PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants

Alexandre Denis

▶ **To cite this version:**

Alexandre Denis. PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants. 14èmes Rencontres Francophones du Parallélisme (RenPar'14), Apr 2002, Hammamet/Tunisie, France. pp.99-106. inria-00000133

HAL Id: inria-00000133
https://inria.hal.science/inria-00000133

Submitted on 24 Jun 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

PadicoTM : un environnement ouvert pour l'intégration d'exécutifs communiquants *

Alexandre Denis

IRISA/IFSIC
Campus de Beaulieu
35042 RENNES – France
Alexandre.Denis@irisa.fr

Résumé
Les grilles de calcul ont une importance croissante dans le monde du calcul scientifique. Leur programmation demande l'utilisation de plusieurs paradigmes de communication qui sont implémentés par des exécutifs différents. Certains de ces exécutifs ne sont pas capables de tirer profit de certaines infrastructures réseau disponibles sur les grilles. Dans cet article, nous décrivons un environnement ouvert pour l'intégration de plusieurs exécutifs qui leur permet de partager efficacement les ressources réseaux d'une grille. Un tel environnement encourage les programmeurs à utiliser le paradigme de communication le mieux adapté à chaque tâche, sans contrainte quant au réseau sous-jacent. Ainsi, il n'y a pas d'obstacle pour déployer les applications sur une configuration spécifique de grille.

Mots-clés : grille de calcul, support exécutif, couplage de code, réseau haute performance.

1. Introduction

Les grilles de calcul sont au carrefour des systèmes de calcul distribué et parallèle. Grâce à la puissance qu'elles mettent à notre disposition, ces grilles permettent le développement d'applications qui n'étaient pas envisageables il y a quelques années. Il est maintenant possible, dans le domaine du calcul scientifique, de simuler des phénomènes de plus en plus complexes. Par exemple, les simulations utilisées dans la conception d'un avion mettent en jeu plusieurs codes couplés dans les domaines de la mécanique, de la dynamique des fluides, de l'électromagnétisme, etc. Chaque code a ses propres contraintes en termes de ressources de calcul (visualisation, parallélisme, calcul vectoriel) ; les codes sont généralement développés indépendamment. Pour déployer ces codes sur la grille et les coupler entre eux, être obligé d'utiliser le même paradigme de communication (par exemple MPI) pour tous les codes est une contrainte très forte. Il est probable que les codes ont des besoins différents (passage de messages, mémoire partagée). De plus, le couplage de codes se fait de façon plus naturelle à l'aide d'un mécanisme qui transfère à la fois le contrôle et les données, tel que Corba [14] ou les RMI Java. Cependant, il existe quelques obstacles qui empêchent parfois les concepteurs de choisir l'exécutif (terme générique que nous emploierons pour désigner les supports exécutifs et les intergiciels) le plus adapté à leur besoins.
Le premier obstacle vient du fait que les exécutifs utilisés habituellement en programmation distribuée (Corba, RMI Java) sont incapables d'utiliser les réseaux haute performance disponibles sur les calculateurs parallèles et les grappes. Les implémentations existantes sont fondées principalement sur le protocole TCP/IP qui pénalise les performances quand un réseau rapide est disponible. Il faut donc actuellement faire un compromis et choisir entre un paradigme de communication adapté au couplage (Corba) avec de mauvaises performances, ou un paradigme moins adapté au couplage (MPI) avec de bonnes performances.
Le deuxième obstacle est l'utilisation simultanée de plusieurs exécutifs. Par conception, la plupart des bibliothèques d'accès aux réseaux haute performance n'autorisent pas le partage. De plus, les différents

* Ces travaux sont supportés par l'ACI GRID "RMI" du ministère de la recherche.

