Expresso: transfert d'objets Java sur réseau haut débit

L. Courtrai – Y. Mahéo – F. Raimbault
Équipe Orcade
Laboratoire du Valoria
Université de Bretagne Sud
Tohannic, rue Yves Mainguy - 56000 Vannes (France)

16 mars 2000

Résumé

Ce rapport contient l'état d'avancement du projet *Expresso* qui vise à accélérer le transfert d'objets *Java* sur un réseau à haut débit. Les mesures que nous avons menées sur un réseau de stations de travail interconnectées par un réseau à haut débit (*Myrinet*) mettent en évidence le goulet d'étranglement constitué par les phases de sérialisation et de désérialisation classiquement employées dans un contexte d'objets distribués : ces phases entourant le transfert représentent 90% du temps total de communication sur un réseau haut débit. La solution que nous avons explorée pour exploiter les performances de tel réseau – réseau dont on peut sans risque prédire l'émergence très prochainement – consiste à se placer dans un contexte d'exécution qui rend inutile la sérialisation et la désérialisation. Dans l'hypothèse où le réseau est homogène (du point de vue des processeurs, des machines virtuelles et des systèmes d'exploitation) on expédie directement le blocs de mémoire contenant le graphe d'objet que l'on reloge à la même adresse dans la mémoire du processeur distant : les références intra-objet et inter-objet d'un même bloc restent valident et ne nécessitent plus aucun traitement. Nous avons démontré la faisabilité, la viabilité et l'intérêt en terme de performance de cette solution en réalisant une bibliothèque de communication de bas niveau en *Java* et en l'utilisant pour la distribution d'un algorithme de programmation génétique.

Table des matières

1 Introduction

Une application répartie programmée dans un langage à objets doit être capable de transporter un objet d'une machine à une autre. Un objet peut contenir des références vers d'autres objets. C'est donc un graphe d'objet, comportant éventuellement des cycles, qui doit être transporté.

Ceci implique généralement une sérialisation du graphe d'objets (l'objet proprement dit – l'objet racine – et les objets qu'il référence) du côté de l'émetteur, suivie du transfert d'un message et d'une phase de désérialisation du coté du récepteur.

La sérialisation consiste en un parcours du graphe d'objets à partir de l'objet racine. Au long de ce parcours, on enregistre dans un flot (fichier, réseau, tampon mémoire) toutes les informations nécessaires à la reconstruction du graphe d'objet.

Dans le contexte de Java, l'API standard du JDK propose deux interfaces. Elles servent notamment lors transfert d'objet mis en oeuvre au sein du passage de paramètres par copie du RMI. L'interface `Serializable` permet de rendre transparent la sérialisation. Lorsque le type d'un objet n'est pas connu au moment de la sérialisation, on passe par une phase d'introspection pour le découvrir et ainsi appliquer une linéarisation adaptée. L'interface `Externalizable` permet quant à elle d'éviter l'introspection et de spécialiser la sérialisation. C'est l'utilisateur qui ajoute deux méthodes à la définition de la classe concernée, méthodes qui permettent à l'objet de se linéariser lui-même et se de reconstruire. À noter que l'objectif de la sérialisation proposée dans le JDK va plus loin puisque la classe déclarée de la référence sur l'objet reçu peut n'être que « compatible » avec la classe de l'objet présent dans le flot.

Lors de la sérialisation, au minimum, seules les données renfermées dans l'objet sont enregistrées. Mais dans le cas général, il faut en plus enregistrer des informations sur les classes des objets afin que le système récepteur puisse reconstruire l'objet, notamment lorsque l'objet envoyé est d'un sous-type de celui déclaré à la réception.

Plusieurs propositions ont été faites pour optimiser le processus de sérialisation-désérialisation [PH99, Que97, Bar97, MvNV+99]. Elles visent d'une part à compacter les informations enregistrées, c'est-à-dire diminuer la taille du message transmis et d'autre part à accélérer la sérialisation en évitant l'introspection, par la spécialisation du sérialisateur en fonction de la classe de l'objet (génération automatique d'interpréteurs spécialisés, compilation). L'idéal visé dans ce cadre se limite au parcours du graphe d'objets durant lequel on recopie seulement les données dans le flot, toutes les informations supplémentaires sur les types étant implicites.

Sur un réseau classique (réseau local Ethernet ou réseau grande distance), la sérialisation-désérialisation appliquée dans le contexte de Java est souvent considérée comme un obstacle à l'efficacité de codes distribués. L'application d'optimisations peut donner des résultats satisfaisant tant que l'on reste sur un réseau classique car le temps de communication domine largement dans le processus comprenant le parcours du graphe avec copie et le transfert. Cependant, lorsque les performances du réseau augmentent, le temps passé à parcourir le graphe d'objets et à copier les données dans le flot d'octets devient proportionnellement plus important voire majoritaire.

La série d'expérimentations décrite ci-après le confirme. On cherche à chiffrer la proportion de temps passé dans la sérialisation dans transfert d'un objet Java sur une grappe de stations de travail relié par un réseau à très haut débit de type Myrinet. Le tableau de comparaison ci-dessous montre les temps de sérialisation, de désérialisation et de transfert pour un graphe d'objets simple (1 référence, 2 chaînes de caractères, 1 entier pour un volume de données de 34 octets) ainsi que pour un graphe plus complexe (arbre binaire de 1000 nœuds comprenant chacun deux références et un entier). Les mesures ont été faites avec la sérialisation-désérialisation standard du JDK 1.2 (interface `Serializable` et `Externalizable`) ainsi qu'avec une version optimisée de la classe `Externalizable` écrite à l'Université de Karslruhe (UKA, [PH99]). Les transferts sont effectués via MPI interfacée avec Java, sur un réseau Myrinet 1 Gb/sec.

Les mesures couvrent un aller-retour du graphe soit la séquence sérialisation – transfert – désérialisation – sérialisation – transfert – désérialisation. Les temps sont donnés en μs. La dernière colonne indique le pourcentage de temps passé en sérialisation-désérialisation par rapport au temps total.

Graphe simple					
Interface	Taille msg.	Sér.	Désér.	Transfert	% ser/deser
JDK Serializable	166 octets	397	627	129	89 %
JDK Externalizable	51 octets	168	245	107	80 %
UKA Xserializable	46 octets	89	149	106	70 %
Arbre binaire					
Interface	Taille msg.	Sér.	Désér.	Transfert	% ser/deser
JDK Serializable	11089 octets	54878	54277	797	99 %
JDK Externalizable	6097 octets	1837	9904	598	95 %
UKA Xserializable	6010 octets	1345	2397	593	86 %

On voit bien que la part liée à la sérialisation est extrèmement importante, même dans le cas d'objets simples. La faible latence des communications n'arrivant pas à masquer ce côut de calcul. L'effort consacré à l'optimisation de la sérialisation dans la classe XSerializable semble peu rentable dans la mesure où le constat n'en est que très partiellement modifié.

