectivement avec l'approche statique (arbre du haut) et avec l'approche dynamique (arbre du bas). Dans l'approche statique, la topologie de l'arbre et l'attribution des machines sur les noeuds de cet arbre est effectuée a priori. Il est donc possible que plusieurs noeuds "chargés" soient sur le même chemin critique (de la racine à une feuille). Comme le chemin critique de cet arbre est une borne inférieure du temps de lancement (sur une branche, chaque appel distant est effectué séquentiellement). Il est évident que le fait d'avoir plusieurs noeuds chargés sur ces chemins dégrade les performances. Dans l'approche adaptative, les noeuds chargés n'appartiennent pas à un chemin critique, ils seront soit des feuilles de l'arbre, soit dans un chemin peu coûteux (non critique).

7.4 Performances des communications en fonction des connecteurs utilisés

Le tableau 7.2 résume les performances de communication entre deux noeuds interconnectés par le réseau de contrôle établi lors du déploiement. Il montre l'impact du type de connecteur utilisé pour réaliser cette liaison sur les performances de

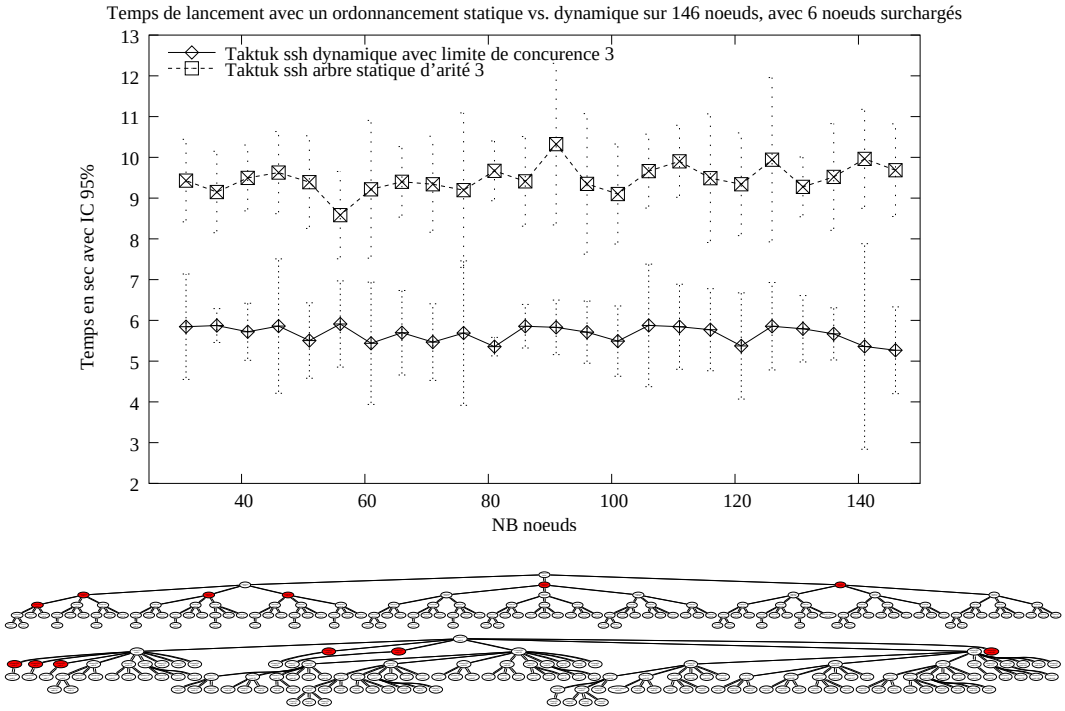


FIG. 7.7 – *Comparaison de l'approche statique (en haut) vs. dynamique (en bas) sur 146 noeuds, comprenant 6 noeuds surchargés (noeuds en noirs). Les arbres représentent les ordonnancements de lancement obtenus dans chaque approche.*

	Latence (m sec)
rsh "interne"	3
rsh "externe"	4
ssh "externe"	7

TAB. 7.2 – Latence et débit d'une liaison de contrôle en fonction du connecteur utilisé pour sa création.

communication se déroulant après la phase de déploiement. Ces différences de performances sont dues au nombre de processus par lequel transit un message entre deux noeuds. Suivant le protocole ou le type de connecteur utilisé (interne ou externe), la section 6.1.2 insiste sur le fait qu'une communication peut transiter par les processus du protocole d'exécution distante utilisé pour construire une liaison. Dans le protocole ssh, par exemple, un message allant du noeud A vers le noeud B transite via le processus client ssh présent sur le noeud A puis par le processus serveur de ssh sur le noeud B.

Ces résultats sont obtenus entre deux noeuds directement connectés l'un à l'autre. Comme le réseau de contrôle établi est arborescent, une communication entre deux noeuds est fonction de la taille du chemin dans l'arbre entre deux noeuds.
Ceci justifie le fait que l'implantation de l'approche adaptative tente de réduire le chemin d'un requête de vol en réalisant un cache local de tâches à chaque noeud (Cf. section 6.2.2).

7.5 Bilan des expérimentations

Ces expérimentations ont montré qu'il est possible de réaliser un déploiement efficace et passant l'échelle de grappes de grandes tailles avec des outils de bases coûteux comme rsh et ssh. Elles valident également les choix d'implantation qui ont été faits pour l'approche adaptative. Les résultats d'un déploiement réalisé sur un arbre de diffusion optimal révèlent sa sensibilité au taux de recouvrement ($\tau = tu/t$) utilisé pour le construire. Les surcoûts de l'implantation ne permettent pas de considérer un taux de recouvrement mesuré sans les prendre en compte. Ce constat encourage une version adaptative qui ne nécessite aucune phase de réglage dépendante de mesures ("benchmark") pour réaliser un déploiement efficace dans un milieu homogène ou hétérogène.

7.6 Conclusion

Cette étude expérimentale valide les choix de conception et d'implantation qui ont été réalisés dans la réalisation de la bibliothèque Taktuk. En effet, elle met en évidence le comportement logarithmique du coût du déploiement en fonction du nombre de noeuds. Ce qui confirme de bonnes capacités de passage à l'échelle. Les performances obtenues montrent que cet environnement de programmation peut également être utilisé dans le domaine des outils d'exploitation de grappes de grandes tailles en fournissant des temps de lancement proches de l'interactivité. Cette étude montre également que les résultats obtenus peuvent être améliorés par une réduction des surcoûts de cet environnement sur le déploiement et par une application stricte d'une stratégie "au plus tôt" dans l'implantation de l'approche adaptative.

Troisième partie

Utilisation d’une machine parallèle virtuelle

Chapitre 8

Réalisations basées sur le déploiement efficace d'une machine parallèle virtuelle

Ce chapitre présente les différents projets qui utilisent la bibliothèque Taktuk présentée dans le chapitre précédent. Ils regroupent un environnement de programmation parallèle (Inukitut), un ensemble d'outils d'exploitation (Katools) et un ordonnanceur de travaux (OAR).

8.1 Lanceur d'applications parallèles utilisant l'environnement Inuktitut : Taktukrun

Cette section introduit l'environnement pour la programmation parallèle sur grappe nommé Inuktitut. Cet environnement est développé dans le cadre du projet APACHE au Laboratoire ID-IMAG. Il inclut un support des communications et de flots d'exécution légers (couramment nommé threads). Cet environnement est directement issu de la couche de portabilité [40] de l'environnement de haut niveau ATHAPASCAN [47]. Son fonctionnement est basé sur une machine parallèle virtuelle. Le protocole de communication de cet environnement est basé sur le paradigme de message actif [84].
La présentation de cet environnement de programmation parallèle est organisée de la manière suivante. La première section exposera les objectifs de cet environnement et cadre son contexte scientifique. Une deuxième section étudiera l'architecture et la conception logicielle d'Inuktitut. Une troisième partie décrit plus en détail les différentes primitives de communication fournies par cet environnement. La quatrième partie présente sur Taktukrun qui constitue l'outil de déploiement d'Inuktitut.

8.1.1 Présentation d'Inuktitut

Il existe de nombreux environnements de programmation parallèle permettant d'homogénéiser une plate-forme physique. Cependant, ces environnements ne sont pas inter-opérables entre eux et provoquent l'émergence d'un nouveau facteur d'hétérogénéité. Il est actuellement difficile de faire coopérer efficacement deux machines parallèles virtuelles différentes. Paradoxalement, même les différentes implémentations du standard MPI [43] ne peuvent s'utiliser conjointement. Le standard MPI ne décrit que les interfaces utilisateurs et les fonctionnalités de ces environnements, le format des communications n'est ni décrit, ni formalisé. Pour résoudre ce problème d'interopérabilité, le projet IMPI [66] tente de normaliser les échanges de communications entre plusieurs implantations du standard MPI [49]. Cependant cette solution reste ciblée sur cet environnement et ne permettra pas d'interactions avec des environnements existants.

Malgré les efforts scientifiques et technologiques pour séparer les parties dépendantes ou non d'une architecture cible, le code d'une application parallèle reste dépendant de la machine parallèle virtuelle qu'il utilise. Ainsi le portage d'une application de type MPI vers un environnement comme PVM reste fastidieux. Ce besoin se fait néanmoins sentir dans certains cas, en effet suivant le type d'architecture cible et suivant les besoins de l'application parallèle, il peut être avantageux pour les performances d'utiliser des environnements de programmation parallèle différents et optimisés.
L'environnement de programmation de haut niveau ATHAPASCAN-1 [47] souhaite pouvoir utiliser différents environnements de programmation parallèle suivant les caractéristiques de l'application parallèle et de l'architecture parallèle cible. Inuktitut correspond à la nouvelle version de la couche de portabilité d'ATHAPASCAN-1, anciennement nommé ATHAPASCAN-0. Cette couche de portabilité doit permettre l'indépendance d'ATHAPASCAN-1 par rapport à l'environnement de programmation parallèle utilisé. Le changement de machine parallèle virtuelle doit non seulement être transparent à l'utilisateur d'ATHAPASCAN mais aussi pour les développeurs d'ATHAPASCAN.

Comme les motivations de ce projet sont très proches de celles qui ont conduit au développement des machines parallèles virtuelles, l'architecture logicielle retenue est similaire. L'objectif étant cette fois d'homogénéiser des machines parallèles virtuelles plutôt que des composants physiques.

8.1.2 Architecture logicielle d'Inuktitut

La figure 8.1 représente les différentes couches ou niveaux d'abstraction utilisés dans l'architecture d'Inuktitut. Le niveau le plus haut constitue l'interface de programmation (API) de cette bibliothèque. Ce niveau fournit l'abstraction d'un réseau virtuel complètement maillé. L'utilisateur peut programmer son application avec des

primitives de communications simples indépendamment de l'environnement réellement utilisé. Ces primitives de communication sont basées sur un modèle d'appel de service distant nommé message actif [84].
Le niveau intermédiaire est responsable de la gestion des différents réseaux logiques (fournis par un environnement de programmation parallèle). Ce niveau propose des services de nommage et de routage nécessaires à la construction du réseau virtuel complètement maillé du niveau supérieur.

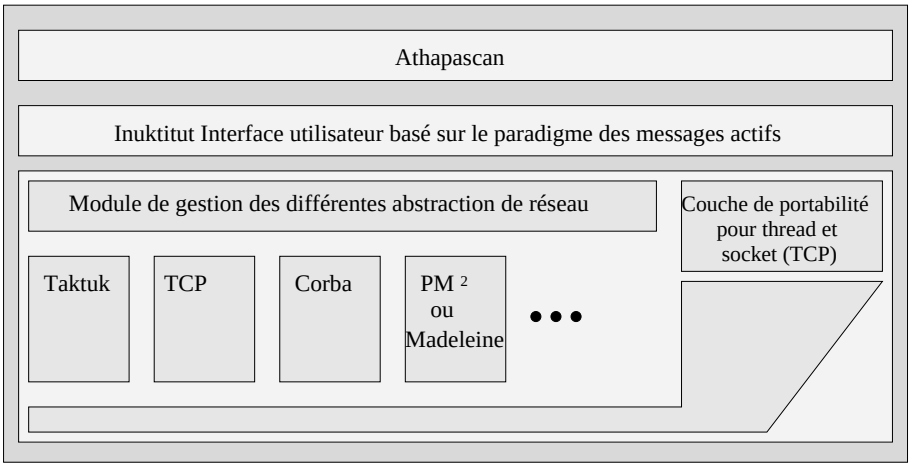


FIG. 8.1 – Présentation de l'architecture logicielle retenue pour la conception de Inuktitut.

