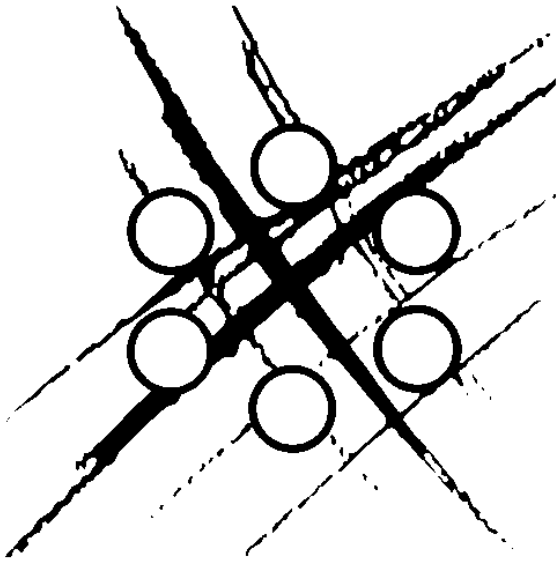
DIRECTION DE L'ÉNERGIE NUCLÉAIRE
DÉPARTEMENT MODÉLISATION DE SYSTÈMES ET STRUCTURES
SERVICE FLUIDES NUMERIQUES, MODELISATION ET ETUDES
LABORATOIRE DE GENIE LOGICIEL ET DE SIMULATION



# RAPPORT DM2S

**SFME/LGLS/RT/02-002/A**

**Introduction du parallélisme dans l'architecture logicielle du projet PAL/SALOME**

Bernard Sécher

COMMISSARIAT A L'ENERGIE ATOMIQUE

DIRECTION DES REACTEURS NUCLEAIRES
DEPARTEMENT DE MECANIQUE ET DE TECHNOLOGIE
SERVICE SIMULATION DES SYSTEMES COMPLEXES ET DE LOGICIELS

# RAPPORT DM2S

REFERENCES : SFME/LGLS/RT/02-002/A

**TITRE** : **Introduction du parallélisme dans l'architecture logicielle du projet PAL/SALOME**



| AUTEURS | SIGNATURES | AUTEURS | SIGNATURES |
|---|---|---|---|
| Bernard SECHER | | | |



**RESUME :**
Ce rapport a pour objet de présenter les premiers résultats concernant la prise en compte de codes parallèles dans l'architecture de la plate-forme SALOME.
L'état de l'art de ce qui se fait dans le domaine est présenté. On introduit ensuite la méthode proposée dans cette étude. Les principaux concepts sont définis, puis illustrés dans un cas d'utilisation basé sur la résolution d'un système linéaire à l'aide de la bibliothèque Numerical Platon.
La méthode idéale pour coupler des codes parallèles dans un environnement à architecture répartie consistera à utiliser un ORB parallèle lorsqu'il sera disponible.
Une approche simple consisterait à déléguer au processus maître l'échange des données parallèles. L'intégration d'un composant parallèle dans l'environnement SALOME serait aisé. Par contre, il y aurait un problème de performance non négligeable dans l'échange de données distribuées.
La méthode proposée dans ce document permet de conserver une bonne performance dans l'échange des données, mais une part importante de la gestion du parallélisme reste à la charge de l'utilisateur qui désire intégrer son code dans la plate-forme SALOME.
La phase suivante consistera à traiter un cas réaliste pour la DEN, de couplage de codes parallèles qui échangeront des maillages et des champs distribués sur différents processeurs.

**MOTS CLES** : Composant, Parallélisme, CORBA, *MPI, Numerical Platon*
**AFFAIRE** : EOTP: A – C – PARA – 01 - 02
**Titre de l'affaire** :

| | | | Visa | | | |
|---|---|---|---|---|---|---|
| A | 30/05/2002 | 30 | Nom<br>Date | M. Tajchman | Ch. Calvin | E. Dorlet |
| **Indice** | **Date** | **Nb.Page** | | **Vérificateur** | **Autre visa** | **Approbateur** |

# LISTE DE MODIFICATION

| Indice | Date | Motif et description de la modification |
|---|---|---|
| A | 30 mai 2002 | Document initial |
| | | |
| | | |
| | | |
| | | |

# 1 Introduction

La Direction de l'Energie Nucléaire (DEN), dans le cadre de sa mission électronucléaire, a lancé la construction d'un environnement de simulation basé sur une plate-forme d'accueil de composants logiciels [7], avec un modèle commun des données échangées : c'est le projet SALOME [1]. Les nouvelles applications doivent pouvoir simuler plus finement des phénomènes plus complexes en taille et en couplages physiques. Ces applications doivent aussi tirer le meilleur parti des performances des machines actuelles (calculateur massivement parallèle, cluster de PC, etc.). L'intégration des codes de calcul de la DEN, dont certains sont parallèles, dans l'environnement SALOME exige la possibilité d'introduire dans cette plate-forme la notion de composants parallèles.