exécutifs ne sont pas conçus pour collaborer entre eux. L'utilisation d'une même ressource réseau par plusieurs exécutifs en même temps n'a pas été prévue. En outre, les bibliothèques de communication sont conçues dans l'optique d'un paradigme particulier – passage de message ou appel de procédure distante – adapté au calcul parallèle ou au calcul distribué, mais pas aux deux en même temps. Faire cohabiter deux exécutifs basés sur deux paradigmes différents et leur faire partager une ressource réseau qui n'est pas prévue pour être partagée est une tâche délicate.
Le risque est donc que le programmeur choisisse un seul paradigme de communication pour réaliser les tâches de couplage de codes et de calcul parallèle. Si on envisage par exemple l'utilisation de MPI seul pour surmonter ces deux obstacles, on se heurtera à de nouveaux obstacles : MPI n'a pas été conçu pour le transfert de contrôle, ce qui oblige à simuler un mécanisme d'appel de procédure à distance au-dessus de MPI pour réaliser le couplage. D'autre part, il peut être souhaitable que certains codes parallèles soient fondés sur un paradigme de mémoire partagée. Notre objectif est de permettre au programmeur d'utiliser l'exécutif adapté à chaque tâche, que tous les exécutifs bénéficient des performances du réseau, et que plusieurs exécutifs puissent être utilisés simultanément comme par exemple lors de la mise en œuvre des objets Corba parallèles [18, 7] qui nécessitent Corba et MPI en même temps.
La section 2 met en évidence les problèmes qui surgissent lors de l'utilisation simultanée de plusieurs exécutifs et décrit les solutions apportées à chaque problème. La section 3 présente les implémentations et les performances obtenues avec plusieurs exécutifs. La section 4 donne un aperçu des travaux connexes dans le domaine des plate-formes d'intégration d'exécutifs haute performance. Enfin la section 5 dresse le bilan de la contribution et présente quelques perspectives.

2. Architecture de PadicoTM

PadicoTM est notre plate-forme de recherche pour explorer le domaine de l'intégration de plusieurs exécutifs communiquants. Le rôle de PadicoTM est de fournir une infrastructure haute performance sur laquelle on peut implémenter des exécutifs tels que Corba [14], MPI, JVM (*Java Virtual Machine*), DSM (*Distributed Shared Memory* – mémoire virtuellement partagée), etc. PadicoTM offre un environnement qui résout les conflits d'accès aux ressources (processeur, réseau) pour autoriser la cohabitation efficace de plusieurs exécutifs au sein d'un même processus. Cette plate-forme est utilisée dans le cadre des objets Corba parallèles [18, 7] qui ont besoin à la fois de Corba et de MPI.

2.1. Vue d'ensemble

La conception de PadicoTM est inspirée des composants logiciels. PadicoTM est très modulaire, chaque module étant composé de fichiers binaires et d'un fichier de description qui leur est attaché. Les modules de PadicoTM sont de trois types : *core*, *services* et modules applicatifs. Les modules *core*[1] mettent en œuvre la gestion des modules, ainsi que le multiplexage et la gestion coopérative du réseau et du multi-threading. Ces modules sont *Puk*, *TaskManager* et *NetAccess* décrits dans les sections suivantes. Les services utilisent le *core* comme fondation. Ils peuvent être vus comme des *plug-ins* du *core*. Les modules applicatifs, créés par l'utilisateur, peuvent utiliser les modules *core*, les services, ou tout autre module utilisateur. Les services sont :

- des interfaces d'accès au réseau, telles que *VSock* et *Circuit* (sections 2.5 et 2.6) qui fournissent différents paradigmes au-dessus de *NetAccess* ;
- des exécutifs (ou intergiciels), tels que Corba, MPI et une machine virtuelle Java (section 3) ;
- des modules de commande, pour diriger les processus à distance (section 2.7).

2.2. Gestion de la dynamicité

Les mondes du calcul distribué (par exemple Corba) et du calcul parallèle (par exemple MPI) sont fondés sur des modèles réseau différents. En parallélisme – sur une grappe de PC ou sur une machine parallèle –, la topologie du réseau est habituellement statique ; on ne peut pas ajouter ou retirer de nœud en cours de session, et le même programme est exécuté sur tous les nœuds – c'est l'approche SPMD (*Single Program Multiple Data*). À l'opposé, Corba utilise une topologie réseau dynamique ; les serveurs et les clients peuvent être démarrés dynamiquement et établir ou rompre des connexions à tout moment – c'est

1. *cœur* ou *noyau* ; par soucis de clarté, pour éviter les confusions avec le noyau du système d'exploitation, nous employons le mot anglais *core* pour désigner le noyau de PadicoTM.

```
<mod name="ORB" driver="binary">
  <requires>VSock</requires>
  <attr label="NameService">paraski.irisa.fr:10000</attr>
  <unit>libORB.so</unit>
</mod>
```

FIG. 1 – *Exemple de description de module en* XML *: le module* ORB.