Enfin le niveau le plus bas est responsable de l'homogénéisation des différents environnements de programmation utilisés. Par similitude ce niveau correspond à l'abstraction d'un réseau virtuel.

Comme nous l'avons vu dans les chapitres précédents, chaque environnement de programmation parallèle fournit un mécanisme de déploiement. Pour cette raison, les réseaux virtuels Inuktitut sont séparés en deux catégories. Les réseaux primaires regroupent les environnements capables de déployer une nouvelle instance de machine parallèle virtuelle. Par exemple un réseau virtuel implanté au dessus de **mpich** est un réseau de type primaire car il est déployé avec l'outil "**mpirun**" fourni avec cet environnement. Les environnements capables de s'initialiser à partir d'un ensemble de processus communiquants existants sont nommés secondaires. L'implémentation d'un réseau virtuel Inuktitut à partir d'un environnement existant peut fournir les deux types d'initialisation. Un réseau à la fois primaire et secondaire peut s'initialiser par la création d'une nouvelle instance de machine parallèle virtuelle ou par la duplication d'un réseau virtuel existant.

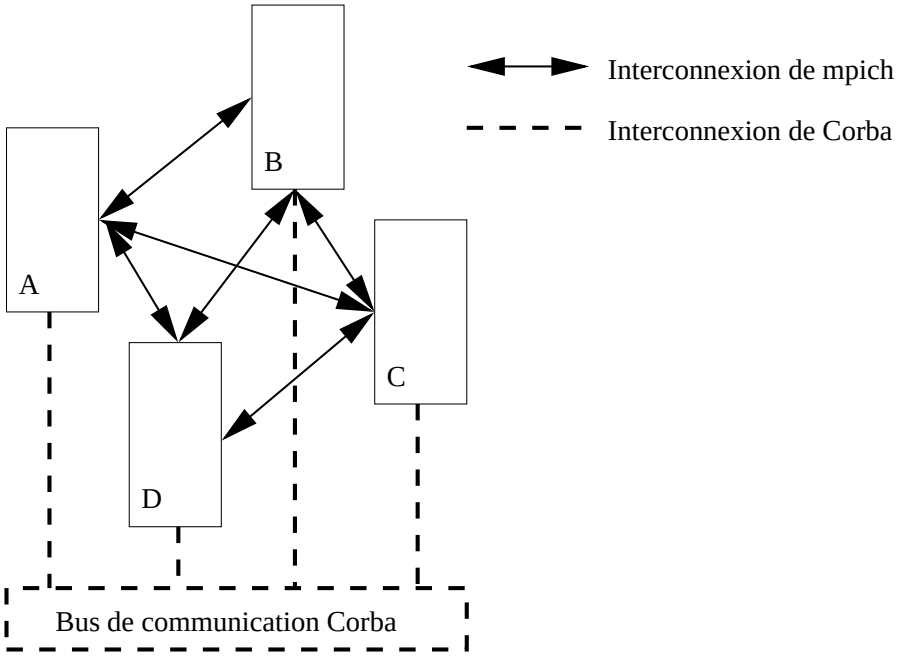


FIG. 8.2 – Présentation de l'architecture logicielle retenue pour la conception de Inuktitut.

La figure 8.2 montre le type de configuration que l'on peut obtenir en utilisant plusieurs environnements de programmation. Dans cet exemple, l'ensemble des machines sont atteintes par un réseau virtuel primaire basé sur MPICH [49]. Un réseau secondaire de type CORBA [83] est mis en oeuvre en utilisant le réseau mpich sur un sous ensemble de la grappe.

Comme toute machine parallèle virtuelle, la complexité de la réalisation réside dans le faible surcoût apporté par les différentes couches d'abstraction. Dans le cas d'Inuktitut, ce surcoût est critique puisqu'il englobe les surcoûts déjà introduits par les environnements d'exécution sous-jacents. Les surcoûts de ces différents environnements sont généralement dus au nombre de copies mémoires nécessaires à la traversée des différentes couches.
La section suivante présente les différentes primitives de communications proposées par Inuktitut. Ces primitives sont basées sur le modèle des messages actifs mais utilisent une gestion mémoire différente. Ces différentes fonctions de communication permettent de ne pas ajouter de nouvelles copies mémoires à celles qui sont éventuellement réalisées par les environnements utilisés.

8.1.3 Modèle de communication

Les primitives de communication fournies par Inuktitut repose sur le modèle des messages actifs. Ils correspondent à un appel asynchrone de service sur un noeud distant (un processus de la machine parallèle virtuelle). Ce modèle permet d'implanter facilement des modèles du type send / receive et d'appel de procédure à

distance nommé RPC ("Remote Procedure Call"). Ce modèle est très proche du RPC, à la différence que les procédures distantes sont définies explicitement par l'utilisateur et sont nommées services. Comme dans un appel de procédure distant, les services prennent des paramètres (données) en entrée, mais ils ne retournent pas de résultats.

Type de message	Gestion mémoire des données
Active **M**essage	Mémoire allouée et libérée par l'environnement (sur l'émetteur et le récepteur) ⇒ **2 copies mémoire**
Write and **S**ignal	Utilisation l'espace d'adressage de l'utilisateur ⇒ **0 copie mémoire**
Allocate **W**rite and **S**ignal	Utilisation l'espace d'adressage de l'utilisateur sur l'émetteur et Mémoire allouée par l'environnement sur le récepteur ⇒ **1 copie mémoire**

TAB. 8.1 – Récapitulatif de la gestion mémoire des différents types de messages actifs

Le tableau 8.1 présente les différentes primitives fournies. La différence de ces différents opérateurs d'appels de services distants se situe sur la gestion mémoire des données associées lors de leur émission et réception. Les appels distants de type AM ("Active Message") sont les plus simples. Les données envoyées comme paramètres du service distant sont allouées et copiées dans l'espace d'adressage du réseau virtuel Inuktitut. Ces données seront libérées à la fin de l'exécution du service par le réseau virtuel sur le coté récepteur et à la fin de l'envoi sur l'émetteur. Les appels de type AWS ("Allocate Write and Signal") ont un comportement similaire coté récepteur mais les données ne sont pas libérées à la fin de l'exécution. La portée et la durée de vie de ces données dépendent de l'utilisateur qui devra les libérer explicitement ultérieurement. Coté émetteur les données sont envoyées directement depuis l'espace d'adressage de l'utilisateur.
Ces différents appels de services sont similaires à la portée des données classiques allouées à l'intérieur d'une procédure. Si les données sont allouées sur la pile, elles sont libérées automatiquement à la fin de la procédure et si elles sont allouées sur le tas, leur durée de vie dépasse l'exécution de la procédure. Enfin, les appels de type WS (Write and Signal) utilise directement l'espace d'adressage de l'utilisateur pour envoyer et recevoir les données et exécutent le service.

Ces différents types d'appels de services distants permettent d'utiliser un large panel d'environnements parallèles existants sans ajouter de surcoûts dûs à des copies mémoires supplémentaires.

8.1.4 Lanceur intégré à Inuktitut

Le schéma 8.1 qui décrit l'architecture en couche d'Inuktitut et présente l'implantation d'un réseau virtuel avec l'environnement Taktuk. Ce réseau possède la capacité

de créer une instance de machine parallèle virtuelle.
Le réseau virtuel implanté au dessus de Taktuk fournit les fonctionnalités d'appels de services distants énoncés dans la section précédente.

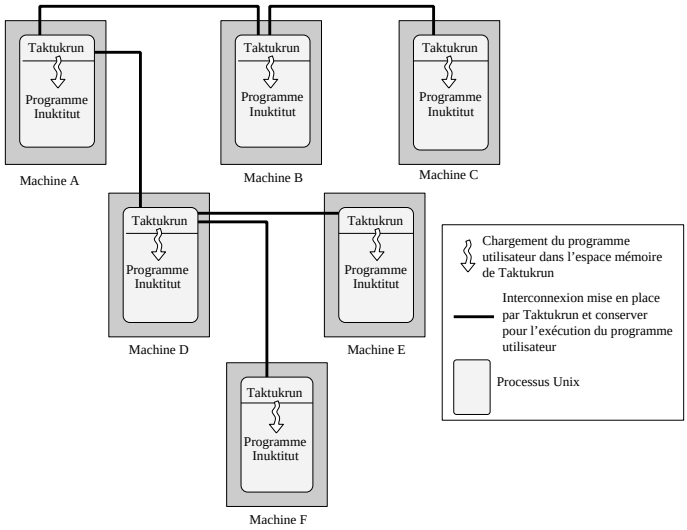


FIG. 8.3 – Présentation du fonctionnement de taktukrun.

Taktukrun est un programme qui utilise la librairie Taktuk pour effectuer le déploiement d'un application Inuktitut.

La figure 8.3 présente l'outil de déploiement de l'environnement Inuktitut. Cet outil est construit au dessus de l'environnement de déploiement Taktuk. Comme Taktuk fourni également un module de réseau virtuel Inuktitut, les applications parallèles développées avec Inuktitut peuvent se déployer d'elles mêmes, sans nécessiter l'utilisation d'un outil de déploiement. Cependant, certains types de connecteurs (dits "internes") doivent disposer de droits d'exécution privilégiés afin de réaliser la partie cliente d'un protocole de communication distante.
Cette contrainte a motivé le développement d'un outil de déploiement nommé Taktukrun. La figure 8.3 présente le fonctionnement de cet outil. Le déploiement d'un réseau de contrôle Taktuk est réalisé par la propagation de l'exécution du programme taktukrun. Ces processus disposent des droits permettant de réaliser le déploiement. Après avoir abandonné ces droits particuliers, chaque processus charge dans son espace d'adressage virtuel le programme utilisateur. Le déploiement de ce dernier est immédiat puisque les liaisons du réseau de contrôle et les différents processus de la machine parallèle virtuelle sont déjà établis.

8.2 Katools, outils d'administration pour grappes de grandes tailles

La suite d'outils d'exploitations Katools inclue principalement un outil de lancement parallèle de commande Unix [62] (Rshp) et un outil de diffusion de fichier (Mput). Cette suite fait partie de la distribution Linux Mandrake Clic [7] dédiée aux grappes de calcul. Ces outils sont largement utilisés car ils constituent la base de l'outil de diffusion est d'installation de nouveaux logiciels ("package") sur l'ensemble des noeuds de la grappe. Les Katools fournissent également une console parallèle interactive permettant de "piloter" l'ensemble des noeuds d'une grappe. Les sections suivantes présentent les deux principaux outils de cette suite : Rshp et Mput.

8.2.1 Exécution de commandes parallèles : Rshp

Rshp constitue un outil d'exploitation de grappes de grandes tailles permettant de déclencher l'exécution d'une commande (Unix) sur l'ensemble ou partie des noeuds d'une grappe. Par extension, cet outil permet également de disposer d'une console sur un ensemble de noeuds. Le processus racine de l'arbre du réseau de contrôle (Taktuk) fournit une redirection bidirectionnelle et complète des différentes commandes distantes.