Nous pensons que dans certaines configurations de réseau (*e.g.* réseaux locaux à haut débit, nœuds homogènes, machines parallèles), la méthode de sérialisation elle-même peut être remise en cause. En effet, le coût relatif à la prise en compte de l'hétérogénéité, de l'interopérabilité ou de l'évolutivité des classes peut être allégé. Nous proposons donc dans la suite une alternative à travers la bibliothèque Expresso. L'objectif principal est ici la performance du transfert du graphe d'objets.

2 Transfert ISO-adresse

Nous nous plaçons dans le contexte de l'exploitation d'une grappe de stations de travail Unix homogènes reliées par un réseau à haut débit. L'objectif est de permettre le transfert rapide d'un objet Java entre deux nœuds du réseau. Une amélioration drastique des per-

formances du transfert d'objets comme celui qui est effectué lors du passage de paramètre d'un RMI peut être obtenue en supprimant les phases de sérialisation–désérialisation. Nous proposons d'utiliser une bibliothèque *Java* nommé *Expresso* permettant de supprimer le parcours du graphe et de transférer directement la représentation en mémoire du graphe d'objets.

2.1 Espace ISO-adresse

Pour un transfert direct de mémoire, nous mettons en place un espace *iso-adresse*. C'est un intervalle d'adresses virtuelles identique sur chacun des nœuds du réseau. Les pointeurs placé dans cette zone et pointant vers une adresse à l'intérieur de cette zone peuvent être déplacés d'un nœud à un autre sans modification ce qui nous permet de transporter la représentation mémoire d'un graphe d'objet d'un nœud à un autre et de pouvoir exploiter ce graphe d'objets tel quel sur le nœud d'arrivée (voir figure 1). Le concept est similaire à celui utilisé pour la migration de thread dans [ABN99].

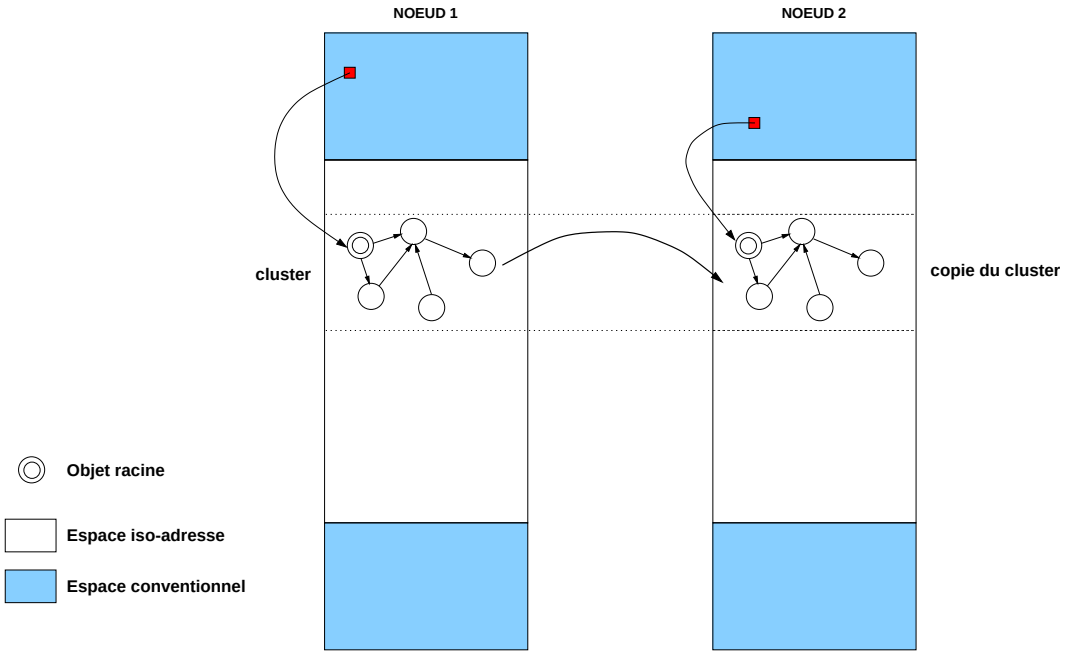

FIG. 1 – Espace iso-adresse

Une plage d'adresse est donc réservée sur chacun des processus. À noter que la réservation porte sur de la mémoire virtuelle. Le fait de bloquer un grand nombre d'adresses dans l'espace d'adressage du processus n'est pas pénalisant si l'on considère que l'espace total est très grand. Il est actuellement typiquement de 4 Go (machines 32 bits) et l'utilisation d'adresses sur 64 bits – qui devrait se généraliser dans un avenir proche – autorise un espace d'adressage quasi illimité.

2.2 Clusters

L'espace iso-adresse est divisé en blocs appelés *clusters ISO*. Un cluster ISO est l'unité de communication sur le réseau. Il est destiné à recevoir un (éventuellement plusieurs) graphe d'objets. Un graphe d'objet est entièrement placé dans un seul cluster (pas de références inter-cluster).

Sur l'ensemble du réseau, un cluster est repéré par un couple (numéro du nœud d'origine, numéro d'ordre de création du cluster sur ce nœud). Il possède un objet racine (point d'entrée du graphe) modifiable.

2.3 Communication

Seuls les objets devant être transportés sont placés dans des clusters, les autres restent placés dans le tas d'objets géré par la JVM. Un cluster est créé sur un nœud, on peut en envoyer une copie sur un autre nœud. Un nœud peut recevoir une copie d'un cluster directement depuis le nœud d'origine ou depuis un nœud qui a déjà reçu le cluster. Les émissions sont non bloquantes et les réceptions bloquantes.

La réception d'une nouvelle copie du cluster efface l'ancienne. Sur un nœud, à un instant donné, une seule copie d'un cluster est donc accessible. Après réception d'une copie du cluster, une référence sur l'objet racine qu'il contient peut être obtenue. Il faut noter que l'on ne dispose pas ici d'un mécanisme classique d'envoi-réception d'objet. Le modèle de communication se place à un niveau plus bas, celui des « variables » – au sens de zones de mémoire. On dispose d'un nombre fixe de ces variables (les clusters), l'envoi-réception permettant de mettre à jour une variable sur un nœud à partir du contenu présent sur un autre nœud. La cohérence des contenus des clusters n'est pas maintenue par *Expresso*, il est à la charge de l'utilisateur.

L'utilisation typique d'un cluster est donc la suivante.
Sur le nœud d'origine émetteur :

1. création d'un cluster
2. création des objets formant un graphe à l'intérieur de ce cluster
3. désignation d'un objet racine dans le graphe
4. émission du cluster

Sur le nœud récepteur :

1. réception du cluster
2. récupération d'une référence sur l'objet racine.

Les point 1 et 2 sont à omettre sur l'émetteur lorsque celui-ci transmet un cluster qu'il a déjà reçu.

3 *Expresso*

Expresso est une bibliothèque de classes *Java* spécialisées utilisant ce concept de transfert ISO–Adresse. pour le transfert ISO adresse d'objets *Java*.