l'approche MPMD (*Multiple Programs Multiple Data*). L'immense majorité des bibliothèques destinées aux réseaux haute performance ont été conçues dans l'optique du parallélisme. Pour que Corba utilise un tel réseau, il est donc nécessaire d'établir une projection du modèle distribué vers le modèle parallèle.
La solution que nous proposons est de démarrer un programme d'amorce (*bootstrap*) identique sur tous les nœuds susceptibles d'être utilisés. De cette façon, la contrainte SPMD de la bibliothèque de communication est satisfaite. L'application est ensuite chargée dynamiquement à l'intérieur des processus déjà démarrés. Les applications sont des modules chargeables dynamiquement (les "`.so`" sous Unix). Grâce à ce mécanisme, des applications différentes peuvent être chargées sur chaque nœud.
Dans PadicoTM, nous appelons ce programme d'amorce *Padico µ-Kernel*, que nous abrégeons en *Puk*. Les tâches de *Puk* sont la gestion des processus de façon adaptée au mode SPMD de la bibliothèque de communication ainsi que les opérations de chargement, exécution et déchargement des modules (modules du *core*, services et modules applicatifs) dans ces processus.
Nous voulons que le concept de module soit ouvert. Nous ne nous restreignons pas au cas des bibliothèques binaires chargeables dynamiquement. Pour cela, chaque module est décrit par un fichier XML [21]. Ce fichier de description contient : le nom d'un pilote (*driver*) capable de prendre en charge ce type de module, des références à d'autres modules pour gérer les dépendances, des unités (*units*) et des attributs. Le sens exact des unités est défini par le pilote associé au module. Un pilote est un ensemble de fonctions qui permettent à *Puk* de charger, exécuter et décharger un type donné d'unités. Il existe par exemple le pilote `binary` pour lequel les unités sont des bibliothèques binaires dynamiques, le pilote `java` pour lequel les unités sont des classes Java, ou encore le pilote `pkg` pour lequel les unités sont des modules, permettant ainsi l'assemblage de modules. Les attributs sont des valeurs positionnées pour configurer le module. La figure 1 est l'exemple de la description XML du module ORB ; ce module est de type `binary`, il contient l'unité `libORB.so`, utilise les services du module *VSock* et est configuré par un attribut `NameService` qui indique l'adresse du service de noms Corba.

2.3. Gestion cohérente du multi-threading

De nombreux exécutifs utilisent maintenant le multi-threading. De plus, certaines applications utilisent elles-mêmes le multi-threading. Cependant, il est probable que des exécutifs différents utilisent des politiques incompatibles, des tâches de fond concurrentes, ou tout simplement des bibliothèques de multi-threading différentes. Ces aspects rendent périlleuse la cohabitation de plusieurs exécutifs multi-threadés dans le même processus. Au pire, la cohabitation ne fonctionne pas ; au mieux, il y a seulement concurrence qui conduit à une sous-exploitation des ressources disponibles.
Pour résoudre cet aspect de la coopération d'exécutifs, PadicoTM sert d'intermédiaire pour l'accès au multi-threading. Nous sommes ainsi assurés que tous les exécutifs et applications utilisent la même bibliothèque : celle fournie par PadicoTM, éliminant du même coup le principal problème de compatibilité. Cela nous assure également une gestion cohérente entre les différents exécutifs puisqu'elle est prise en charge par PadicoTM qui voit tous les modules chargés.
Une solution serait d'utiliser les threads Posix (`pthread`). Cependant, il a été mis en évidence [5] que les threads Posix et les communications réseau (MPI en particulier) ne cohabitent pas efficacement. C'est pourquoi nous choisissons plutôt la bibliothèque de multi-threading Marcel [6, 15] qui nous apporte la portabilité sur de nombreux systèmes sans négliger les performances, et en particulier les performances réseau. Marcel est une bibliothèque de multi-threading en espace utilisateur qui implémente un ordonnancement N:M sur les architectures SMP et qui fournit une interface proche de Posix. Marcel a été conçu de façon à garantir une bonne réactivité des communications réseau quand celles-ci sont

réalisées par la bibliothèque Madeleine [3, 15]. Le module de PadicoTM que nous appelons *TaskManager* gère le multi-threading de façon cohérente, et présente aux modules une API semblable à Posix en se basant sur Marcel. Il fournit la possibilité d'enregistrer des procédures de rappel (*callback*) pour éviter aux exécutifs d'effectuer des boucles d'attente coûteuses en performance ; il sert d'intermédiaire pour les appels systèmes de façon à ne pas bloquer le processus en cas d'appel bloquant ; enfin, il fournit un accès asynchrone aux opérations *Puk* en tenant compte de la réentrance.