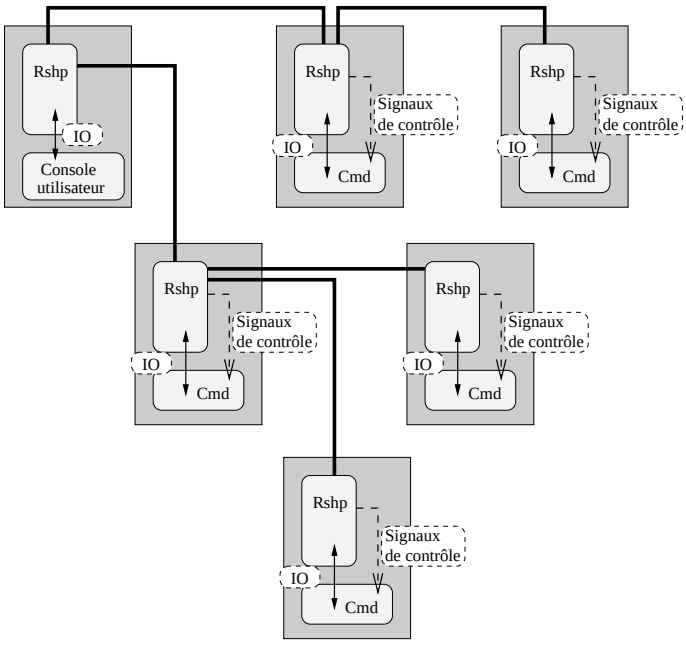


FIG. 8.4 – Présentation du fonctionnement de Rshp.

Le programme Rshp s'exécute en même temps que la commande désirée et permet de rediriger ces entrées/sorties et de lui envoyer des signaux de contrôle.

La figure 8.2.1 illustre le fonctionnement de cet outil construit au dessus de l'environnement Taktuk. Dans un premier temps, le programme Rshp propage son exécution sur les noeuds cibles grâce à l'environnement Taktuk. Une fois cette phase de déploiement terminée, les processus Rshp créent un nouveau processus pour exécuter la commande Unix spécifiée. De cette manière, les processus Rshp restent actifs pour contrôler l'exécution de ces commandes (redirection des signaux de contrôle).

8.2.2 Outil de diffusion de fichier : Mput

L'utilitaire Mput permet de réaliser la diffusion efficace d'un fichier sur un ensemble de noeuds d'une grappe. Cet outil est développé au dessus de l'environnement Inuktitut. Il utilise conjointement deux réseaux virtuels de cet environnement. Le premier type de réseau utilisé est Taktuk qui permet de déployer rapidement un réseau de processus "Mput" sur l'ensemble de noeuds cibles. Le deuxième réseau virtuel utilisé est un réseau implanté au dessus du protocole standard TCP/IP. La particularité de ce réseau et de construire dynamiquement une interconnexion logique lors de l'exécution. Ceci permet d'initialiser seulement les liaisons logiques qui seront utilisées par l'application. Ces liaisons seront initialisées sous la forme d'une chaîne permettant de diffuser efficacement le fichier sur l'ensemble des noeuds. L'initialisation de ces liaisons est effectuée avec le service de communication offert par le réseau de contrôle établi par Taktuk. Le fichier est diffusé par paquets de 2Ko ce qui permet de former un pipeline et de s'approcher du débit théorique de 11Mo/sec du réseau LAN 802 100Mb/s sous-jacent.
La figure 8.2.2 illustre le fonctionnement de cet outil et l'utilisation conjointe de deux réseaux virtuels Inuktitut. La courbe 8.2.2 présente les performances de cet outil lors de la diffusion d'un fichier sur 200 noeuds.

8.3 Allocateur de ressources OAR

La popularité des grappes de PC et leurs large diffusion augmentent les besoins en outils souples et robustes pour leur administration et leur exploitation. Le projet de distribution Linux CLIC [2] qui regroupe un ensemble d'outils pour grappes est un exemple de réponse. Parmi ces besoins, un élément central pour l'exploitation des grappes est la disposition d'un gestionnaire de ressources. Il permet l'allocation de ressources processeurs aux tâches séquentielles ou parallèles soumises par les utilisateurs puis leur lancement ainsi que leur contrôle pendant leur exécution.

Il existe un grand nombre de gestionnaire de travaux pour grappes et machines parallèles. Parmi les plus connus nous pouvons citer PBS/OpenPBS[11], LSF[88], NQS[20], LoadLeveler[53], Condor[58], Glunix[48]. Les études [52] et [23] répertorient les principaux systèmes ainsi que leurs fonctionnalités. Actuellement peu de travaux de recherches s'intéressent directement à ces systèmes, bien que des études récentes aient été menées sur le lancement et le déploiement d'applications à grande échelle qui sont des éléments importants dans la conception de ces systèmes [46]. De plus,

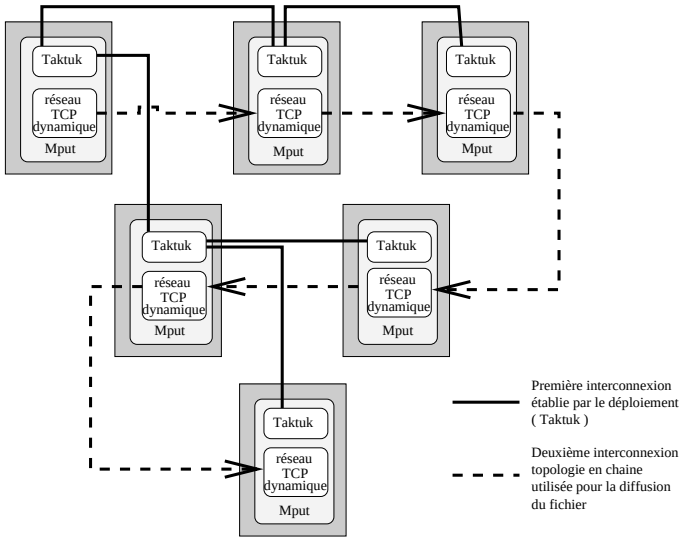


FIG. 8.5 – Présentation du fonctionnement de Mput.

Mput utilise deux réseaux virtuels Inuktitut, un réseau arborescent mis en oeuvre par le déploiement, est utilisé pour initialiser un réseau chaîné permettant de diffuser le ou les fichier(s) sur tous les noeuds atteints par le déploiement.

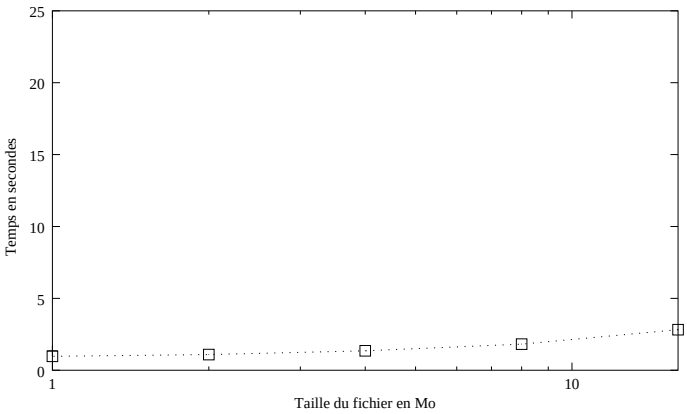


FIG. 8.6 – Performances obtenues pour la diffusion d'un fichier de 1 à 16 Mo sur 200 noeuds.

dans l'ensemble, ces systèmes sont d'une conception ancienne et ne répondent pas complètement à certaines situations récentes (passage à l'échelle, flexibilité, forte extensibilité, robustesse).

Dans la suite de cette section, nous motivons l'étude d'un nouveau système de gestion de ressources pour grappe, baptisé OAR, et posons les objectifs recherchés. Dans la section 8.3.2, nous présentons les choix de conception puis l'architecture générale en section 8.3.3. Nous présentons ensuite l'évaluation de notre système d'une part en comparaison au système OpenPBS et d'autre part en présentant une extension simple pour une exploitation de type *Global Computing*. Finalement nous concluons sur quelques perspectives.

8.3.1 Motivations, objectifs et systèmes apparentés

La figure 8.7 représente l'architecture classique d'un système gestionnaire de ressources. Les principales fonctions qui les composent sont : un module de soumission qui éventuellement exerce un contrôle sur la validité de la requête, un module d'ordonnancement associé à un mécanisme d'appariement (*matching*) de ressources et un module d'exécution qui contrôle le déroulement des tâches. A cela se rajoute des modules d'administration pour le suivi d'activité (*log*), de comptabilité (*accounting*), et de surveillance de ressources (*monitoring*).

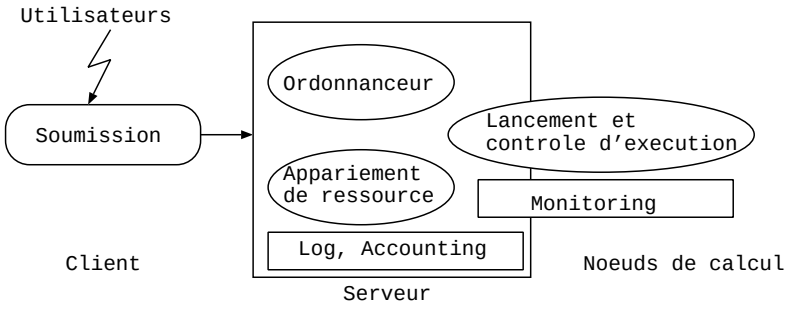


FIG. 8.7 – Architecture générale d'un gestionnaire de ressources pour grappe et machine parallèle.

Les principaux systèmes existants [52] ont été développés dans les années 90. Pour la plupart de ces systèmes, les concepteurs ont opté pour une intégration du plus grand nombre de fonctionnalités afin de répondre à un large ensemble de situations d'exploitation. L'accumulation de fonctionnalités a conduit à des logiciels de grande taille et d'une complexité importante (150 000 lignes de code pour OpenPBS, cf paragraphe 8.3.4.1). Il s'ensuit que les niveaux de robustesse et de performance sont difficiles à maîtriser dans ces conditions. De plus, ce choix d'une architecture globalement monolithique rend toute extension imprévue ou toute modification

potentiellement problématique.

Le système PBS, par exemple, possède une API pour faciliter l'implantation de politique d'ordonnancement. Bien que cette API soit très riche en fonctionnalités elle n'est que très rarement exploitée. Ainsi, le projet Maui scheduler [8] qui a pour objectif le développement de politiques d'ordonnancement avancées n'utilise le système PBS que comme système de soumission de tâches de type FIFO, laissant de coté l'API en question. Nous pouvons également remarquer que l'ordonnanceur Maui est devenu lui-même très important en taille de code. Ceci montre qu'il est difficile de maîtriser la complexité des logiciels de ce domaine.
Finalement la définition d'une API *universelle* pour le développement de modules d'ordonnancement reste un problème ouvert.

Une autre expérience basée sur le système PBS est celle du passage à l'échelle menée autour de la plate-forme Cplant [27]. Une extension a été apportée sous la forme d'un module d'exécution et de contrôle fournissant une meilleure capacité de passage à l'échelle. La critique de cette expérience est similaire au cas précédent : l'extension se faisant par ajout de code et modification du système de base, la complexité globale du système est augmentée et la robustesse (voire la performance) de l'ensemble est fragilisée.

Récemment, une nouvelle génération de systèmes a fait son apparition, par exemple le système Maui Scheduling Molokini Edition [9]. Dans ce système, une base de donnée a été introduite pour maintenir certaines informations sur les utilisateurs et leur compte mais surtout comme solution de sauvegarde pour l'ensemble de l'état du système (ex. tâches en cours). Le même choix apparaît dans des systèmes "pair à pair" apparentés comme XtremWeb [42], qui est un système pour l'exploitation des ressources inutilisées des machines volontaires (PC) connectées sur Internet. Par ailleurs, du coté des grappes de grandes tailles, sont apparus des systèmes de lancement et de contrôle d'exécution extensibles en nombre de noeuds (*scalable*) [46, 62, 29, 67]. Ils peuvent, pour certains, soit s'intégrer dans des gestionnaires existants, soit donner lieu au développement de nouveaux gestionnaires.
Ainsi le projet Storm [46] propose des fonctions de déploiement, de lancement et de contrôle d'application qui reposent sur des capacités de communication globale efficace fournies par un matériel réseau spécialisé.

L'exploitation d'une grappe de 225 PC, le projet *icluster*, a permis à notre laboratoire d'acquérir une expérience à la fois du point de vue des utilisateurs et des concepteurs de logiciels d'exploitation de ces plate-formes.