La totalité de l'espace ISO adresse est découpé en n zones machines. Chaque machine écrit dans sa zone ISO et peut lire celles des autres. Chaque espace ISO d'une machine est lui même découpé en clusters.

3.1 Interface

L'interface de la bibliothèque *Expresso* contient la classe `ClusterISO` dont l'interface est présentée figure 2.

```
public class ClusterISO{
  // Connexion réseau et réservation memoire ISO
  public static void init(String[] argvs);
  // Numero de noeud sur le réseau
  public static int MyNode;
  // Déconnexion réseau et libération memoire
  public static void quit();
  // Creation d'un nouveau cluster
  public ClusterISO();
  // Positionnement du cluster comme cluster courant
  public void setCurrent();
  // Envoie du cluster sur une autre machine
  public void send(int node)
  // Reception d'un cluster d'un noeud
  public static ClusterISO recv(int origine, int numCluster)
  // Reception d'une copie d'un cluster d'un autre noeud
  public static ClusterISO recv(int origine, int numCluster, int emetteur)
  // Creation d'un objet dans le cluster courant
  public static Object newObject(Class uneClasse);
  // Creation d'un objet tableau dans le cluster courant
  public static Object [] newArray(Class laClasseDesElts,int nbElts);
  // Recuperation de l'objet racine d'un cluster
  public Object getRootObject();
  // Positionnement de l'objet racine d'un cluster
  public void setRootObject(Object unObjet);
  // Vidage du cluster
  public void empty();
```

FIG. 2 – Interface de la classe `Cluster`

3.1.1 Les primitives *Expresso*

Connexion La méthode de classe `init(String[] argvs)` initialise l'espace ISO adresse en réservant le maximum de mémoire disponible auprès du système d'exploitation et effectue la connexion entre les nœuds. L'attribut de classe `myNode` donne le numéro du nœud .

La méthode de classe `Quit()` termine proprement l'application en fermant préalablement les connexions entre les machines.

Création des clusters Avant toute manipulation de cluster, le programmeur créé un nouveau cluster où seront effectuées les futures instanciations. Dans un cluster un objet principal `root`, permet lors du transfert de clusters entre machines, de référencer un des objets du cluster. Tous les objets referencés directement ou indirectement par cet objet sont alors accessibles.

Le constructeur de la classe `ClusterISO` crée un nouveau cluster. Une fois créé, la méthode d'instance `setCurrent()` positionne le cluster comme le cluster courant où seront mises les futures créations d'objets.

La méthode de classe `changeCluster` change le cluster courant.

Instanciation des objets L'opérateur **new** de *Java* doit alors être remplacé par la méthode `newObject(Class uneClasse)` de classe `ClusterISO`. Les objets ainsi créés sont alors mis dans le cluster courant. Les autres, ceux instanciés par l'opérateur **new**, restent dans l'espace conventionnel.

La primitive `newObject` retourne une poignée sur le nouvel objet. L'objet possède une structure identique à un objet crée par la machine virtuelle et peut donc être gérée par celle ci comme un objet *Java* ordinaire.

La méthode `newArray(Class laClasseDesElts,int nbElts)` crée un objet tableau dans le cluster courant. Le programmeur spécifie le type et le nombre d'éléments du tableau.

3.1.2 L'envoi et réception de cluster

La méthode `send` envoie un cluster vers un autre noeud. La primitive est non bloquante Un objet racine doit préalablement être spécifié par la méthode `setRootObject(Object unObjet)`

La méthode de classe `reveive(int noeudOrigine, int numCluster)` attend un cluster spécifique d'un noeud. Le couple `(noeudOrigine,numCluster)` identifie le cluster voulu. Une surcharge de cette méthode, `reveive(int noeudOrigine, int numCluster, int noeudEmette` permet de recevoir une copie de cluster venant une machine différente du noeud d'origine du cluster. Les deux primitives sont bloquantes.

Si la valeur d'un paramètre est égale à `ANY` la primitive attend un cluster de n'importe quel noeud ou n'importe quel cluster d'un noeud donné, (voir les deux).

A partir de ce cluster la méthode `getRootObject()` retourne une poignée sur l'objet racine du cluster.

3.1.3 Exemple

La figure 3 contient un exemple minimal de programme *Java* utilisant la bibliothèque *Expresso* pour transférer un arbre binaire contenant trois noeuds entre deux machines. Le même programme s'exécutant sur chacune des machines (modèle de programmation parallèle SPMD), une conditionnelle portant sur le numéro de noeud détermine le code réellement exécuté par chaque noeud. Le noeud 0 réserve un cluster, y crée un arbre binaire, le remplit puis l'envoie au noeud 1. Le noeud 1 crée un cluster et y recoit le cluster envoyé par le noeud 1.

```
import expresso.*;       // Utilisation du package Expresso
// Exemple de programme de transfert du arbre entre 2 machine
//  avec la librairie Expresso
public class TestISO {

  public static void main(String argv[]) {

    ClusterISO.init(argv); // initialise Espresso
    Btree arbre; // arbre binaire a tranferer

    if (MyNode == 0) { // code de la machine 0
      ClusterISO unCluster = new ClusterISO(); // demande un cluster
      arbre =(Btree)Cluster.newObject(Btree.class);
      arbre.init(); // initialise l'arbre construit
      arbre.addValue(10);// ajoute des elements dans l'arbre
      arbre.addValue(25);
      arbre.addValue(3);
      unCLuster.setRootObjet(arbre);
      unCluster.send(1); // envoie du graphe sur le noeud 1

    } else { // code la machine 1
      // lecture du pemier cluster de la machine 0
      ClusterISO unCluster = new ClusterISO(0,ANY);//un cluster de la machine 0
      // extrait l'objet principal du cluster
      arbre = (Btree)unCluster.getRootObject();
      System.out.println(arbre);//affiche le contenu de l'arbre
    }
    ClusterISO.quit();// termine l'application
  }
}
```

FIG. 3 – Transfert d'objets avec Expresso et MPI

3.2 Implantation

La plate-forme d'expérimentation est un réseau de PC (Pentium II à 400 Mhz) sous Linux (noyau 2.0.36). Le réseau est de technologie Myrinet [BCF+95], développée par la société Myricom. La configuration repose sur des cartes d'interface PCI 32 bits à 33Mhz et un commutateur 8x8 ports LAN. Le commutateur possède une bande passante agrégée suffisante pour supporter simultanément un débit maximal de 1,28 Gbits/s sur tous les liens entrant et sortant.

Les communications de bas niveau sur ce réseau sont assurés par la bibliothèque propriétaire GM (version 1.01). Cette bibliothèque sert de support à la bibliothèque standard MPI [MPI] (portage MPICH 1.2..3 fourni par Myricom), que nous utilisons dans *Expresso*.

Expresso utilise *Mpi* ([MPI]) pour le transfert des clusters les machines. *Mpi* est une bibliothèque de fonctions C (ou Fortran) de communication entre les noeuds d'un réseau. Elle offre principalement une ensemble de primitives d'envoi et de réception de message de type (`send` et `receive`).