Dans SALOME ce composant parallèle devra pouvoir interagir avec d'autres composants qu'ils soient séquentiels ou parallèles, co-localisés ou distants, tout en optimisant les communications et l'allocation mémoire. Si l'intégration de composants parallèles à mémoire partagée (utilisant par exemple OpenMP), ne pose pas de problème spécifique du fait de la localisation du parallélisme dans des régions bien circonscrites internes aux services du composant, il n'en va pas de même pour les composants parallèles à mémoire distribuée (utilisant par exemple MPI).

La couche logicielle de communication entre composants choisie dans SALOME est CORBA [2], [3]. Malheureusement, la norme CORBA actuelle (V2), ne prend pas en compte le parallélisme à mémoire distribuée. Il est donc nécessaire de réussir l'intégration de composants parallèles à mémoire distribuée dans SALOME en utilisant un logiciel de « middleware » CORBA (ORB) séquentiel.

On se propose de présenter d'abord l'état de l'art de ce qui se fait dans le domaine, puis d'introduire la méthode utilisée dans cette étude. Les principaux concepts sont définis dans un premier temps, puis on présente un cas d'utilisation basé sur la résolution d'un système linéaire à l'aide de la bibliothèque Numerical Platon [8].

# 2 Etat de l'art dans le domaine

L' « Object Management Group » (OMG) définit et spécifie actuellement la notion de CORBA parallèle [4,9]. Ainsi, dans le futur, la plate-forme SALOME bénéficiera de ces nouvelles fonctionnalités qui permettront de simplifier grandement l'introduction de composants parallèles qui ne devrait pas poser plus de problème que l'introduction de composants séquentiels, puisque la gestion du parallélisme (notamment l'échange de données distribuée) sera incluse dans le middleware et non plus dans le composant (pour plus de détails on peut se référer au document [9]). De plus, des efforts sont déjà entrepris pour fournir une première implémentation de CORBA parallèle : PaCO [5], ParDis [6]. Malheureusement, ces projets ne sont encore qu'à l'état de recherche et ne sont pas utilisables tels quels dans SALOME. Ils s'appuient sur un IDL étendu de CORBA, et donc imposent trop de contraintes sur leur utilisation. Enfin la version de PaCO testée au LGLS a présenté quelques problèmes d'utilisation (bug, fonctionnalité manquante) qui ont empêché d'effectuer un exercice complet dans le cadre de SALOME. Aujourd'hui, il n'existe aucune implémentation stable du CORBA parallèle qui reste encore à l'état de spécification ; d'où l'importance des travaux à réaliser dans ce domaine.

Le laboratoire de l'IRISA situé à Rennes travaille sur différents projets concernant l'utilisation de CORBA pour le calcul haute-performance. Tout d'abord il a entrepris des études sur une implémentation portable du concept d'objet CORBA parallèle (PaCO++) [11]. Dans ce dernier, il n'y a plus besoin de modifier la syntaxe du langage IDL. Les spécifications de distributions sont incluses dans un fichier XML. Un prototype est en cours de mise en oeuvre. Ce laboratoire a récemment utilisé cette implémentation pour expérimenter le réseau VTHD (1 Gb/s entre plusieurs grappes de l'INRIA). Il a réussi à obtenir un débit entre deux collections de l'ordre de 800 Mbit/s sur deux grappes distantes de 1000 km. Il a également lancé un nouveau projet sur la réalisation d'une plate-forme d'objets distribués haute-performance. Il s'agit du projet Padico [11]. L'objectif est de permettre d'utiliser de manière transparente des réseaux d'interconnexion (Myrinet, VIA, SCI, réseaux dans les machines parallèles) au travers de middleware (CORBA), exécutifs (MPI, DSM, ...) et de langages (Java). Avec ce type de plate-forme, une application qui utilise à la fois CORBA et MPI est assurée de pouvoir utiliser les meilleurs réseaux disponibles. Une première implémentation est en bonne voie d'achèvement avec des performances remarquables (CORBA: 240 Mo/s avec OmniORB et Myrinet-2000). Cette plate-forme est capable d'intégrer plusieurs implémentations de CORBA. L'intégration d'un ORB dans la plate-forme ne nécessite que quelques modifications de l'ORB. Enfin, le troisième projet concerne l'introduction du parallélisme au sein des modèles de composants (CCM, EJB). Il y a une thèse qui démarre à l'IRISA suite à un stage de DEA. Ils utilisent OpenCCM [10] (implémentation de CCM avec Java). Leur objectif est de pouvoir proposer une composition « scalable » permettant d'utiliser les réseaux rapides (via Padico).