A coté de ces nouveaux systèmes et de l'évolution des plate-formes dans leurs tailles, nous pouvons remarquer une évolution des besoins des utilisateurs. Par exemple, en terme de réactivité à travers la soumission de tâches interactives lors de phase de développement. Ce besoin n'est pas récent mais des garanties sur sa satisfaction deviennent nécessaires. Un autre exemple de besoin nouveau est le support des applications multi-paramétriques. Celles-ci se caractérisent par un très

grand nombre d'exécutions d'une même application avec un jeu de paramètres en entrée différent. Les problèmes posés ici sont liés à la gestion de l'ensemble des tâches à effectuer, à la maîtrise des erreurs (qui deviennent inévitables avec le nombre des tâches envisagées) et à la gestion des résultats (rapatriement, exploitation).

Du point de vue du concepteur de système, le principal besoin porte sur l'extensibilité du système. Les exemples pour lesquels une telle extensibilité est profitable sont nombreux : pour tester de nouvelles politiques d'ordonnancement, intégrer de nouveaux mécanismes d'administration ou encore gérer des ressources dynamiques (qui se retrouvent dans les contextes d'exploitation de type *Global Computing* [58, 42] ou Cluster à la demande [34]). L'architecture logicielle de ces systèmes induit une modularité importante (Cf. figure 8.7). Cependant, l'indépendance entre les différents composants reste difficile à maintenir au cours de leur implantation. Cette indépendance garantie la facilité d'extension de ces systèmes, pour cela les intéractions entre les différents composants doivent être limitées, voir inexistantes. Jusqu'à présent il ne semble pas exister de tel système de gestion de ressources et chaque extension nécessite d'importantes modifications.

Afin de répondre à ces besoins, nous avons défini les objectifs que devrait remplir un gestionnaire de ressources permettant à la fois d'offrir un service de gestion effectif et d'être une plate-forme de recherche. Les qualités prioritaires que nous avons considérées au cours du développement du système OAR sont la facilité d'extension, la capacité de passage à l'échelle et la robustesse. Pour atteindre cet objectif, nous avons choisi de réduire le nombre de fonctionnalités proposées par OAR par rapport à ses prédécesseurs, pour nous concentrer sur la majorité des besoins classiques des sites de production. Nous avons également conçu ce système avec en tête une architecture sous-jacente homogène, ce qui rend OAR moins adapté aux grappes hétérogènes (qui tendent à être moins répandues).

8.3.2 Choix de conception

Pour répondre à nos objectifs de simplicité, d'extensibilité et de robustesse, nous avons choisi de faire reposer la conception du système sur plusieurs composants logiciels de haut niveau : une base de donnée relationnelle (*MySQL*[10]) et un outil générique d'administration pour grande plate-forme (*Taktuk*[64, 63, 62]).

Ce choix permet de résoudre en partie le problème de la robustesse. En effet la base de donnée utilisée est largement reconnue pour sa stabilité. De plus la charge prévue se trouve être modeste (cf. section 8.3.5). Pour l'outil d'exploitation générique, l'expérience acquise est plus réduite, par contre un nombre réduit de fonctionnalités sont utilisées et compte parmi les plus éprouvées. Finalement nous jugeons que les risques de dysfonctionnement des fonctionnalités réalisées par ces outils sont très faibles, ce qui permet de disposer d'une base logicielle fiable et solide pour le reste du système.

L'originalité du choix d'une base de donnée dans ce contexte se situe dans la place centrale qu'elle occupe au sein de l'architecture du système. En effet, contrairement

aux systèmes Maui Molokini[9] ou XtremWeb[1][42] qui sont de conception classique, nous avons choisi d'ouvrir pleinement l'état interne du système en le maintenant entièrement dans la base de données et en utilisant celle-ci comme seul mode d'échange d'informations. La conséquence directe est qu'il n'y a pas de spécification d'API (*Application Program Interface*) pour les différents modules constituant le reste du système mais plutôt une spécification d'architecture de la base de données. Chaque module interagit dans le système via des requêtes SQL qui peuvent être locales ou distantes. Cette approche assure une modularité extrême à l'ensemble du système ainsi qu'une totale indépendance dans le choix du langage d'implantation de chaque module (la plupart des langages de programmation possèdent des bibliothèques d'accès aux bases de données SQL). Grâce à cette approche, la spécification du système se réduit au diagramme d'état d'une tâche, à la signification des tables et de leurs champs et aux fonctions des différents modules. L'ensemble de ces caractéristiques assurent au système à la fois une grande simplicité et une forte extensibilité.

Dans une perspective de passage à l'échelle, notre choix de composants de haut niveau est une fois de plus un point clé. Pour l'échange de données au sein du système et la soumission massive de requêtes, nous pouvons nous reposer sur des systèmes de base de données prévues pour soutenir de fortes charges de travail. Pour le déploiement d'application parallèle et l'administration du système sur grappes massives (plusieurs milliers de noeuds), nous nous reposons sur le composant d'administration spécifiquement conçu avec l'environnement *Taktuk* pour rester efficace sur un grand nombre de nœuds.

En ce qui concerne le développement des modules, nous avons retenu le langage de script interprété Perl. Ce langage possède nativement des structures de donnée de haut niveau comme les tables associatives et le support des expressions régulières. Ces capacités permettent un cycle de développement très court et renforcent la concision et la simplicité du code. Bien que n'ajoutant pas un surcoût démesuré, si le fait d'utiliser un langage de script pose un problème de performance, il est tout à fait possible et simple de réécrire un ou plusieurs modules dans un langage plus efficace (du moment qu'il dispose d'une bibliothèque d'interface avec une base de données SQL). Toutefois, ce choix de langage privilégie le critère de faible complexité logicielle par rapport à celui de la performance sans pour autant remettre en question la capacité de passage à l'échelle du système. Cette faible complexité est un élément favorisant à la fois la robustesse ainsi que l'extensibilité.

8.3.3 Architecture générale

Le système OAR [31] est bâti selon le modèle présenté en section 8.3.1 et repris par la figure 8.7. Chaque élément fonctionnel de ce modèle est implémenté par un module indépendant. Ces modules communiquent par l'intermédiaire d'une base de données relationnelle et un module central est chargé d'orchestrer le fonctionnement de l'ensemble. Les utilisateurs communiquent avec le système par le biais de commandes de

[1]Cette comparaison s'applique à la première version d'XtremWeb. La dernière version utilise, dans certain cas, directement la base de donnée comme moyen d'intéraction entre modules.

soumission, d'annulation, de monitoring, ... (à la manière de PBS).

Les spécifications nécessaires à la définition de l'architecture du système dans son ensemble sont le diagramme d'états des tâches et les différents tables et champs de la base de données. La figure 8.8 présente le diagramme d'état des tâches soumises au système. L'insertion, la retenue et la phase d'ordonnancement se déroulent dans les états *waiting* et *hold*. Les états suivants sont associés aux phases d'exécution. Un état de demande de retrait/arrêt de la tâche (non représenté) se positionne en parallèle, il est traité en priorité.

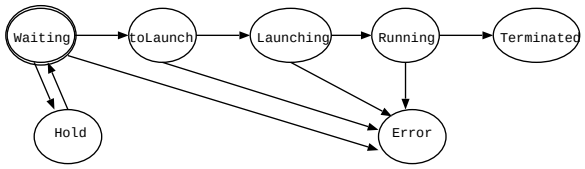


FIG. 8.8 – Diagramme d'état des tâches dans le système.

Nous ne détaillons pas l'ensemble des tables de la base de données. Deux de ces tables sont essentielles au cœur du système : la table décrivant les tâches, les ressources demandées ainsi que les dates des principaux événements (soumission, départ/fin d'exécution) et celle décrivant la plate-forme d'exécution ainsi que son état. Parmi les autres tables constituant le système, nous retrouvons celles stockant les informations pour l'ordonnancement, les règles d'admission et l'appariement de ressources.

8.3.3.1 Soumission de tâche

Du point de vue de l'utilisateur, le système OAR fonctionne à la manière de PBS : l'interaction avec le système s'effectue par l'intermédiaire de commandes indépendantes pour la soumission (commande *qsub*), l'annulation (commande *qdel*) ou l'observation de travaux (commande *qstat*). Ces commandes sont découplées au maximum du reste du système, elles transmettent ou récupèrent les données dont elles ont besoin en accédant directement à la base de données et interagissent avec le système par l'intermédiaire de simples notifications envoyées au module central.

La figure 8.9 présente le déroulement d'une soumission d'une tâche. Celle-ci débute par une connexion à la base de donnée suivie du rapatriement des règles d'admission. Ces règles servent à fixer les paramètres manquants (non fournis par l'utilisateur) liés à la soumission et à vérifier la validité de la requête. Parmi les paramètres pris en compte à la soumission, nous retrouvons un identifiant de file d'attente, un temps limite, un contrôle des droits d'accès, etc. Les règles se présentent sous la forme

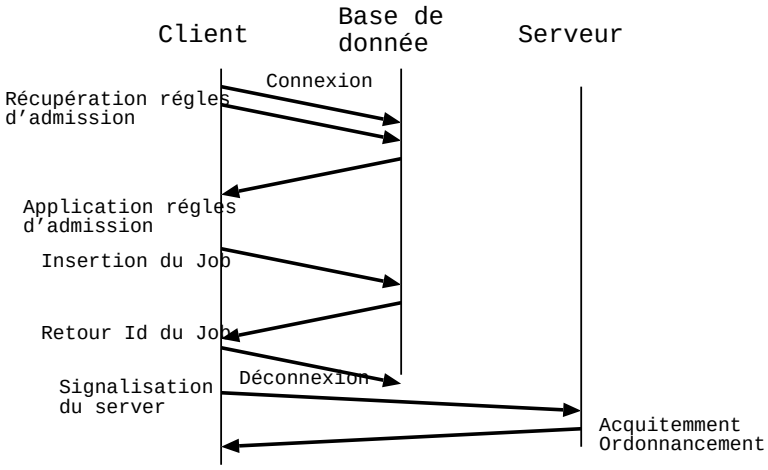


FIG. 8.9 – Déroulement de la soumission d'une tâche.

de ligne de code en langage Perl et peuvent être remplacées par l'appel à un exécutable. La tâche est ensuite insérée dans la base qui, en retour, lui donne son identifiant correspondant à la clé nouvellement créée de l'index de la table des tâches. Après la déconnexion, le module de soumission notifie le module central de l'arrivée d'une nouvelle tâche. Nous avons choisi d'implanter ce mécanisme de notification avec des `sockets` TCP.

8.3.3.2 Module central

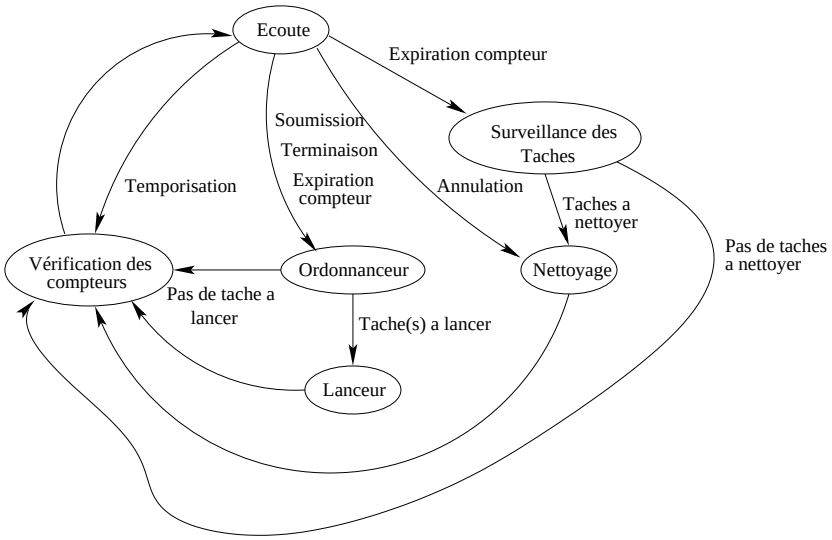


FIG. 8.10 – Automate du module central du système de gestion de ressource OAR.