2.4. Accès coopératif au réseau

L'accès aux réseaux haute performance est l'aspect le plus problématique lors de l'utilisation simultanée de plusieurs exécutifs. Certains pilotes réseau ne supportent qu'un seul client (par exemple BIP [17] sur Myrinet) interdisant plus d'un exécutif à la fois s'il n'y a pas coopération. Certains réseaux ont des ressources très limitées (par exemple SCI) qui pourraient être épuisées rapidement si plusieurs exécutifs les utilisaient en même temps. Certains réseaux sont utilisables par plusieurs pilotes qui ne peuvent pas coexister (par exemple Myrinet utilisable par BIP *ou* GM, pas les deux en même temps). Comme pour le multi-threading, si plusieurs exécutifs sont chargés en même temps et ne collaborent pas, au pire ils ne fonctionnent pas ; au mieux, ils entrent en concurrence pour l'accès au réseau.

Il serait donc plus judicieux d'avoir une base commune qui effectue une gestion intelligente et cohérente des accès au réseau. Les idées directrices d'une telle base commune sont :

- factoriser le code nécessaire aux communications (thread de communication, par exemple) plutôt que d'avoir ce code dupliqué dans chaque exécutif ;
- utiliser le réseau sous-jacent avec la méthode la mieux adaptée, et fournir une interface de programmation uniforme de façon à ce que l'adaptation au réseau ne soit pas faite dans chaque exécutif ;
- faciliter l'écriture ou le portage d'exécutifs en présentant plusieurs API adaptées aux différents paradigmes des exécutifs ;
- gérer, le cas échéant, plusieurs exécutifs en même temps de façon coopérative plutôt que concurrente.

Pour nous assurer la portabilité vis-à-vis du réseau et apporter de bonnes performances de communication, nous avons choisi d'utiliser la bibliothèque Madeleine [3, 15] comme fondation. Comme indiqué en section 2.3, Madeleine est une bibliothèque de communication conçue pour fonctionner conjointement avec la bibliothèque de multi-threading Marcel. Madeleine fournit une interface de programmation uniforme pour l'accès à des réseaux variés tels que Myrinet, SCI ou Ethernet. Madeleine assure la portabilité par rapport au réseau, et la haute performance grâce notamment à l'utilisation de méthodes sans copie des données en mémoire.

La gestion des communications réseau dans PadicoTM est confiée à un module que nous appelons *NetAccess*. *NetAccess*, fondé sur Madeleine, met en œuvre les mécanismes communs de réception depuis le réseau en se basant sur les procédures de rappel de *TaskManager*. L'avantage de cette approche est triple : les exécutifs sont déchargés de la gestion des threads de communication, ce qui simplifie leur code ; la gestion des communications au niveau de PadicoTM améliore les performances en évitant la concurrence lors de l'exécution ; *NetAccess* réalise le multiplexage d'accès au réseau ce qui permet à plusieurs exécutifs d'utiliser Madeleine qui n'est pourtant pas multi-client – Madeleine ne "voit" que PadicoTM, et les différents exécutifs ne "voient" que PadicoTM. De cette façon, les exécutifs n'ont pas besoin de tenir compte les uns des autres pour accéder coopérativement au réseau.

NetAccess donne aux modules la possibilité d'enregistrer directement des procédures de rappel qui réagissent aux réceptions de messages. Ceci permet la factorisation des mécanismes de réception réseau dans *NetAccess*. Ces mécanismes réalisent un multiplexage logique, c'est-à-dire que différents canaux de communications logiques sont créés au-dessus de Madeleine. Quand plusieurs exécutifs sont actifs simultanément dans un processus PadicoTM, ils utilisent *NetAccess* exactement de la même façon que lorsqu'ils sont seuls. La séparation logique des communications qui sont multiplexées vers un seul canal Madeleine est totalement transparente pour les modules au-dessus de *NetAccess*.

Le multiplexage nécessitant d'envoyer des données supplémentaires sur le réseau pour identifier le canal logique, il ajoute un surcoût à la latence. Nous introduisons un mécanisme d'en-têtes combinés qui concatène les en-têtes des différentes couches logicielles. Il n'est ainsi pas nécessaire d'envoyer de paquet supplémentaire sur le réseau pour réaliser le multiplexage. Grâce aux en-têtes combinés, le coût