Expresso est associé à la version *Kaffe* de *Java*, dont il dépend pour les structures internes. Les objets mis dans les clusters ont la structure C des objets KAFFE.

Les méthodes de la classe `ClusterISO` sont essentiellement implantées par des méthodes natives Java, JNI (méthode *Java* dont le corps est une fonction C).

3.2.1 Gestion mémoire

Le run-time *Expresso* gère lui-même la mémoire à coté de celle de Java pour son espace ISO adresse.

L'espace d'adressage d'un processus Unix est de 2^{32} octets (sur un processeur 32). Un grand espace libre (non encore alloué) se situe entre la pile et le tas. L'espace ISO adresse est réservé dans cet emplacement au sommet du segment de donnée. Il représente l'ensemble des espaces ISO des machines de l'application. L'espace est réservé par un appel système `brk` (qui positionne la taille du segment de donnée).

L'espace ISO est lui même découpé comme indiqué sur la figure 4.

Chaque machine possède un emplacement pour la copie des clusters des autres machines.

La méthode newObjet Un objet *Expresso* alloué par la méthode `newObject` est mémorisé dans l'espace ISO de la machine locale. Mais pour qu'il puisse être géré par la machine virtuelle, il possède une structure conforme à celle des autres objets *Java* (voir figure 5).

Les attributs de l'objet sont sur les octets suivants la structure. (On ne parlera pas ici des informations concernant le GC qui sont mis avant la structure `Hjava_lang_Object`.).

Pour que *Kaffe* puisse voire cet espace mémoire comme un objet *Java*, il suffit de mettre le champs `dtable` sur l'adresse de la structure `dtable` comprenant un lien vers l'objet classe et un tableau des méthodes. Cette adresse est connue dans la structure d'une classe *Kaffe* qui est passée en argument de la méthode `newObjet(Class c)`.

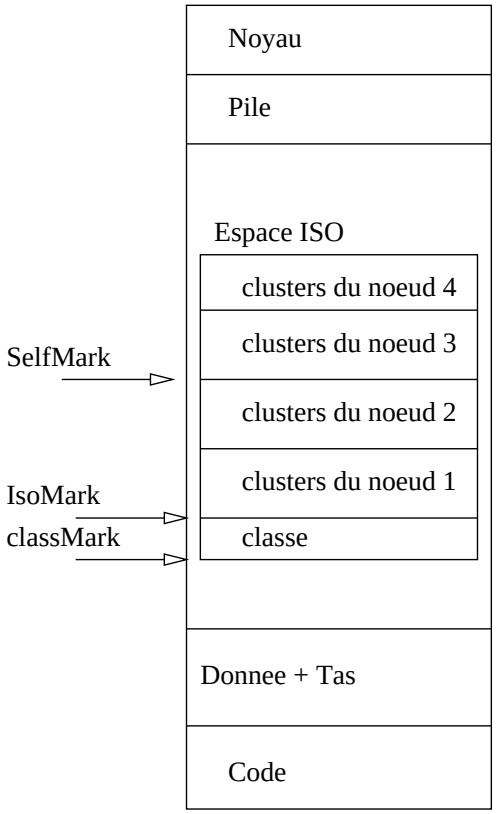

FIG. 4 – organisation de l'espace ISO

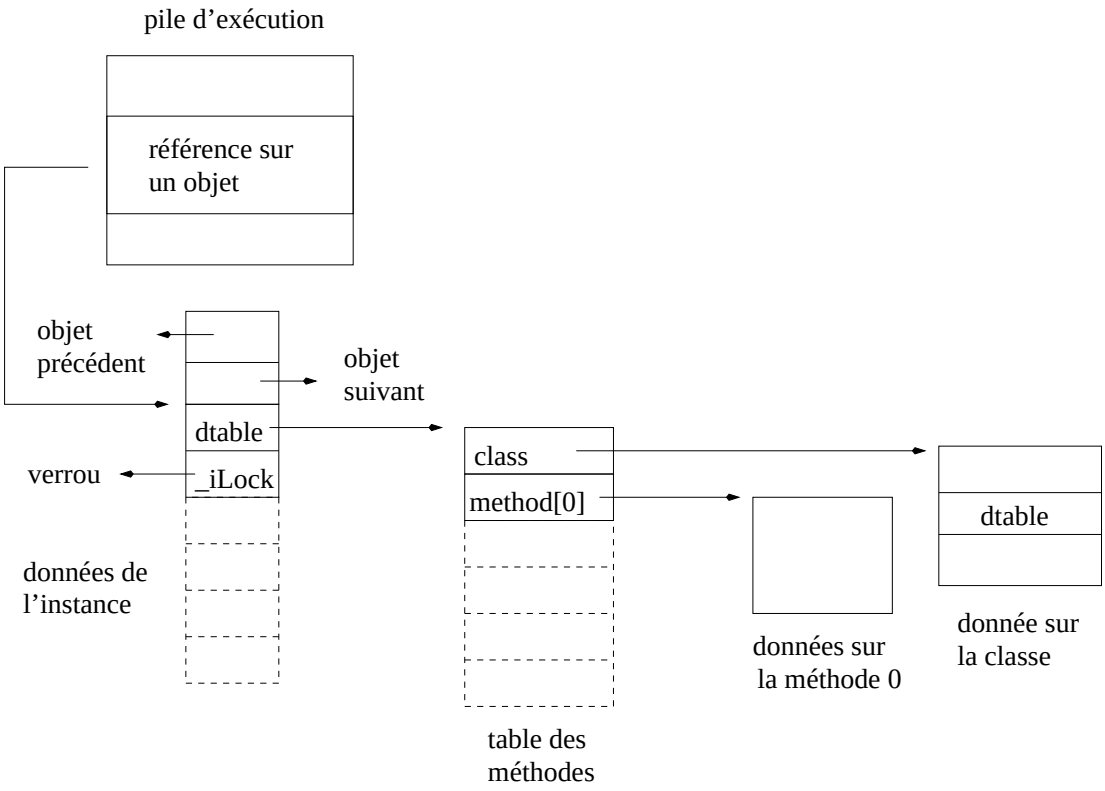

FIG. 5 – Structure de données des objets de la machine virtuelle *Kaffe*

Transfert de cluster L'appel de la méthode `send` sur une machine, associé à celui de la méthode `receive()` sur une autre machine est une copie mémoire entre ces deux machines. La copie s'effectue directement par les primitives de la bibliothèque +MPI `MPI\_Send` et `MPI\_Recv`.

Pour transférer un graphe d'objets entre deux machines le système envoie donc une partie de son espace vers l'autre machine. A la réception la primitive `Recev` replace cet espace à la même adresse. Cette adresse est calculable en fonction du nom du cluster. Tous les liens entre objets du cluster (attributs références sur d'autres objets) restent donc valides (il contiennent les mêmes adresses mémoire).

Le lien `dtable` doit aussi être valide. Un espace spécifique dans l'espace ISO Adresse est réservé pour stocké une copie de cette structure. Au début de l'application le système charge via son propre `Classloader` les classes de l'application par anticipation. Il effectue en même temps en une copie de cette structure dans l'espace ISO.