Le principal souci lors de la conception générale de OAR a été de garantir simultanément la réactivité et la robustesse du système. Ce sont ces objectifs qui

ont guidé notre choix dans la structure du module central (figure 8.10). Ce module central est constamment à l'écoute de notification pouvant provenir de chacun des autres modules. C'est ce système de notification qui assure une bonne réactivité du système : dès qu'une commande ou un module ajoute des informations dans la base de données, le module central reçoit une notification et met en œuvre le traitement approprié. La robustesse, quand à elle est assurée par le découpage entre transmission des informations et notification. Pour qu'une requête soit traitée, il faut qu'elle transmette ses informations à la base de données et que le module de traitement correspondant soit lancé. Autrement dit, la centralisation des notifications ne constitue pas un goulot d'étranglement si le module central garantit le lancement régulier des modules de traitement : la tolérance du système à la charge de travail est déportée sur le système de base de données. Comme nous pouvons le remarquer sur la figure 8.10, toute phase d'écoute des notifications dans le module central est immédiatement suivie d'une phase de traitement puis de mise-à-jour de compteurs internes afin, le cas échéant, de pouvoir lancer certaines opérations à intervalles réguliers. Dans le cas d'une surcharge du système, bien que l'afflux de notifications ne puisse pas être intégralement être pris en compte, le mécanisme des compteurs garantit un bon fonctionnement du système.

De façon plus détaillée, le module central est axé sur un état d'écoute (figure 8.10). De cet état d'écoute, toute nouvelle notification fait passer le module dans un état de traitement. Selon le cas il s'agit d'un ordonnancement suivi, si besoin d'un lancement, d'un nettoyage des tâches ne devant plus se trouver sur le système ou bien d'un contrôle de l'expiration du temps d'exécution alloué aux tâches. Entre deux phases de traitement, le passage par l'état de vérification des compteurs permet de réclamer à intervalle régulier le passage dans les états d'ordonnancement ou de contrôle de l'expiration du temps d'exécution alloué aux tâches.

8.3.3.3 Ordonnancement

A l'heure actuelle, le comportement du module d'ordonnancement est similaire à celui de PBS dans sa configuration par défaut (FIFO). Nous avons implanté l'appariement de ressources. Une requête peut demander des noeuds ayant des propriétés spécifiques, comme un commutateur réseau, connectant la machine, particulier. Nous n'avons pas encore implémenté le remplissage des emplacements laissés vacants lorsque des tâches réclamant beaucoup de processeurs sont en attente (*backfilling*). En revanche, le support de tâches de type "meilleur effort" (*Best Effort*) est déjà disponible.

La structure générale de l'ordonnanceur est décrite dans la figure 8.11. Lors de leur soumission, les tâches sont rattachées à une file d'attente. Chaque file d'attente est caractérisée par une priorité et possède son propre ordonnanceur. L'ordonnancement de l'ensemble est assuré par un module d'ordonnancement entre files d'attente (ou méta-ordonnanceur), qui détermine la file dans laquelle il faut chercher une tâche en attente et lance l'ordonnanceur de cette file. Comparée à une approche centrée sur les tâches (ordonnancement d'un ensemble de tâches avec priorité, à

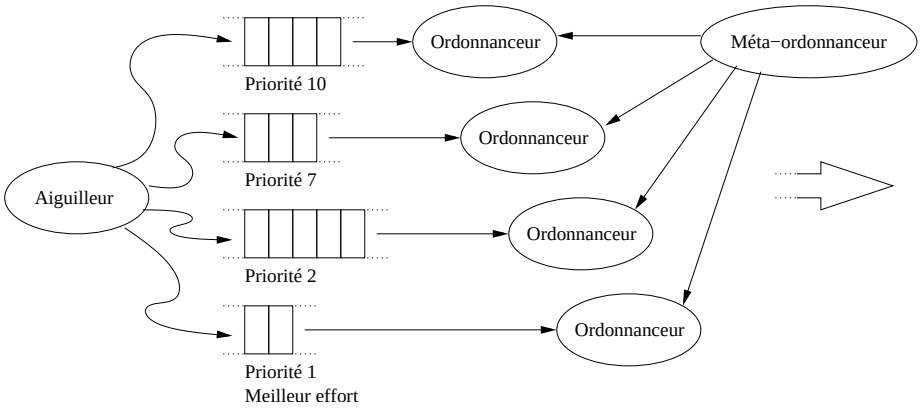


FIG. 8.11 – Structure d'un ordonnanceur dans OAR (exemple avec 4 files).

l'image de Maui), cette approche perd une vision globale du problème dont pourrait tirer parti un ordonnanceur agressif (optimisant par exemple le rendement global par programmation linéaire). Cependant, ce découpage représente le meilleur compromis entre simplicité et richesse d'expression. D'une part, la conception et la compréhension de l'ordonnanceur deviennent extrêmement simples (politique de choix d'une file et politique de choix d'une tâche dans une file) et d'autre part, l'administration de l'ensemble du système est très intuitive (gestion des tâches au niveau des files : interruption/reprise d'une file, déplacement des tâches entre files, etc.).

A l'heure actuelle le méta-ordonnanceur est un simple ordonnanceur à base de priorités, il donne la main à l'ordonnanceur de la file non vide de plus haute priorité. De même, l'ordonnanceur en question est pour l'heure un simple algorithme FIFO (premier arrivé, premier servi). Cependant nous prenons en compte les tâches de type "meilleur effort". Ces tâches sont destinées à occuper les ressources inutilisées et sont interruptibles, sans reprise, si une autre tâche a besoin de ce noeud. Notre ordonnanceur, grâce à une modification de l'allocateur de ressources, est à même de déterminer un ensemble de tâches "meilleur effort" à interrompre en cas de besoin (avec éventuellement une priorité entre de telles tâches). Cette fonctionnalité, requise dans les systèmes de calcul global (ex : Condor [58], XtremWeb [42]) nous a permis de démontrer la simplicité d'extension de OAR, y compris dans la prise en charge de fonction dépendant de plusieurs modules (cf. 8.3.4.1).

8.3.4 Evalutations de OAR

Pour démontrer la validité de notre approche, nous avons effectué 3 types d'évaluations. Le premier est une comparaison qualitative de la complexité logicielle de plusieurs systèmes de gestion de ressources. Le deuxième porte sur les capacités d'extension du système OAR. Nous terminons par une comparaison des performances de OAR avec le système OpenPBS.

8.3.4.1 Complexité

Afin d'évaluer qualitativement la complexité logicielle des différents systèmes de gestion de ressources, nous nous fondons sur les indicateurs suivants : le langage majoritairement utilisé, le nombre de fichiers sources et le nombre total de lignes de code source. Dans les décomptes, nous avons seulement considéré les fichiers nécessaires au fonctionnement du système à l'exclusion de tous les fichiers relatifs à la documentation, aux interfaces graphiques, aux outils d'installation, etc. Le tableau 8.2 regroupe les données pour ces différents indicateurs dans les systèmes suivants : OpenPBS[11], Maui Scheduler [8], Maui Scheduler Molokini Edition [9], XtremWeb[42], OAR. Dans le cas des systèmes Maui Scheduler et OAR, les fonctions effectives d'allocation de ressources (pour Maui) et de monitoring (pour OAR) sont assurées respectivement par OpenPBS et *Taktuk*. Il est donc nécessaire d'ajouter la complexité supplémentaire due à ces outils pour obtenir la complexité de l'ensemble du système.

	OpenPBS	Maui Scheduler (+ OpenPBS)	Maui Scheduler Molokini	*Taktuk*	OAR (+*Taktuk*)	XtremWeb
Version	2.3.16	3.2.5	1.5.2	3.0	-	1r2-rc5
Langage (majoritaire)	C	C	Java	C++	Perl	java
Nb fichiers sources	350	142	116	120	30	235
Nb lignes sources	148000	142000	25000	19500	4500	45000

TAB. 8.2 – Éléments de comparaison logicielle pour différents gestionnaires de ressources .

Ces résultats révèlent que les systèmes de première génération, que sont OpenPBS et Maui Scheduler, sont de loin plus complexes que Maui Molokini ou OAR. Ceci est dû d'une part au choix du langage d'implémentation du système et d'autre part au nombre de fonctionnalités prises en charge directement par celui-ci. Dans le cas d'OpenPBS, la complexité logicielle du système est d'autant plus importante que l'accent a été mis sur le nombre de fonctionnalités offertes par le système, la prise en compte de l'hétérogénéité et le choix du langage C pour l'implémentation de ces fonctions. A l'inverse, des systèmes comme Maui Scheduler Molokini ou OAR (+*Taktuk*) démontrent que la complexité d'un gestionnaire de ressource n'est pas intrinsèque à ce type d'applications mais dépend fortement des choix de conception.

Ces deux systèmes possèdent des tailles de codes comparables. Notons que dans le cas de OAR, le composant *Taktuk* représente la plus grande partie du code du système.

Cependant, bien que *Taktuk* soit nécessaire au fonctionnement de OAR, il est développé indépendamment de ce dernier. Autrement dit sa complexité est restée complètement cachée lors du développement de OAR lui-même. La différence notable entre Maui et OAR vient du nombre d'*API* qu'il est nécessaire de manipuler pour modifier ou développer une nouvelle politique d'allocation dans Maui. Dans notre système, ceci se fait simplement par la construction de requêtes à une base de donnée *SQL*. Ainsi OAR est tout à fait représentatif de la nouvelle génération de gestionnaires de ressources : il démontre qu'une conception extrêmement simple est possible pour ce type d'application à condition de tirer parti des outils logiciels de haut niveau désormais disponibles.

8.3.4.2 Extension : *Global* ou *Desktop computing*

De plus en plus de systèmes sont mis en oeuvre dans les Intranet d'entreprise, les grappes ou l'Internet afin d'exploiter les ressources matérielles durant leurs périodes d'inactivité, on parle alors de *Global* ou *Desktop computing*. Généralement, ceci se réalise par l'ajout d'un mécanisme de détection de ressources libres qui déclenche lui-même la récupération et l'exécution de tâches [79][42]. Lorsque la ressource hôte est de nouveau réclamée pour son utilisation normale, elle est alors restituée (éventuellement au détriment d'une tâche de *Desktop computing* en cours de calcul, on parle alors de tâche "meilleur effort"). Généralement, ces systèmes s'attachent à être les plus transparents possible. Cependant, dans le cas de l'exploitation d'une grappe, la transparence n'est pas nécessairement souhaitée : pour permettre une comptabilité des ressources consommées ou pour effectuer des choix d'ordonnancement plus adaptés, il est souvent nécessaire d'impliquer le système de gestion de ressources dans la mise en place du *Desktop computing*.

Cette extension a été implémentée dans OAR en ajoutant une propriété aux travaux de type "meilleur effort". Cette propriété est positionnée par l'aiguilleur, lors de l'insertion de la tâche dans la base (par exemple, en précisant lors de la soumission l'appartenance de la tâche à une file d'attente de type "meilleur effort"). Il est également nécessaire d'être à même d'annuler les tâches "meilleur effort" en cours d'exécution sur des nœuds occupés. Nous avons choisi d'implémenter la demande d'annulation des tâches dans les fonctions qui fournissent les noeuds disponibles aux ordonnanceurs et l'annulation elle-même dans le module générique de nettoyage lancé habituellement lors de l'expiration du temps alloué à une tâche. Ainsi, dans cette nouvelle version des fonctions de détermination des ressources libres, l'information doit remonter jusqu'au module central afin que celui-ci lance le module de nettoyage en question. L'inconvénient majeur de cette approche est que l'information doit transiter à travers divers modules du système. Ceci se traduit par l'ajout d'un nouvel état dans l'automate du module central (figure 8.10) et correspond à une nouvelle séquence d'exécution prenant en charge la libération des noeuds. Néanmoins, cette approche nous permet de conserver à la fois notre découpage initial des fonctions réalisées par les différents modules, l'indépendance totale des ordonnanceurs vis-à-vis du *Desktop computing* et la vision de haut niveau des tâches

de type "meilleur effort". En outre, elle illustre parfaitement les facilités offertes par OAR pour l'implémentation de fonctions transversales dans l'ensemble du système.