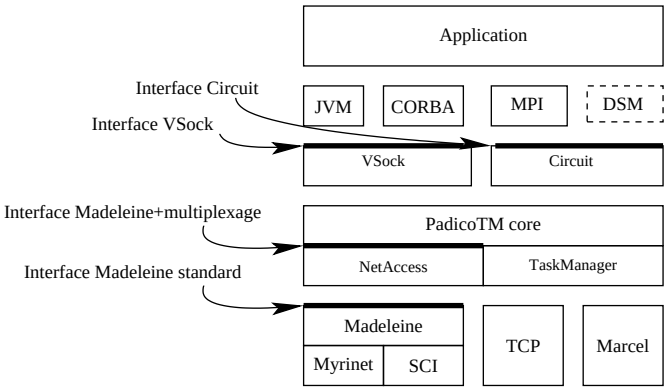

FIG. 2 – *Modules de PadicoTM et interfaces aux différents niveaux*

du multiplexage est négligeable. De plus, la latence logicielle supplémentaire ajoutée par le mécanisme de procédure de rappel est imperceptible.

Puk, *TaskManager* et *NetAccess* constituent le *core* de PadicoTM. Les autres modules, appelés services, viennent se brancher sur le *core*. La figure 2 résume les différents modules de PadicoTM.

2.5. Sockets virtuelles

Pour adapter rapidement des exécutifs existants à une utilisation haute performance sur PadicoTM, une approche consiste à fournir au-dessus de *NetAccess* des interfaces standard. Introduite avec BSD 4.2, l'interface socket est probablement l'interface de programmation réseau la plus populaire. Elle est habituellement utilisée pour accéder au protocole TCP/IP. Cependant, même si le protocole TCP/IP est trop lourd pour l'utilisation que nous voulons en faire, l'interface socket de Berkeley en elle-même convient. Nous choisissons donc d'implémenter un module qui fournit une interface de type socket au-dessus de *NetAccess*. Nous appelons ce module de "sockets virtuelles" en espace utilisateur *VSock*. *VSock* emprunte une démarche similaire à Fast Sockets [19] construit au-dessus de la bibliothèque de communications Active Messages. De nombreux exécutifs sont fondés sur des sockets TCP/IP, et certains ont une implémentation très intimement liée au modèle de programmation par socket. Grâce à *VSock*, leur adaptation à PadicoTM s'effectue simplement, sans les modifier.

Le module *VSock* implémente un sous-ensemble de l'API standard des sockets. Dans un but de haute performance, il ne fournit qu'un transport de datagrammes en mode "zéro copie". À la différence de TCP/IP, *VSock* ne supporte pas les flux continus ; ce n'est pas gênant si le logiciel qui utilise *VSock* gère convenablement les frontières des messages, et cela permet de n'avoir aucune copie supplémentaire. Ces services suffisent pour les exécutifs que nous visons. L'interopérabilité étant cruciale, les sockets ouvertes par *VSock* doivent pouvoir être connectées de façon transparente à des sockets standard TCP/IP. Le module *VSock* gère plusieurs protocoles, et sélectionne automatiquement le protocole adapté en fonction du matériel disponible. Pour chaque lien, en examinant les paires adresse IP–numéro de port, *VSock* détermine s'il est possible d'utiliser les communications Madeleine ou s'il faut se limiter au protocole standard TCP/IP. Les mécanismes d'établissement de connexion sont exactement ceux des sockets, et *VSock* utilise l'adressage IP standard. Du point de vue de l'application ou de l'exécutif, *VSock* se comporte comme des sockets standard, même si parfois la connexion est court-circuitée par *NetAccess*/Madeleine plutôt que TCP/IP. De cette façon, porter un exécutif sur *VSock* peut être réalisé simplement en détournant les appels des primitives socket standard vers leur équivalent *VSock*.

2.6. Groupes et circuits

Pour utiliser le réseau, les modules peuvent également passer par l'interface native de *NetAccess*. C'est le cas par exemple du module MPI décrit à la section 3 dérivé de l'implémentation MPICH/Madeleine [2]. D'une manière générale, les exécutifs qui ont une approche SPMD (MPI, une DSM, etc.) et qui n'ont pas besoin de mécanismes de connexion lien par lien préfèreront utiliser une interface de type *NetAccess*