La structure `_dispatchTable` possède un lien vers la structure de la classe ; puis les adresses des fonctions implantant les méthodes. La méthode `newObject` met alors le champ `dtable` de l'objet sur l'adresse de la copie de l'espace ISO.

Les classes étant chargées sur toutes les machines, une copie est donc toujours présente au même emplacement mémoire. Le lien `dtable` d'un objet reste valide après son transfert.

3.2.2 Mesure des performances du prototype

Temps de primitives Nous cherchons ici à mesurer les temps d'appel des primitives hors communication comparés à ceux de Kaffe.

des primitives hors communication comparés à celles de Kaffe.

Les temps sont pris sur un PC ayant l'architecture suivante : Pentium II 400 Mhz, chipset BX, mémoire 128 Mo, disque IDE 4,5 Go sous le système Linux redhat 5.2 (noyau 2.0.36).

Nous utilisons la version 1.0.5 de Kaffe.

L'appel à la primitive `newObject(Class uneClasse)` coûte actuellement 5 fois plus de temps de l'opérateur Java **new** ; ceci s'explique par le coût supplémentaire lié à l'appel d'une méthode native (Jni Java). Une fois l'objet créé, les temps d'accès aux attributs ou ceux des appels aux méthodes de l'objet sont identiques à ceux calculés sur des objets Java créés par le **new**.

Temps de tranfert d'objets Nous mesurons ici les temps de transfert d'un ensemble d'objets entre deux machines connectées par MPI. Nous cherchons à comparer les transferts de notre bibliothêque par rapport aux méthodes classiques de sérialisation-désérialisation offertes par Java

Nous transférons ici des arbres binaire de recherche (Dans un objet nœud de l'arbre nous avons une donnée pour deux références et la motié des références sont nulles.

Les temps sont donnés en micro-secondes

Nombre d'objets	*Serialisable*	*Externalizable*	*Expresso*	$\frac{Expresso}{Serialisable}$	$\frac{Expresso}{Externalisable}$
10	11700	8900	210	55	42
100	21300	10800	470		
1000	143100	35300	1700		
10000	5083000	1208000	16200		

Les temps de transferts d'objets pour la librairie Expresso sont proches de ceux de simple envois de message. Il n'y pas ici aucun autre traitement effectué de type emballage et de déballage d'objets que l'on a dans les versions `Serialisable` ou `Externalizable` .

4 Application

Les applications pouvant bénéficier du mécanisme de communication par ISO-adresse de *Expresso* doivent satisfaire actuellement plusieurs contraintes techniques.

- L'application est distribuée sur un réseau de processeurs pour des raisons de performance ou de contraintes géographiques.
- Le réseau de processeurs et de systèmes d'exploitation utilisé est homogène.
- Le réseau dispose d'un mécanisme de transfert à haut débit, ce qui nous restreint à des réseau locaux.

En dehors de ces contraintes techniques, l'intérêt de *Expresso* par rapport à des solutions classiques de sérialisation et de désérialisation apparait clairement quand les structures de données échangées entre les noeuds sont assez complexes – en tout état de cause non réduites à des tableaux de nombres – et qu'elles mettent en jeu des types variés.

Des applications répondant à ces conditions sont par exemple des outils de travail coopératifs, ou des calculs intensifs sur des structures non linéaires. Cependant le modèle de programmation offert par *Expresso*étant de très bas niveau, cette bibliothèque est réservée à des programmeurs possédant une bonne expérience du parallélisme et de la distribution.

A titre d'illustration on choisit un cas d'école, une application très simple réalisant un calcul d'optimisation sur des arbres. Ce calcul est un algorithme génétique particulier, la programmation génétique. La partie 4.1 rappelle les principes des algorithmes génétiques. La partie 4.2 décrit l'algorithme séquentiel de programmation génétique et sa parallélisation à l'aide d'*Expresso*. La partie 4.3 conclue par l'énoncé des performances obtenues grace à *Expresso*.

4.1 Les algorithmes génétiques

Les problèmes résolus par les algorithmes génétiques appartiennent à la classe des problèmes d'optimisation. Leur objectif est de maximiser une fonction d'évaluation, appelée aussi indice de qualité, pour des valeurs appartenant à l'espace de recherche. L'étude mathématique des problèmes d'optimisation est souvent limitée aux fonctions continues et différentiables. Ce n'est pas le cas d'une fonction d'évaluation dont l'espace de recherche

n'est plus limité à des sous-ensembles de $\mathbb{R}^n$ mais peut s'étendre à des espaces discrets (espace des arbres n-aires, espace des séquences binaires de longueur 10, etc.).

En dehors des problèmes pour lesquels ont connait une solution analytique, des méthodes générales, telles que la méthode du gradiant ou la méthode du simplex, sont utilisées pour converger vers des solutions optimales. Les algorithmes génétiques appartiennent à cette classe d'algorithmes d'optimisation stochastique.

4.1.1 Principes

Un algorithme génétique est un processus itératif qui consiste à faire évoluer un ensemble de solutions potentielles vers une (ou plusieurs) solution optimale. L'idée originale de J. Holland [Hol75], popularisée par Goldberg en 1989 [Gol89], est de simuler le mécanisme de l'évolution biologique pour faire évoluer les solutions. La terminologie employée pour décrire le principe d'un algorithme génétique est donc empruntée à la génétique.

- Les solutions potentielles sont appelées des chromosomes.
- Un ensemble de solutions potentielles est une population. La population initiale est aléatoire.
- Une génération est une population à une étape de l'évolution.
- L'indice de qualité (*fitness*) d'un chromosome est une valeur abstraite permettant de classer les chromosomes par ordre décroissant en tant que solution à un problème.
- Le passage d'une génération à l'autre est obtenu après application successive aux chromosomes de trois opérateurs standards : la reproduction, le croisement et la mutation.
- l'opération de reproduction consiste à dupliquer des individus de la population courante. Plusieurs algorithmes de reproduction sont utilisés. Ils diffèrent principalement par la méthode de selection des chromosomes qui se reproduisent.
- Le croisement consiste à combiner deux chromosomes pour en créer deux nouveaux qui contiennent chacun une partie des chromosomes parents.
- L'opération de mutation consiste à modifier de manière aléatoire la valeur d'une composante d'un chromosome.
- Les opérateurs sont guidés par un certain nombre de paramètres dont le choix conditionne la convergence de l'algorithme vers une solution optimale : la probabilité de croisement et la probabilité de mutation.

4.1.2 La programmation génétique

La programmation génétique repose sur l'utilisation d'algorithmes génétiques pour construire automatiquement des programmes répondant à un critères exprimable par une fonction d'évaluation. L'exemple le plus simple, que nous allons réaliser avec *Expresso*, consiste à rechercher une expression arithmétique qui approche une fonction d'une variable, évaluable numériquement.

Dans ce cadre, les chromosomes sont constitués d'arbres binaires dont les noeuds sont des opérateurs (en nombre fini) et les feuilles sont une constante ou une variable.