Avec cette approche, il est possible de définir une politique de sélection des tâches à annuler, comme un tri selon leur date de démarrage (afin de stopper les tâches les plus jeunes et essayer de terminer les plus anciennes) ou selon le nombre de noeuds occupés (pour tenter de minimiser le nombre de tâches "meilleur effort" à annuler).

Une fois de plus, les modifications à apporter au système sont simples et peu nombreuses. Ceci est entièrement la conséquence de la haute modularité, de l'ouverture (par la base de données) et du haut niveau (obtenu grâce à Perl) du système OAR.

8.3.5 Performances

Deux plates-formes ont été utilisées pour les tests de performance. La première est constituée de 5 PC (bi-processeurs Xeon 2,4Ghz 512Mo RAM, réseau ethernet 1 Gbits), par la suite elle est dénommée plate-forme Xeon. La deuxième est la plate-forme Icluster (225 processeurs PIII 733, 256Mo, réseau ethernet 100 Mbits). Lors des tests, 119 nœuds de calcul étaient disponibles. Pour l'ensemble des tests sur la plate-forme Xeon, le client test, le module central et la base de donnée pour OAR ou le démon serveur pour OpenPBS sont exécutés sur la même machine biprocesseur. Sur la plate-forme Icluster, le client est exécuté sur une nœud distinct et les aspects serveurs (module central, base de donnée et démon OpenPBS) sur une machine PIII 866Mhz, 256Mo de mémoire. Les tâches soumises correspondent à la demande d'exécution d'un processus exécutant la commande système *date*, le nombre de nœuds réservés dépend de l'expérience menée. Dans les tests nous limitons le nombre maximum de tâches soumises simultanément à 100. Cette limite est largement suffisante si on considère des soumissions de tâches directement par les utilisateurs, et pour des soumissions automatiques (via des scripts ou des agents logiciels), elle reste convenable. De plus au-delà de cette limite, nous avons eu des problèmes de stabilité avec le système OpenPBS.

La figure 8.12 montre les performances obtenues sur la plate-forme Xeon. Le test consiste à mesurer le temps moyen de réponse pour une tâche demandant un nœud en fonction du nombre de tâches soumises simultanément. Le temps de réponse unitaire mesuré est la différence entre la date de terminaison de la tâche et la date de soumission. La variance est faible et non représentée sur le graphique. Les différentes courbes correspondent au temps obtenu par OpenPBS, puis OAR avec et sans vérification de l'état des nœuds avant l'exécution de la tâche. La vérification consiste à tester l'état du nœud et son accessibilité en effectuant une première exécution distante d'une commande *vide* via *rsh* ou *ssh*. Ainsi, la chaîne de lancement est testée jusqu'à la couche application.

Sur cette plate-forme récente on observe que le niveau de performance atteint par notre système est meilleur que celui obtenu avec OpenPBS. Malgré une approche de haut-niveau et l'emploi d'un langage interprété, la montée en charge pour notre système est meilleure. De plus la base de donnée, pour le traitement de 10 tâches, reçoit

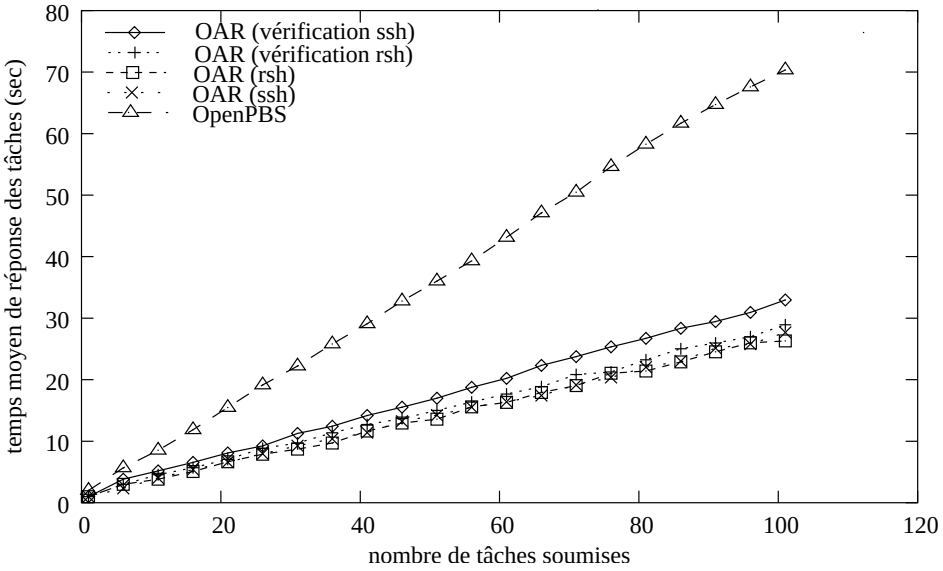


FIG. 8.12 – Temps de réponse moyen d'une tâche demandant 1 nœud en fonction du nombre de tâches soumises pour les systèmes OpenPBS et OAR (avec et sans vérification des états des noeuds) sur la plate-forme Xeon (4 nœuds de calcul).

350 requêtes *SQL* soit environ 70 reqs/sec sur la plate-forme Xeon. Cette charge est une très faible par rapport à la capacité de traitement de la base (>3000 reqs/sec).

Cela montre un bon emploi de la base de donnée et une bonne architecture du système. De plus, le surcoût d'une vérification systématique de l'état des nœuds est faible.

La figure 8.13 présente les mêmes tests que précédemment sur la plate-forme Icluster entre OpenPBS et OAR sans vérification d'état. Sur cette plate-forme d'ancienne génération, les coûts de l'emploi d'une approche de haut-niveau n'ont pas pu être amortis. Le taux de remplissage des nœuds de calcul est moins bon que celui d'OpenPBS. Nous n'avons pas détaillé les différents coûts, mais sur cette plate-forme le coût d'un lancement d'un script Perl est important puisqu'il nécessite une phase de pré-compilation (0,3 seconde pour débuter une commande de soumission). Or ce type de coût intervient au lancement de chaque module à chaque cycle du module central.

La figure 8.14 présente le temps de réponse moyen d'une tâche en fonction du nombre de noeuds demandés. A chaque point 20 tâches sont soumises simultanément. Le système OAR, sans vérification d'état possède de bonnes performances. Avec vérification d'état, le plus faible niveau de performance par rapport à OpenPBS reste convenable avec *rsh*. Mais, il est nettement moins bon avec le protocole *ssh*. Il faut noter qu'OpenPBS n'effectue pas de vérification systématique au niveau du protocole d'exécution sur l'ensemble des nœuds attribués avant le lancement de la tâche. Cette vérification systématique confère au système OAR un niveau de qualité de service élevé puisque chaque nœud alloué est testé jusqu'au niveau applicatif de lancement d'exécution à distance. En terme de coût, on retrouve les mêmes constatations que précédemment, mais ils sont atténués lorsque le nombre de ressources demandées

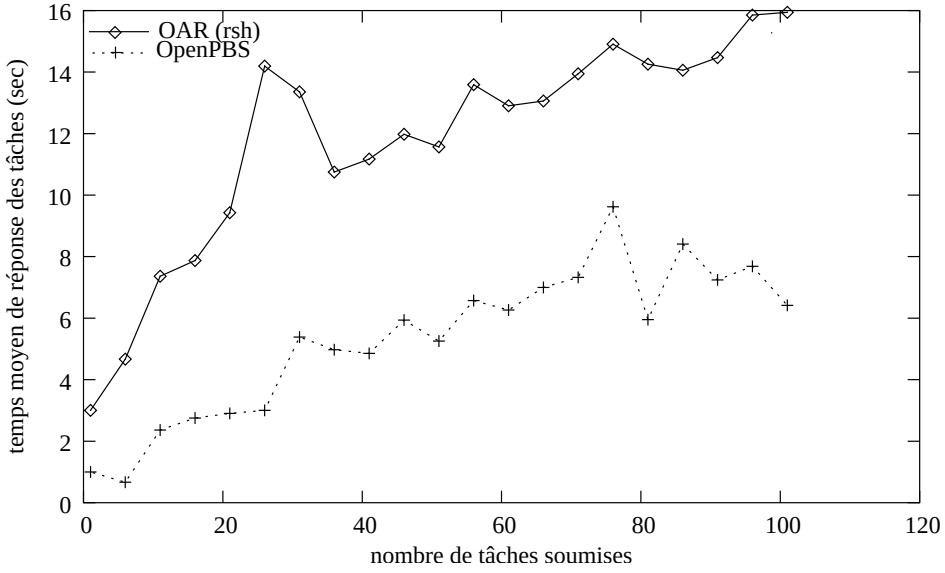


FIG. 8.13 – Temps de réponse moyen d'une tâche demandant 1 nœud en fonction du nombre de tâches soumises pour les systèmes OpenPBS et OAR (sans vérification des états des noeuds) sur la plate-forme Icluster (119 nœuds de calculs).

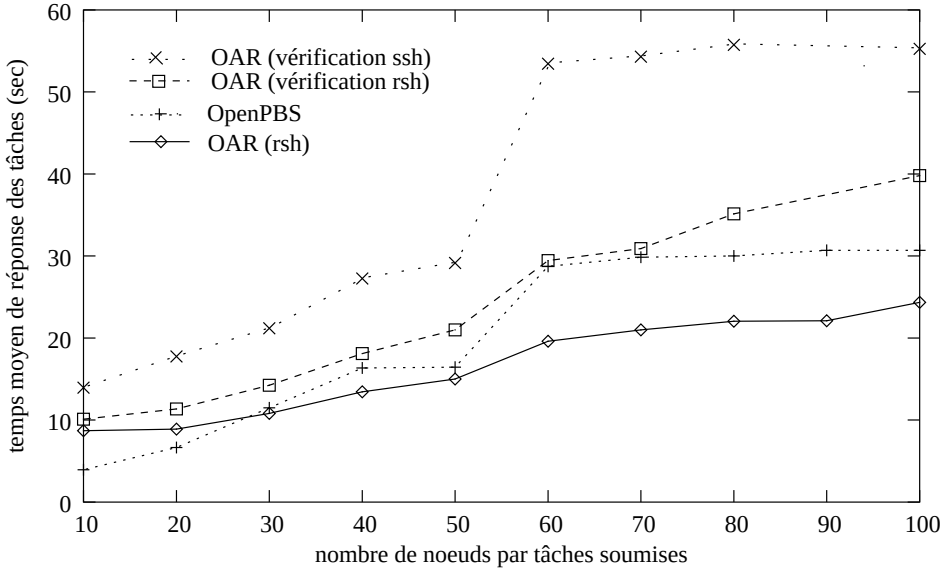


FIG. 8.14 – Temps de réponse moyen d'une tâche en fonction du nombre de nœuds demandés pour les systèmes OpenPBS et OAR (avec et sans vérification des états des noeuds) sur la plate-forme Icluster. Pour chaque point, 20 soumissions sont soumises simultanément.

augmentent.

Globalement, les performances sont bonnes sur des machines récentes où l'emploi d'une approche de haut-niveau n'est pas pénalisante. De plus la gestion d'un nombre important de ressources est maîtrisée.

8.3.6 Conclusion

Nous avons présenté un nouveau système de gestion de ressources pour grappe, baptisé OAR. La principale contribution de OAR provient de sa conception. Il repose sur deux composants de haut-niveau : une base de données SQL et un outil générique d'administration de grappes passant à l'échelle. La base de donnée est utilisée comme seul médium d'échange d'information entre les différents modules et assure ainsi une ouverture totale du système. La conception de l'ensemble est volontairement très modulaire et l'implémentation dans un langage de haut niveau assure au système extensibilité, robustesse et simplicité. Nous avons validé ces objectifs d'exploitation de type *Global Computing* à partir du système initial en montrant le bon niveau de performance de OAR en comparaison à celui de OpenPBS. Ces résultats démontrent l'avantage de l'emploi d'une méthode moderne de conception. Le résultat est une plate-forme ouverte et performante d'expérimentation et de recherche dans laquelle nous allons pouvoir envisager d'aborder de nouvelles problématiques (ordonnancement par lots, tâches malléables, plate-formes hétérogènes, réseaux non fiables, relation avec le *Grid Computing*,...) sans nous heurter à la complexité du système de gestion de ressources sous-jacent.