native plutôt que *VSock*. L'interface de *NetAccess* fournit un accès à des canaux qui s'étendent sur tous les nœuds de la grappe où le code est déployé. Imaginons que l'on veuille déployer deux codes MPI sur une grappe, chacun sur la moitié des nœuds disponibles, et que les codes soient couplés par Corba. Pour que toutes les communications utilisent le réseau rapide, il est nécessaire que *NetAccess*/Madeleine soit lancé sur tous les nœuds ; cependant, nous désirons utiliser ces canaux sur un sous-ensemble de la topologie.
Pour gérer de tels cas, nous introduisons le concept de groupe dans PadicoTM. Nous définissons un groupe comme un ensemble de nœuds d'une grappe ou d'une machine parallèle ; l'ensemble des nœuds d'un groupe doit être sur le même canal Madeleine. Nous définissons un *circuit* comme un canal de communication *NetAccess*/Madeleine restreint à un groupe. Ces principes de gestion de la topologie réseau au-dessus de canaux statiques sont implémentés dans le module *Circuit* de PadicoTM.
Les circuits permettent de déployer MPI ou une DSM sur un sous-ensemble d'une grappe ; ainsi, la topologie logique n'a pas besoin de correspondre à la topologie physique. C'est une approche différente de la création de groupes MPI : quand MPI est déployé sur un circuit, le monde MPI (`MPI_COMM_WORLD`) est le circuit. Gérer la topologie au niveau de PadicoTM a pour avantage de ne pas demander de modification des applications MPI. De plus, les circuits peuvent être utilisés par d'autres modules que MPI, par exemple une DSM.
Pour gérer les exécutifs qui utilisent une approche SPMD, nous introduisons le pilote `multi` dans le gestionnaire de modules *Puk*. Le pilote `multi` réalise les opérations *Puk* (chargement, exécution, déchargement) de façon synchrone sur tous les nœuds d'un groupe. Il transforme donc les modules qu'il contient en modules SPMD, et effectue les synchronisations nécessaires.

2.7. Contrôle à distance

Puisque les applications sont chargées et déchargées dynamiquement sous forme de modules dans des processus déjà démarrés sur les nœuds des grappes, il est souhaitable de pouvoir contrôler ces opérations à distance. Padico comprend *PadicoControl*, un ensemble d'applications clientes destinées à diriger à distance des processus PadicoTM. *PadicoControl* existe sous deux formes : une interface graphique et une version en ligne de commande. Ces applications dialoguent avec un module contrôleur d'accès (*Gatekeeper*) chargé dans chaque processus PadicoTM et servant d'interface vers *Puk* pour les opérations de chargement, exécution et déchargement de modules. Le transport des requêtes est assuré au choix par Corba ou un mécanisme d'appel de procédures à distance utilisant XML, ce qui autorise l'interfaçage avec d'autres outils le cas échéant. Pour le moment, la sécurité est assurée par un mécanisme de clefs de session : un utilisateur ne peut piloter à distance que les processus qu'il a lancés lui-même.

3. Expérimentations avec différents exécutifs sur PadicoTM

Nous présentons dans cette section les expérimentations menées avec les exécutifs MPI, Corba et Java dans PadicoTM. Le module MPI de PadicoTM est dérivé de MPICH/Madeleine [2] ; très peu de changements ont été nécessaires pour passer de Madeleine aux circuits PadicoTM. L'implémentation Corba est fondée sur OmniORB 3 [1] d'AT&T. Le portage d'OmniORB sur *VSock* et *TaskManager* est simple car les interfaces fournies par ces deux modules sont un sous-ensemble des interfaces classiques pour l'accès au réseau et au multi-threading. Nous avons également porté MICO [13], une autre implémentation Corba, sur PadicoTM. Ceci illustre la flexibilité et l'ouverture de PadicoTM, mais cette implémentation reste marginale en raison de ses performances médiocres ; l'article [8] détaille les raisons de ces mauvaises performances. Enfin, la machine virtuelle Java Kaffe [10] a été portée sur *VSock* et *TaskManager*.
Nous mesurons les performances réseau de ces exécutifs sur des grappes composées de bi-Pentium II 450 MHz équipés des réseaux Ethernet-100 et SCI, et des bi-Pentium III 1 GHz équipés de Myrinet-2000. Les débits obtenus par les versions PadicoTM de MPI, Corba et sockets Java sont présentés dans la figure 3. Pour chacun des exécutifs, le débit est excellent, pour atteindre 240 Mo/s sur Myrinet-2000, soit 96 % du débit offert par Madeleine. Pour MPI, le débit monte à 75 Mo/s sur SCI contre 89 Mo/s pour Corba. La latence de MPI est mesurée à 11 μs sur Myrinet-2000 et 23 μs sur SCI, ce qui est exactement la même performance que MPICH/Madeleine [2] : PadicoTM ajoute le multiplexage des communications et la coopération entre exécutifs, mais ne pénalise pas les performances. La latence de Corba est mesurée à 20 μs sur Myrinet-2000 et 55 μs sur SCI. Même si ces chiffres sont légèrement supérieurs à ceux de MPI,