4.2 Mise en oeuvre avec *Expresso*

L'intérêt de la parallélisation d'un algorithme génétique apparait clairement dès que les expérimentations nécessitent de nombreuses itérations sur des populations conséquentes pour approcher une solution optimale. Le temps de calcul augmentant, il est naturel de répartir la charge sur plusieurs processeurs.

Nous présentons dans un premier temps une version séquentielle de l'algorithme utilisé. Nous montrons ensuite la méthode de parallélisation retenue, le *scatter-gather* et la manière de l'appliquer avec *Expresso*.

4.2.1 Présentation de l'algorithme séquentiel

La figure 6 contient la partie principale de l'algorithme séquentiel de la programmation génétique. Deux variables sont utilises alternativement pour la population : l'une représente la population courante et l'autre contient la population en cours de construction.

Comme tout algorithme génétique, la programmation génétique se décompose en trois phases génériques :

1. génération aléatoire de la population initiale : une liste d'arbres d'opérateurs unaires et binaires de hauteur maximale fixée.
2. itération sur le nombre de générations (fixé statiquement ou dynamiquement à partir d'une condition sur l'évaluation du meilleur individu obtenu).
3. affichage du meilleur individu obtenu.

Chaque itération consiste à créer une nouvelle génération à partir de la précédente en réalisant séquentiellement les opérations suivantes.

- Croisement des arbres, deux par deux : un noeud est choisi au hasard dans deux arbres et les sous-arbres correspondants sont échangés.
- Mutation des arbres : chaque noeud d'un arbre est succeptible de changer de contenu avec une probabilité déterminée par le taux de mutation fixé au départ.
- Reproduction des arbres : on calcule de la probabilité de reproduction de chaque arbre [Bri]. Cette probabilité est proportionnelle à la valeur de la fonction d'évaluation appliqué à l'arbre. Elle tend à favoriser la reproduction des arbres dont l'évaluation est la meilleure.

La figure 7 illustre les opérations génétiques quand elles sont appliquées à des arbres d'opérateurs.

L'indice de qualité d'un individu (un arbre) est obtenu en effectuant l'évaluation de l'individu pour plusieurs valeurs de la variable et en calculant la somme des écarts quadratique moyen entre la valeur obtenue par évaluation de l'individu et la valeur de la fonction à approcher. La fonction d'évaluation de chaque individu consiste à effectuer un parcours récursif descendant gauche-droite de l'arbre en appliquant les opérateurs rencontrés aux valeurs des feuilles.

```
Population population0= new Population();
Population population1= new Population();
// initialisation aleatoire
population0.generate();
// iteration sur le nombre de generation
for (int i=0; i<n ; i++){
  population1= population0.crossCells(); // croisements
  population1.mutateCells();             // mutations
  population0= population1.reproduceCells(); // reproductions
}
// affichage de la meilleure solution obtenue
population0.print_best();
```

FIG. 6 – Algorithme séquentiel de la programmation génétique

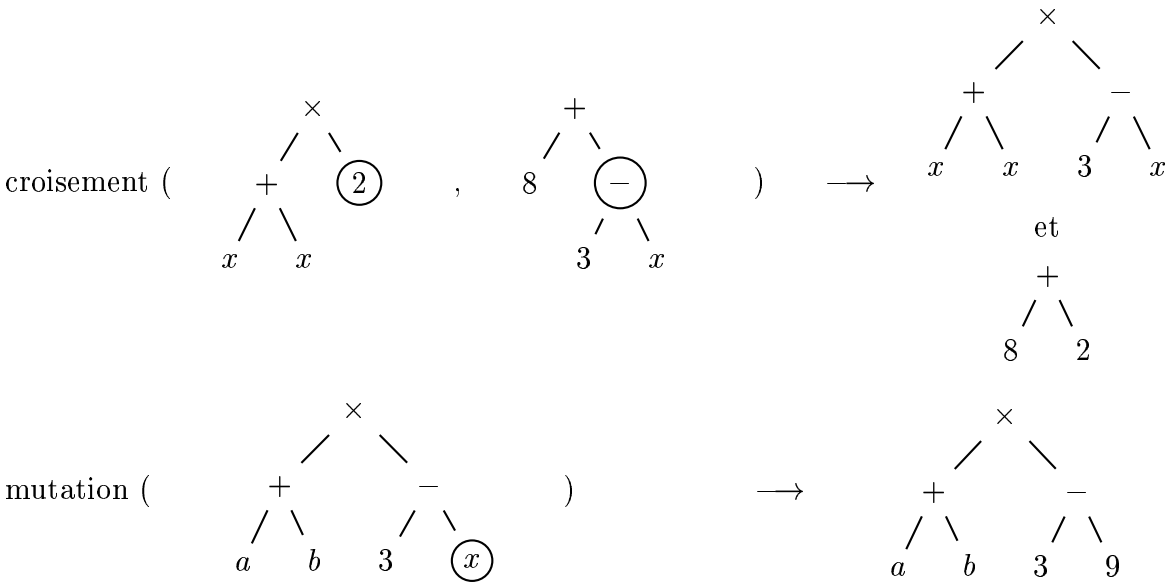

FIG. 7 – Opérations génétiques appliquées à des arbres d'opérateurs

4.2.2 Distribution des données avec *Expresso*

Une méthode de parallélisation directe de cet algorithme consiste à appliquer le paradigme du *scatter/gather* (voir figure 8) : Les données initiales sont dupliquées sur tous les nœuds mais chaque nœud n'est propriétaire (pour un accès en écriture) que d'une partie des données. Tous les noeuds effectuent des transformations sur les données dont ils sont propriétaires, en utilisant éventuellement des données non propriétaires. Les résultats de chaque noeud sont diffusés à l'ensemble des autres noeuds pour reconstituer un nouvel ensemble de données commun. Ce processus est réitéré systématiquement jusqu'à détermination d'une condition d'arrêt globale.