8.4 Bilan

Ce chapitre a présenté les différents projets qui utilisent le service de déploiement de l'environnement Taktuk. Ces réalisations sont diverses et traitent chacune des problèmes différents. Les outils d'exploitations comme les Katools et OAR axent leur utilisation sur les performances, la tolérance aux pannes, et l'adaptativité de Taktuk. Tandis que le déploiement d'une application parallèle vise surtout le passage à l'échelle et éventuellement le traitement de plates-formes composées de plusieurs grappes. Ces deux domaines ont des contraintes très différentes car le temps d'exécution d'une commande d'exploitation est très court, tandis que l'exécution d'un programme parallèle peut se dérouler sur plusieurs jours.
Cependant, la bibliothèque de déploiement Taktuk permet de traiter correctement ces deux problématiques, ce qui confirme son caractère flexible et générique.
Ces différents projets sont exploités par de nombreux utilisateurs : les utilisateurs d'ATHAPASCAN-1[47], les grappes utilisant la distribution Linux CLIC [7] et enfin les utilisateurs du système OAR [31] installé sur une partie des plates-formes du projet CIMENT (4 grappes). Cette utilisation démontre l'utilisabilité et la fiabilité de ces outils.

Quatrième partie

Conclusion et perspectives

Chapitre 9

Conclusion

9.1 Bilan

Le déploiement d'une machine parallèle virtuelle se résume à la diffusion d'une requête de création de processus sur un ensemble de noeuds et à l'interconnexion de ces derniers. Dans notre approche, nous utilisons des protocoles d'exécution distante pour réaliser cette diffusion. La multiprogrammation et un lancement récursif basé sur un arbre permettent de paralléliser cette phase de création de processus.

Nous avons montré que le problème d'une diffusion efficace de requêtes de création de processus est équivalent au problème déjà étudié de la diffusion d'un message ("single message broadcast"). Nous proposons d'utiliser la même stratégie d'ordonnancement de communication dite "au plus tôt" pour réaliser un déploiement performant. Cette modélisation permet également de calculer une borne inférieure sur le temps de déploiement en fonction du protocole utilisé pour la création distante des processus.

Cette stratégie "au plus tôt" a été implantée de deux façons. La première consiste à construire l'arbre de lancement optimal de manière statique et centralisée en fonction de paramètres mesurés sur la plate-forme cible. La deuxième utilise un algorithme "glouton de vol de travail hiérarchique" permettant de réaliser un déploiement adaptatif. Ces deux méthodes sont respectivement plus adaptées à un environnement d'exécution homogène ou hétérogène.

Les premiers résultats obtenus confirment qu'il est possible de réaliser un déploiement efficace et performant avec des outils de base (protocole d'exécution distante comme rsh et ssh) standards et coûteux. Ces évaluations confirment un coût de déploiement logarithmique du nombre de noeuds.
L'introduction d'une abstraction générique de protocole d'exécution distante garantit une totale indépendance du protocole utilisé ainsi qu'une forte portabilité. L'utilisation de protocole d'exécution distante de manière générique permet également une utilisation et une mise en oeuvre simple puisqu'il ne nécessite pas de modifier la

configuration des noeuds d'une grappe (installation et utilisation au niveau utilisateur).
Cet environnement utilise un langage de description facilement extensible qui permet de traiter simplement et efficacement des topologies variées de grappes : grappes homogènes, grappes hétérogènes, parc de machines (intranet) et grilles légères (agglomérat de grappes).

Ces stratégies et principes de déploiement ont été implantés sous la forme d'une bibliothèque nommé Taktuk. Cet environnement a permis de développer des outils de déploiement d'application parallèle ayant une grande capacité de passage à l'échelle. Ces performances permettent également de l'utiliser pour construire des outils d'exploitations pour des grappes de grandes tailles.

Sa fiabilité et sa flexibilité sont démontrées par son utilisation dans des projets variés comme Inuktitut, Katools (Distribué et utilisé dans la distribution linux Mandrake clic dédiée aux grappes) et OAR.

9.2 Perspectives à court terme

La bibliothèque de déploiement Taktuk a été développée de façon incrémentale. Cet environnement a été utilisé par d'autres projets dès que sa première version a été disponible. Les utilisateurs ou développeurs de projets utilisant Taktuk ont parfois utilisé les structures internes de Taktuk soit pour accéder des informations non diponibles via l'API, soit ces projets ont effectué des hypothèses fortes sur les effets de bord (non spécifiés) de certaines fonctions. Ceci a conduit à la conservation de structures de données obsolètes et coûteuses pour garantir la compatibilité avec ces projets.

Pour réduire le surcoût de cette implantation par rapport au coût unitaire du protocole utilisé, il est nécessaire d'effectuer des simplifications et des optimisations sur certaines parties critiques.
De plus, l'implantation des stratégies adaptatives et statiques peut être améliorée de manière à suivre de façon stricte un ordonnancement des requêtes d'exécution distante de type "au plus tôt".

Les premières expérimentations de cette bibliothèque de déploiement ont été effectuées sur une grappe composée de 100 noeuds mono-processeurs. Afin de valider cette approche, il serait souhaitable de réaliser des expérimentations sur des grappes de SMP. Ces expérimentations seront effectuées prochainement sur une grappe de 100 bi-processeurs Itanium (IA64). Une première série de mesures a été conduite sur la grappe de 50 bi-processeurs Intel Pentium4 du laboratoire ID et laisse envisager un bon comportement.

9.3 Perspectives à moyen terme

Un deuxième type d'expérimentations devra être effectué sur des grappes ou des agglomérats de grappes regroupant des milliers de noeuds en vue de tester le comportement du déploiement. Ces expérimentations pourront être effectuées sur les plate-formes des projets CIMENT et GRID 5000 en cours de mise en oeuvre à Grenoble et au niveau national. Sur ces plate-formes de très grande taille la probabilité de défaillance d'un noeud augmente fortement, cette caractéristique permettra de valider et d'améliorer les mécanismes de description, de tolérance aux pannes et de qualité de service de l'outil de déploiement Taktuk.

Durant ces travaux de recherche, nous nous sommes intéressés uniquement au problème du déploiement de l'exécution d'une application parallèle en considérant que les fichiers nécessaires étaient disponibles sur chaque noeud. Sur des plate-formes ne disposant pas de systèmes de fichiers communs (agglomérat de grappe, grappe hétérogène), il serait souhaitable d'étendre les fonctionnalités du déploiement de manière à traiter la diffusion ou la multi-diffusion des fichiers nécessaires à l'exécution d'une application parallèle (exécutables, fichiers de données, etc...). L'outil Mput (Cf. section 8.2) développé dans le projet Katools, donne une première solution dans ce domaine en utilisant un réseau virtuel applicatif dédié à la diffusion de fichiers. Cette base pourra être complétée en considérant des topologies de diffusion adaptées.

De plus, les plate-formes de grandes tailles fournissent un environnement formé de noeuds relativement "volatiles", les taux de défaillances étant important. Pour permettre à une application parallèle de s'adapter au mieux à la plate-forme et de l'exploiter le mieux possible, le mécanisme de déploiement pourrait être étendu de manière à gérer, durant l'exécution un ensemble dynamique de noeuds (ajouts / retraits). Une première idée pourrait consister à ne jamais terminer la phase de déploiement de Taktuk. Actuellement le lancement effectué dans l'approche adaptative fonctionne de manière à ajouter dynamiquement de nouveaux noeuds durant la phase de déploiement. Les mécanismes mis en jeu pour permettre l'ajout dynamique de noeuds sont arbitrairement terminés lorsque tous les noeuds sont atteints, de manière à libérer des ressources non utilisées. Le traitement de l'ajout de noeuds dynamique peut donc être facilement effectué en laissant fonctionner ces mécanismes pendant l'exécution de l'application parallèle.

Bibliographie

[1] C. Martin et O. Richard. Algorithme de vol de travail appliqué au déploiement d'applications parallèles Dans *Renpar '15, pages 64–71, La Colle sur Loup, France, Octobre 2003*

[2] N. Capit, G. Da Costa, G. Huard, C. Martin, G. Mounié, P. Neyron et O. Richard. Expérience autour d'une nouvelle approche de conception d'un gestionnaire de travaux pour grappe Dans *CFSE'3, pages 602–613, La Colle sur Loup, France, Octobre 2003*

[3] O. Richard, G. Da Costa et C. Martin. Expériences sur les systèmes distribuées de grande taille. École d'hiver GRID 2002, Aussois.

[4] C. Martin et W. Billot. Lancement d'applications sur des grappes de grande taille. Dans *Renpar '14*, pages 17–24, Hammamet, Tunisie, Avril 2002.

[5] P. Augerat, C. Martin and B. Stein Scalable monitoring and configuration tools for grids and clusters, 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing, 2002.

[6] C. Martin and O. Richard Parallel launcher for cluster of PC In *Proceedings ParCo 2001*, pages 473–480, Naples, Italy, September 2001.

Bibliographie

[1] Computational plant. http ://www.cs.sandia.gov/cplant/.

[2] *Distribution CLIC (Cluster Linux pour le calcul)*. http ://clic.mandrakesoft.com/.

[3] Ganglia : distributed monitoring and execution system. http ://ganglia.sourceforge.net/.

[4] *The GM-2 Message Passing System*. http ://www.myri.com/scs/index.html.

[5] *Inuktitut : Overview*. http ://www-id.imag.fr/Logiciels/inuktitut/.

[6] *LAM/MPI user survey*. http ://www.lam-mpi.org/user_survey.

[7] "mandrake soft". http ://www.mandrakesoft.com/products/clustering ?wslang=fr.

[8] *Maui Scheduler*. http ://www.supercluster.org/maui.

[9] *Maui Scheduler Molokini Edition*. http ://mauischeduler.sourceforge.net/.

[10] *MySQL Reference Manual*. http ://www.mysql.com/documentation/.

[11] *OpenPBS*. http ://www.openpbs.org.

[12] Oscar : Open source cluster application resources. http ://oscar.sourceforge.net/.

[13] *PBS Technical Overview*. http ://www.openpbs.org/overview.html.

[14] Sandia national laboratories. http ://www.sandia.gov/.

[15] Ieee standard for scalable coherent interface. Technical report, The Institute of Electrical and Electronics Engineers, 1992.

[16] R. namyst and j.-f. méhaut. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vivouroux, editors, *EuroPVM '95*, pages 179–184, Lyon, France, 1995. LIP, ENS Lyon, Hermès.

[17] Mpich-pm/clump : an mpi librairy based on mpich 1.0 implemented on top of score. Technical report, Parallel and Japan Distributed System Software Laboratory, 1998.

[18] *Cplant, Proc. Second Extreme Linux Workshop, Monterey, California*, 1999.

[19] Quadrics qsnet interconnect, 2002. http ://www.quadrics.com.

[20] Carl Albing. Cray NQS : production batch for a distributed computing world. In *Proceedings of the 11th Sun User Group Conference and Exhibition*, pages 302–309, Brookline, MA, USA, December 1993. Sun User Group, Inc.

[21] T. Angskun, P. Uthayopas, and C. Ratanpocha. Ksix parallel programming environment for beowulf cluster. In *Parallel and Distributed Proceeding Techniques and Applications 2000,Las Vegas, Nevada , USA*.

[22] Olivier Aumage. *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes.* PhD thesis, Ecole Normale Supérieure de Lyon, LIP, 2002.

[23] M. Baker, G. Fox, and H. Yau. Cluster computing review. Technical report, Northeast Parallel Architectures Center, Syracuse University, 1995.

[24] Amotz Bar-Nov and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message passing systems. In *SPAA*, pages 13–22, 1992.

[25] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Symposium on Foundations of Computer Science*, pages 356–368, Santa-Fe, New Mexico, 1994.

[26] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Steitz, Jakov N. Seizovic, and Wen-King Su. Myrinet : A gigabit per second local area network. *IEEE-Micro*, 15(1) :29–36, February 1995.