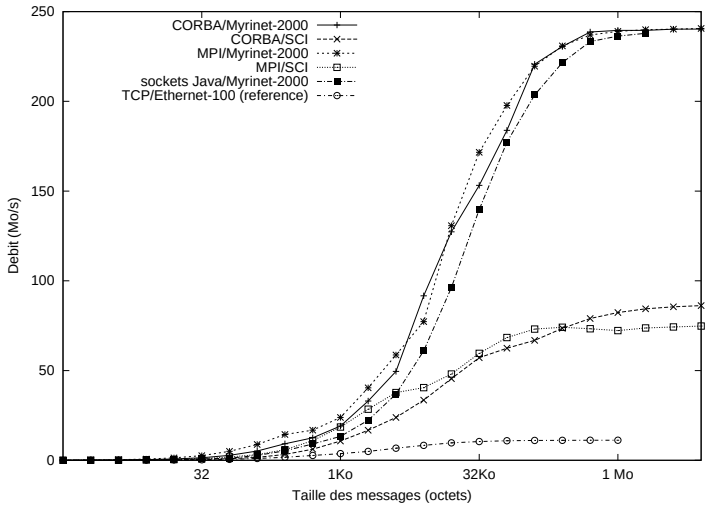

FIG. 3 – *Débits de Corba,* MPI *et sockets Java sur PadicoTM*

ils sont excellents comparés à la latence de 160 μs mesurée sur Ethernet-100. On remarque que les performances de MPI et Corba sont tout de même très similaires, ce qui valide notre approche d'utilisation conjointe de MPI et Corba pour une meilleure structuration des applications : utiliser Corba ne pénalise pas les performances.

4. Travaux connexes

Il existe à notre connaissance très peu de travaux dans le domaine de l'intégration d'exécutifs. La plupart des travaux concernent l'optimisation d'un exécutif en particulier.
Panda [4] est un environnement dédié aux exécutifs de programmation parallèle, notamment pour implémenter MPI et PVM. ADAPTIVE [20] (connu actuellement sous le nom ACE – *Adaptive Communication Environment*) cible les applications client/serveur et a servi notamment à l'implémentation Corba temps réel TAO (*The Ace Orb*). Ni l'un ni l'autre n'est conçu pour servir de base à des exécutifs parallèles *et* des exécutifs client/serveur. De plus, nous n'avons pas connaissance d'environnement qui permet l'utilisation de plusieurs exécutifs en même temps.
En ce qui concerne les réseaux haute performance, leur utilisation est courante dans le cadre du parallélisme avec MPI. Elle l'est beaucoup moins pour les applications client/serveur. En particulier, très peu de travaux ont été réalisés autour de Corba sur les réseaux haute performance. Citons TAO [11] qui a été porté sur les réseaux ATM, mettant l'accent principalement sur les aspect temps réel plutôt que sur la haute performance, CrispORB [9] développé par Fujitsu spécifiquement pour le réseau Synfinity-0, et un prototype d'OmniORB 2 [12, 16] sur ATM et SCI qui a été abandonné suite aux résultats décevants.

5. Conclusion et perspectives

Nous avons présenté PadicoTM, une plate-forme ouverte qui permet l'intégration d'exécutifs variés. Cette plate-forme permet l'exécution d'applications basées à la fois sur les paradigmes de programmation distribuée et parallèle sur les grilles de calcul, indépendamment des infrastructures réseau sous-jacentes. Une telle approche encourage les concepteurs d'applications à utiliser l'exécutif le mieux adapté à chaque tâche. Bien que cette plate-forme ajoute une couche logicielle supplémentaire entre les applications et le matériel, nous avons montré que le surcoût est insignifiant. Grâce à la généricité de PadicoTM, Corba peut bénéficier de la haute performance des réseaux rapides actuels ; c'est à notre connaissance l'implémentation Corba la plus rapide existante. Nous avons également montré que Corba et MPI peuvent atteindre des niveaux de performance tout-à-fait comparables, ce qui retire l'argument

principal qui existait en défaveur de Corba dans le choix d'un exécutif. PadicoTM ainsi que les modules Corba, MPI et Java sont implémentés. Le code sera distribué à partir du printemps 2002. Les travaux en cours concernent l'implémentation de modules DSM et CCM (*Corba Component Model*), ainsi que le domaine de la gestion des ressources.