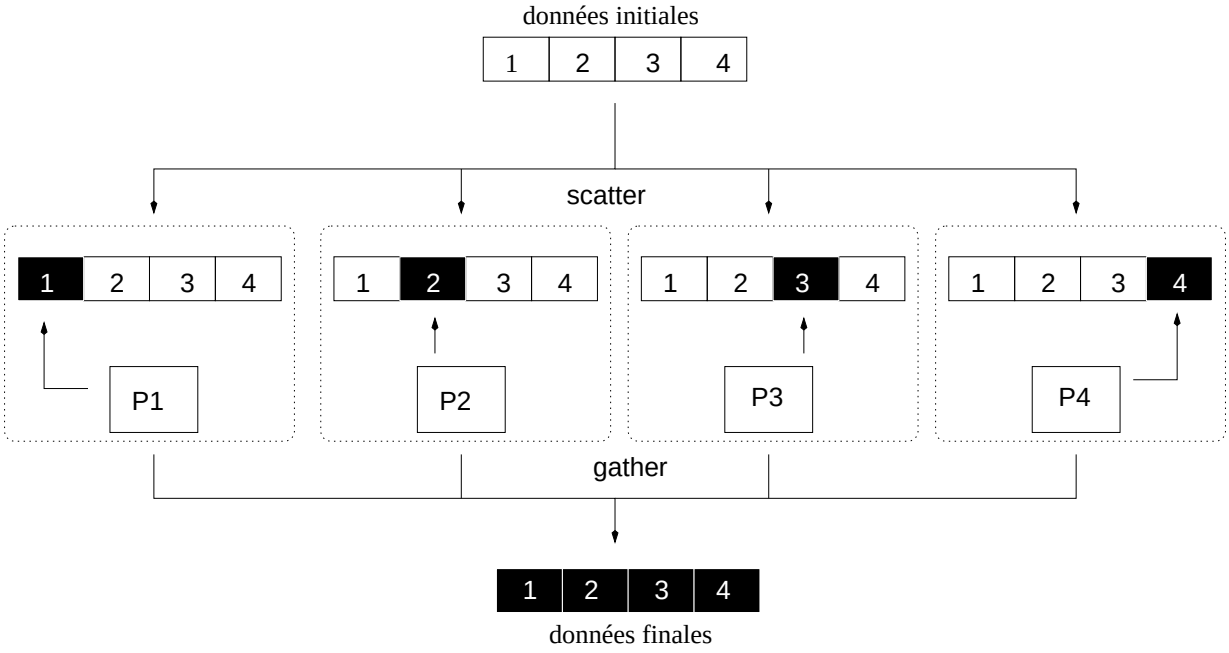

FIG. 8 – Paradigme du scatter/gather

Cette méthode a été choisie pour la facilité avec laquelle elle peut être appliquée à l'algorithme séquentiel, sans risque de se perdre dans les considérations complexes de synchronisation fréquemment rencontrées dans les algorithmes parallèles. Cependant l'efficacite de cette méthode suppose :

1. un moyen rapide de tranfert des données et de collecte des résultats,
2. une répartition égale des calculs entre les noeuds,
3. une mémoire suffisante sur chaque noeud.

Ces contraintes étant satisfaites par l'utilisation d'*Expresso* et par le principe des algorithmes génétiques, on peut appliquer la méthode du *scatter/gather* et accélérer considérablement la programmation génétique.

La figure 9 contient les modifications à apporter à la définition de la classe `Population`

pour placer ses instances dans des clusters et échanger les parties de population sur lesquels les différents noeuds du réseau travaillent de manière indépendante.

La figure 10 contient l'algorithme de chaque noeud de l'algorithme de la programmation génétique parallélisée.

Au début de chaque itération, la population courante est diffusée à l'ensemble des noeuds. Chaque noeud peut alors effectuer de manière indépendante les opérations de croisement puis de mutation sur la partie de la population qui lui est assignée. Par contre l'opération de reproduction nécessitant de faire la somme des indices de qualité de l'ensemble des individus de la population courante, chaque noeud diffuse auparavant la partie de population qu'il a modifié. Ensuite chaque noeud effectue l'opération de reproduction sur sa partie de population. Enfin la population obtenue est diffusé à l'ensemble des noeuds pour le calcul de la prochaine génération.

4.3 Résultats

Les algorithmes présentés dans la partie précédente ont été testé sur la plateforme *Myrinet* que nous possédons. Les mesures ont été effectuées pour deux taille de population : une faible, 16 individus, et une plus importante de 120 individus. Trois algorithmes ont été exécutés : la version séquentielle sur un seul noeud, la version parallélisée avec la bibliothèque standard `Externalizable` et la version parallèlisée avec la bibliothèque *Expresso*. Pour les deux versions parallèles, la bibliothèque de communication utilisée est *Mpi* au dessus de *GM* (la bibliothèque de bas niveau fournie par *Myrinet*).

La figure 11 contient la courbe d'accélération obtenue par les deux versions parallèles avec une population de 16 individus et avec une population de 128 individus.

Sur une faible population, où le temps de calcul est peu important, les phases de sérialisation et de désérialisation utilisées par la version de l'algorithme parallèle utilisant la classe `Externalizable` est prohibitive. Les performances se dégradent considérablement au dessus de 2 noeuds. Par contre la version de l'algorithme utilisant la bibliothèque *Expresso* reste proche de la courbe optimale.

Avec une population plus importante (la taille de la partie de la population sur une machine est de l'ordre de à $50\,Ko$) le temps de calcul représente 90% du temps total d'exécution. Pour un nombre de noeuds inférieur ou égal à 4 les deux versions sont performantes. A partir de 5 noeuds, le temps de sérialisation et désérialisation de la version `Externalizable` font chuter les performances. La version *Expresso* reste proche de la courbe idéale. Elle tend à s'en écarter à cause de la méthode de parallélisation choisie et non à cause de la bibliothèque *Expresso* : le nombre de calculs effectués par chaque noeud est proportionnel à la taille des arbres qu'il traite. Or ceux ci étant de taille variable d'un noeud à l'autre en fonction des opérations de croisement réalisées, le temps de calcul de chaque noeud entre deux phases de diffusion est déterminé par le noeud le plus lent. Plus les arbres sont de taille importante, plus les écarts de taille sont statistiquement élévés et moins la répartition de charge de travail entre les noeuds est uniforme.

En conclusion, la bibliothèque *Expresso* permet en l'état de bénéficier des performances d'un réseau haut-débit pour accélérer des applications parallèles tout en continuant à pro-

```
Class Population
  extends ClusterISO{ // herite de ClusterISO

  // decoupage de la population en NbNode
  Cell cell[][]= new Cell[nbNodes][nbCells];

  // ...

  public void diffusion(){

    // diffusion de la population privee a tous les autres noeuds
    this.setRootObject(this.cell[myNode]);
    for (int i=0; i<nbNodes; i++){
        this.send(i);
    }
    // reception de la population des autres noeuds
    for (int i=0; i<nbNodes-1; i++){
        ClusterISO oneCluster= ClusterISO.recv(ANY);
        this.cells[oneCluster.machine] = (Cell[])oneCluster.getRootObject();
    }
  }

  // ...

  }
```

FIG. 9 – Extrait de la définition de la classe `Population`

```
// chaque noeud est proprietaire d'une partie de la population
Population population0 = new Population();
Population population1 = new Population();
// initialisation de la population privee
population0.generate();
// iteration sur le nombre de generation
for (int i=0; i<n ; i++){
  // envoie de la population privee et
  // reception de la population des autres noeuds
  population0.diffusion();
  // bascule du cluster courant sur population1 et vidage
  population1.setCurrent();
  population1.empty();
  population1.crossCells(); // croisements
  population1.mutateCells(); // mutations
  // envoie de la population privee et
  // reception de la population des autres noeuds
  population1.diffusion();
  // bascule du cluster courant sur population0 et vidage
  population0.setCurrent();
  population0.empty();
  population0.reproduceCells(); // reproductions
}
// affichage (par un seul noeud) de la meilleure solution obtenue
if (myNode == 0){
  population0.printBest();
}
```