[27] Ron Brightwell and Lee Ann Fisk. Scalable parallel application launch on cplant. In *Proceedings of SC'2001*.

[28] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The portals 3.0 message passing interface. Technical report, Sandia Technical Report, November 1999.

[29] Michael Brim, Ray Flanery, Al Geist, Brian Luethke, and Stephen Scott. Cluster command and control (c3) tool suite. In *Proceedings of 3rd Austrian-Hungarian Workship on Distributed and Parallel Systems (DAPSYS 2000) in conjunction with EuroPVM/MPI*, 2000.

[30] Alejandro Calderón, Félix García, Jesús Carretero, Jose M. Pérez, and Javier Fernández. An implementation of MPI-IO on Expand : A parallel file system based on NFS servers. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2474 of *Lecture Notes in Computer Science*, pages 306–313. Springer-Verlag, 2002.

[31] Nicolas Capit, Guillaume Huard Georges Da Costa, Cyrille Martin, Gregory Mounié, Pierre Neyron, and Olivier Richard. Expérience autour d'une nouvelle approche de conception d'un gestionnaire de travaux pour grappe. In *CFSE'3*, pages 602–613, La Colle sur Loup, France, October 2003.

[32] T.L. Casavant and J.G. Khul. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14 :141–154, 1988.

[33] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[34] Jeff Chase, Laura Grit, David Irwin, Sara Sprenkle, and Justin Moore. Dynamic resource trading in a grid site manager. In *High Performance Distributed Computing (HPDC-12)*, 2003.

[35] G. Chiola and G. Ciaccio. Efficient parallel processing on low-cost clusters with GAMMA active ports. *Parallel Computing*, 26 :333–354, 2000.

[36] Brent N. Chun. Gexec. http ://www.theether.org/gexec/.

[37] Brent N. Chun and David E. Culler. Rexec : A decentralized, secure remote execution environment for clusters. In *Proceedings of 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, 2000.

[38] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using multirail networks in high-performance clusters. *Concurrency and Computation : Practice and Experience*, 15(7-8) :625–65, 2003.

[39] The MPI-IO Committee. "mpi-io : A parallel file i/o interface for mpi, version 0.5. world-wide web http ://lovelace.nas.nasa.gov/mpi-io ", April 1996.

[40] Alexandre da Silva Carissimi. *Athapascan-0 : Exploitation de la multiprogrammation légère sur grappes de multiprocesseurs*. PhD thesis, INPG, LMC, Grenoble, November 1999.

[41] By Daniel, J. Barrett, and Richard Silverman. *SSH, The Secure Shell : The Definitive Guide*. O'REILLY and Associates, 2001.

[42] G. Fedak, C. Germain, V. N'eri, and F. Cappello. Xtremweb : A generic global computing system. In *In IEEE Int. Symp. on Cluster Computing and the Grid*, 2001.

[43] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[44] Message Passing Interface Forum. Mpi-2 : Extensions to the message-passing interface. Technical report, University of Tennessee, 1996.

[45] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus : An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[46] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. Storm : Lightning-fast resource management. In *Suppercomputing 2002*, Baltimore, MD, November 2002.

[47] F. Galilée. *Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, September 1999.

[48] D. R. Ghormley, D. Petrou, S. H. Rodrigues, and A. M. Vahdat. GLUnix : A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28(9), 1998.

[49] William Gropp and Ewing Lusk. *MPICH, Users guide*.

[50] William Gropp, Ewing Lusk, David Ashton, Rob Ross, Rajeev Thakur, and Brian Toonen. *MPICH2 Abstract Device Interface 3*. Argonne National Laboratory, University of Chicago, September 2003. http ://www-unix.mcs.anl.gov/mpi/mpich2/docs/index.htm.

[51] Atshushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. Highly efficient gang scheduling implementation. In ACM, editor, *Parallel Computing*, Orlando, Florida, USA, November 1998.

[52] National HPCC. Software exchange review. *http ://www.crpc.rice.edu/NHSEreview*, 1996.

[53] IBM Corporation. *Using and Administering LoadLeveler – Release 3.0*, 4 edition, August 1996. Document Number SC23-3989-00.

[54] IEEE Standards Department. *1003.4d8 POSIX System application program interface : Threads extensions [C language]*, 1994.

[55] Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O'Carroll, and H. Harada. Rwc pc cluster ii and score cluster system software – high performance linux cluster. In *of the 5th Annual Linux Expo*, pages 55–62, Raleigh, North Carolina, May 1999.

[56] J.A Kohl, G.A. Geist, P.M. Papadopoulos, and S. Scott. Beyond pvm 3.4 : What we've learned, what's next, and why. In *Proceedings of EuroPVM-MPI 97*, November 1997.

[57] Eliezer Levy and Abraham Silberschatz. Distributed file systems : Concepts and examples. *Departement of Computer Sciences, University of Texas at Austin, Texas 78712-1188*, 1990.

[58] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.

[59] P. Lombard and Y. Denneulin. nfsp : A Distributed NFS Server for Clusters of Workstations. In *16'th IPDPS*. IEEE, April 2002.

[60] Pierre Lombard and Yves Denneulin. nfsp : A distributed nfs server for cluster of workstations. April 2002.

[61] Ewing Lusk and William Gropp. Scalable unix commands for parallel processor. In IEEE Computer Society Press, editor, *Scalable High-performance Computing Conference*, pages 56–62, 1994.

[62] Cyrille Martin and Wilfrid Billot. Lancement d'application sur des grappes de grande taille. In *proceedings of RenPar'14*, pages 17–24, 2002.

[63] Cyrille Martin and Olivier Richard. Parallal launcher for cluster of pc. In Wolrd Scientific, editor, *Proceedings ParCo 2001*, pages 473–480, 2001.

[64] Cyrille Martin and Olivier Richard. Algorithme de vol de travail appliqué au déploiement d'applications parallèles. In *Renpar '15*, pages 64–71, La Colle sur Loup, France, October 2003.

[65] R. Namyst. *PM2 : un environnement pour une conception portable et une excution efficace des applications parallles irrgulires*. PhD thesis, Université de Lille 1, France, January 1997.

[66] National Institute of Standards and Technology. *IMPI : Interoperable MPI*, 2000. http ://impi.nist.gov/IMPI/.

[67] Emil Ong, Ewing Lusk, and William Gropp. Scalable unix commands for parallel processors : A high-performance implementation. In *Proceedings of Euro PVM-MPI*, 2001.

[68] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, and Adolfy Hoisie. Performance Evaluation of the Quadrics Interconnection Network. *Journal of Cluster Computing*, (6) :125–142, January 2003.

[69] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers : Early Experiences. In *Proc. 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 85–105, Edinburgh, Scotland, UK, July 24 2002. Available at http ://www.cs.ualberta.ca/~paullu/.

[70] L. Prylli and B. Tourancheau. BIP : A new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, volume 1388 of *Lect. Notes Comp. Science*, pages 472–485. LNCS, IEEE, Springer, April 1998.

[71] Robert D Russell and Philip J Hatcher. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing*, Atlanta, Georgia, February 1998.

[72] Roger L.Haskin Frank B. Schmuck. The tiger shark file system. *In Proceedings of the 41st IEEE Computer Society International Conference (COMPCON '96), pages 226-231, Sanat Clara, CA, USA*, February 1996.

[73] Roger L.Haskin Frank B. Schmuck. Gpfs : A shared-disk file system for large computing clusters. *In Proceedings of the 5th Conference on File and Storage Technologies*, January 2002.

[74] R. Srinivasan. XDR : External data representation standard. RFC 1831, Network Working Group, Sun Microsystems, Inc., August 1995.

[75] W.Richard Stevens. *Unix Network Programming*. Prentice Hall, 1990.

[76] W.Richard Stevens. *Unix Network Programming*, chapter 14. Prentice Hall, 1990.

[77] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley Verlag, München, 4. edition, 2000.

[78] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2 : High performance communication middleware for heterogeneous network environments. In *Super Computing*, pages 52–53, Dallas, USA, November 2000.

[79] Douglas Thain and Miron Livny. Condor and the grid. In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, *Grid Computing : Making The Global Infrastructure a Reality*. John Wiley, 2003.

[80] P. Uthayopas, J. Maneesilp, and P. Ingongnam. Scms : An integrated cluster management tool for beowulf cluster system. In *Proceedings of the International Conference on Parallel and Distributed Proceeding Techniques and Applications*, pages 26–28, June 2000.

[81] Putchong Uthayopas, Sugree Phatanapherom, Thara Angskun, and Somsak Sriprayoonsakul. Sce : A fully integrated software tool for beowulf cluster system. In *HPC Revolution*, University of Illinois, USA, 2001.

[82] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly & Associates, June 2002.

[83] Steve Vinoski. New features for corba 3.0. *Communications of the ACM*, 1998.

[84] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages : A mechanism for integrated communication and

computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[85] M.H. Willebeek-Le-Mair and P. Reeves. Strategies for dynamic load-balancing on higly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9) :979–993, 1993.

[86] Len Wisniewski, Brad Smisloff, and Nils Nieuwejaar. Sun MPI I/O : Efficient I/O for parallel applications. In *Proceedings of SC99 : High Performance Networking and Computing*, Portland, OR, November 1999. ACM Press and IEEE Computer Society Press.

[87] David E. Womble and David S. Greenberg. Parallel I/O : An introduction. *Parallel Computing*, 23(4) :403–417, June 1997.

[88] Songnian Zhou. LSF : load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992. Supercomputing Computations Research Institute, Florida State University.

Résumé

Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles

La taille grandissante des grappes de calcul pose le problème du "passage à l'échelle" des applications qui s'exécutent sur ces plates-formes. Ceci concerne les applications de calculs scientifiques et les applications permettant d'exploiter ces plates-formes (administration, surveillance de charge, etc...).
Dans ce travail de thèse nous nous sommes intéressés au déploiement d'une application parallèle sur une grappe de grande taille. L'objectif de cette étude était de fournir une méthode de déploiement efficace sur des grappes composées de milliers de noeuds et pouvant être facilement étendue aux grilles de calcul. Le déploiement inclut d'une part le lancement du programme parallèle sur tous les noeuds et d'autre part la mise en oeuvre d'un environnement de communication entre ces instances de programme. L'efficacité est obtenue par la parallélisation systématique des différentes initiations d'exécution distante. Ces travaux montrent que le problème de la diffusion optimale d'une requête d'exécution est similaire au problème largement étudié de la diffusion d'un message sur un réseau complètement maillé. Nous proposons une bibliothèque, "*Taktuk*", permettant de réaliser un ordonnancement dynamique (par vol de travail) des communications (appels d'exécution distante) de manière générique.
L'utilisabilité et le bon fonctionnement de l'outil que nous proposons sont validés par son utilisation et sa diffusion dans plusieurs projets : KaTools (inclus et utilisé par la distribution Linux Mandrake Clic), OAR (gestionnaire de travaux pour grappes) et Inuktitut (bibliothèque de communication d'ATHAPASCAN).

Mots-clés : déploiement, diffusion, ordonnancement, protocole d'exécution distante, efficace, générique, passage à l'échelle.

Abstract

Deployment and control of parallel applications on large cluster

The increasing size of cluster of workstations sets down the scalability problem of applications running on these platforms. This concerns both numerical parallel applications and exploitation tools (administration, monitoring...). In this thesis work, we study the deployment of parallel applications on large clusters, that can be extended to grids. The deployment includes on one hand the launch of the parallel program on all nodes and on the other hand the setting up of a communication layer. Efficiency is obtained thanks to the overlay of all independent steps of the deployment.
This work shows this problem as equivalent as the well known problem of the single message broadcast. Performance gap between the cost of a network communication and this of a remote execution call enable us to use a work stealing algorithm to realize a near-optimal schedule of remote execution calls.
The good properties and performance figures of this tool, Taktuk, are demonstrated by its use in several projects like : KaTools (included and used by the Clic Mandrake Cluster Linux distribution), OAR (Job manager) and Inuktitut (Communication layer of the environment ATHAPASCAN).

Keywords : deployment, parallel launch, broadcast, scheduling, remote execution call protocol, scalability, efficient, generic.