Bibliographie

1. AT & T Laboratories Cambridge. – OmniORB Home Page. – `http://www.omniorb.org`.
2. Aumage (O.), Mercier (G.) et Namyst (R.). – MPICH/Madeleine: a true multi-protocol MPI for high-performance networks. *In : Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*. IEEE, p. 51. – San Francisco, CA, avril 2001.
3. Aumage (Olivier), Bougé (Luc), Denis (Alexandre), Méhaut (Jean-François), Mercier (Guillaume), Namyst (Raymond) et Prylli (Loïc). – A portable and efficient communication library for high-performance cluster computing. *In : IEEE Intl Conf. on Cluster Computing (CLUSTER 2000)*, pp. 78–87. – Technische Universität Chemnitz, Saxony, Germany, novembre 2000.
4. Bal (Henry), Benson (Gregory), Bhoedjang (Raoul), Langendoen (Koen) et Rühl (Tim). – Experience with a portability layer for implementing parallel programming systems. *In : International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pp. 1477–1488. – Sunnyvale, CA, août 1996.
5. Bougé (L.), Méhaut (J.-F.) et Namyst (R.). – Efficient communications in multithreaded runtime systems. *In : Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*. In conj. with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, pp. 468–482. – San Juan, Puerto Rico, avril 1999.
6. Danjean (V.), Namyst (R.) et Russell (R.). – Integrating kernel activations in a multithreaded runtime system on Linux. *In : Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*. In conjunction with *IPDPS 2000. IEEE TCPP and ACM*, pp. 1160–1167. – Cancun, Mexico, mai 2000.
7. Denis (A.), Pérez (C.) et Priol (T.). – Portable parallel corba objects: an approach to combine parallel and distributed programming for grid computing. *In : Proc. of the 7th Intl. Euro-Par'01 conf.* pp. 835–844. – Manchester, UK, août 2001.
8. Denis (Alexandre). – Corba et réseaux haute performance. *In : 13ième Rencontres Francophones du Parallélisme (RenPar'13)*, pp. 189–194. – Paris, avril 2001.
9. Imai (Yuji), Saeki (Toshiaki), Ishizaki (Tooru) et Kishimoto (Mitsushiro). – CrispORB: High performance CORBA for system area network. *In : Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pp. 11–18. – Redondo Beach, California, août 1999.
10. Kaffe: an OpenSource implementation of a Java Virtual Machine. – `http://www.kaffe.org`.
11. Kuhns (Fred), Schmidt (Douglas) et Levine (David). – The design and performance of a real-time I/O subsystem. *In : Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS99)*. – Vancouver, Canada, juin 1999.
12. Lo (Sai-Lai) et Pope (Steve). – *The Implementation of a High Performance ORB over Multiple Network Transports*. – rapport de recherche, Cambridge, Olivetti & Oracle Laboratory, mars 1998.
13. MICO, an OpenSource CORBA implementation. – `http://www.mico.org`.
14. Object Management Group. – The Common Object Request Broker: Architecture and Specification (Revision 2.5). – OMG Document formal/01-09-34, septembre 2001. `http://www.omg.org/corba/`.
15. PM2 High Perf. – World Wide Web document, `http://www.pm2.org`.
16. Pope (Steve) et Lo (Sai-Lai). – *The Implementation of a Native ATM Transport for a High Performance ORB*. – rapport de recherche, Cambridge, Olivetti & Oracle Laboratory, juin 1998.
17. Prylli (L.) et Tourancheau (B.). – Bip: a new protocol designed for high performance networking on myrinet. *In : 1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*. pp. 472–485. – Springer-Verlag. In conjunction with *IPPS/SPDP 1998*.
18. René (C.) et Priol (T.). – MPI code encapsulating using parallel CORBA object. *In : Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-99)*, pp. 3–10. – Redondo Beach, California, août 1999.
19. Rodrigues (Steven H.), Anderson (Thomas E.) et Culler (David E.). – High-performance local area communication with fast sockets. *In : USENIX '97*, pp. 257–274. – Anaheim, California, janvier 1997.
20. Schmidt (Douglas C.). – The ADAPTIVE Communication Environment, object-oriented network programming components for developing client/server applications". *In : Proceedings of the 12th Annual Sun Users Group Conference (SUG)*, pp. 214–225. – San Fransisco, CA, juin 1994.
21. World Wide Web Consortium. – Extensible Markup Language (XML) 1.0 (Second Edition). – W3C Recommendation, octobre 2000.