FIG. 10 – Programmation génétique parallélisée : algorithme de chaque noeud

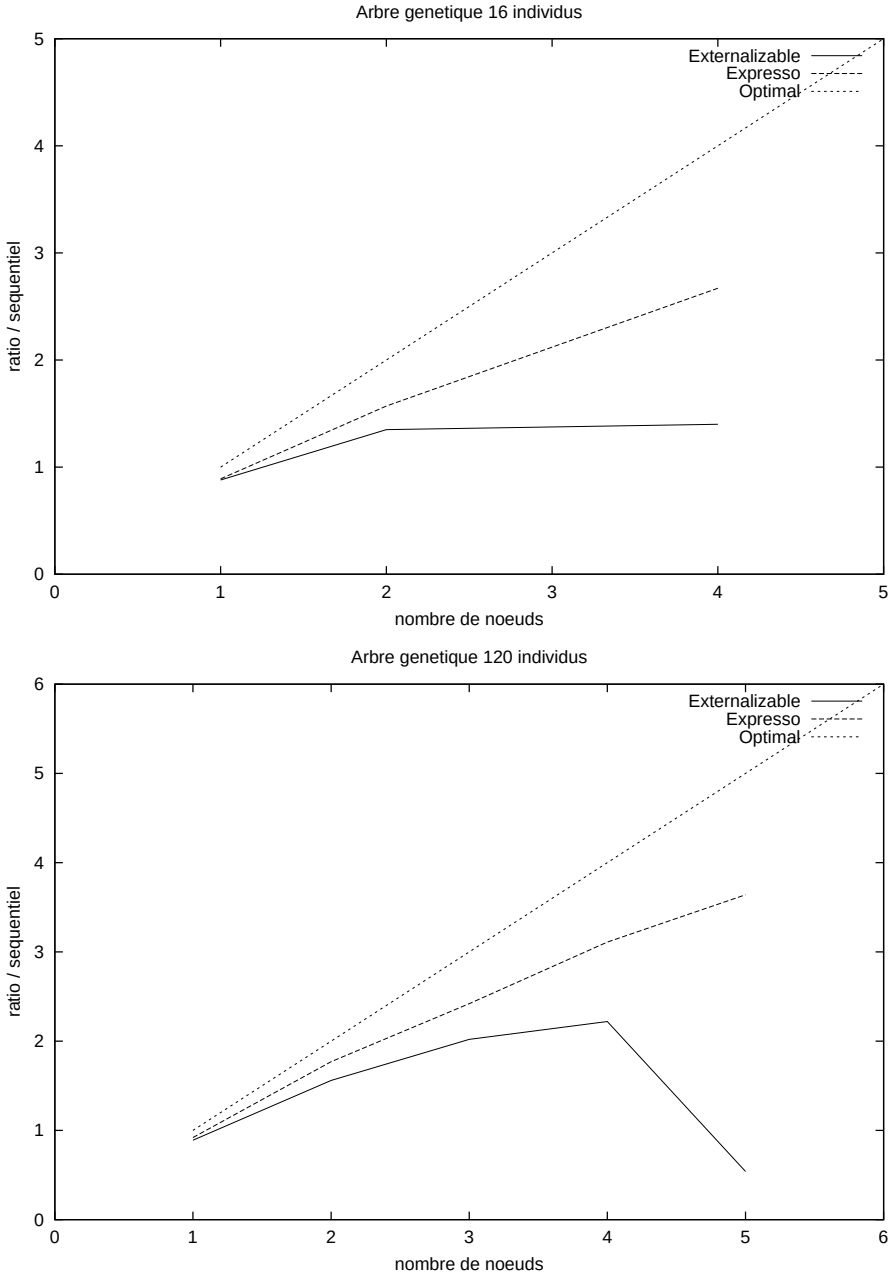

FIG. 11 – Courbe d'accélération de la programmation génétique parallélisée

grammer dans un langage de haut niveau.

5 Conclusion

La bibliothèque *Expresso* permet de disposer d'un moyen rapide de transfert d'un graphe d'objets sur un réseau haut débit de type Myrinet. Elle met en œuvre un espace iso-adresse découpé en clusters dans lesquels sont placés les objets Java à transporter. Ces clusters peuvent être transféré directement de mémoire à mémoire sans modification. Le principal avantage est l'absence des phases sérialisation-désérialisation qui se révélaient très coûteuses dans le contexte de l'utilisation d'un réseau haut débit. Une fois le transfert du cluster accompli, les objets contenus dans celui-ci peuvent être exploités normalement dans le code Java.

Les tests des primitives de la bibliothèques ainsi que les mesures effectuées sur une application de programmation génétique montrent que les performances obtenues sont excellentes.

Le prix à payer pour ces performances est modèle de communication de bas niveau mettant en jeu des variables contenant des objets plutôt que directement des objets. En effet, le principe du transfert iso-adresse ne permet notamment pas la définition de références inter-clusters sans risquer de rendre incohérentes ces références après nouvelle récpetion du même cluster. La bibliothèque *Expresso* ne vise pas un utilisateur final (bien que ce soit possible pour certaines applications comme l'application de programmation génétique présentée) mais plutôt un middleware offrant des services de plus haut niveau (communication d'objets, objets partagés, appel de méthode à distance, etc.).

Plusieurs pistes de modification et d'extension de la bibliothèque *Expresso* sont envisagées :

- L'intégration dans la machine virtuelle de la gestion de l'espace iso-adresse permettrait d'accélérer la création des objets dans les clusters qui nécessite actuellement un JNI relativement côuteux.
- Une possibilité de transférer des clusters relogés à une adresse différente à leur réception. Le côut de transfert est supérieur au transfert iso-adresse mais il supprime bon nombre d'inconvénient quant aux références inter-clusters. Les deux types de cluster (iso-adresse et relogeables) fourniraient un éventail de services efficaces assez large.

Références

[ABN99] G. Antoniu, L. Bougé, and R. Namyst. And efficient and transparent thread migration scheme in the pm2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP'99)*, San Juan, Puerto Rico, April 1999.

[Bar97] A. Bartoli. A novel approach to marshalling. *Software – Practice and Experience*, 27(1) :63–85, January 1997.

[BCF+95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network. *IEEE Micro*, February 1995. `http ://www.myri.com/`.

[Bri] A. Brindle. Genetic algorithms for function optimization. Unpublished doctoral dissertation, University of Alberta, Edmonton.

[Gol89] D. Goldberg. *Algorithms in Search, Optimization and machine Learning.* Addison-Wesley, 1989.

[Hol75] J. H. Holland. *Adaptation in Naturel and Artificial Systems.* PhD thesis, University of Michigan, 1975.

[MPI] *Message Passing Interface.* `http ://www.mpi-forum.org`.

[MvNV+99] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of java's remote method invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, Georgia, May 1999.

[PH99] M. Philippsen and B. Haumacher. More efficient object serialization. In *Proc. International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.

[Que97] C. Queinnec. Sérialisation–désérialisation en dmeroon. In Omar Rafiq, editor, *NOTERE'97*, pages 333–346, Pau, France, novembre 1997.