D'autre part, « Mercury Computer Systems » et « MPI Software Technology » travaillent actuellement à une première implémentation des spécifications de l'OMG concernant un ORB parallèle. Une première version devrait être disponible en mars 2003.

D'autres laboratoires comme le CERFACS ou l'INRIA Rhône-Alpes proposent des solutions basées soit sur CORBA (pilote Athapascan-CORBA) soit uniquement sur MPI (PALM) [12].

# 3 Approche développée dans cette étude

L'introduction du parallélisme dans l'architecture logicielle du projet PAL/SALOME doit être directement liée à la définition exacte du grain des composants. En effet, si le couplage des codes reste au niveau du chaînage, il n'est pas indispensable (bien que cela soit plus performant) d'introduire dans la notion de composant parallèle la possibilité d' échanger des données distribuées. Par contre si l'on vise des couplages beaucoup plus fins, alors la nécessité de prendre en compte les échanges de données distribuées est à prendre en considération.

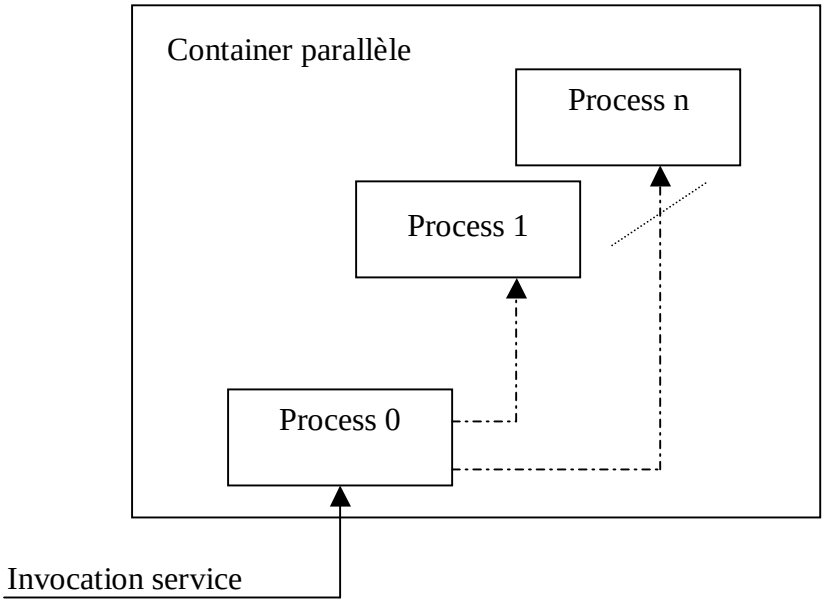
Ce document traite donc de la communication entre composants parallèles pouvant être distribués sur des nombres de processus différents. Ces composants vont échanger des « données distribuées » directement de processus à processus sans passer par un serveur unique qui deviendrait un goulot d'étranglement pour le transfert de ces données. Il y a donc redistribution des données. Cependant cela ne reste possible assez facilement que dans certains cas précis où les deux composants manipulent des données de même nature : discrétisation identique du problème traité. Nous n'étudierons dans ce document que cette situation.

Un autre point important est de pouvoir garder en mémoire les données générées par un service, à la fin de son exécution. Cela permet d'activer un autre service, local ou distant, qui utilise ces données sans avoir à les recharger en mémoire.

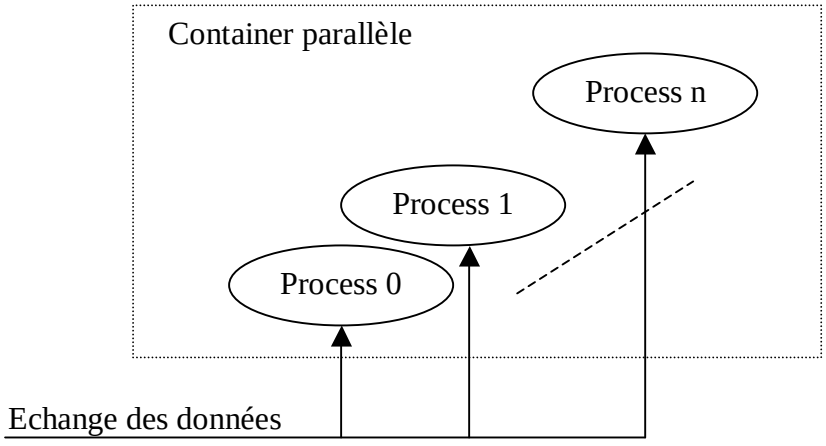
L'approche présentée dans ce document consiste à utiliser un ORB séquentiel pour coupler des composants parallèles. La gestion du parallélisme d'activation de service CORBA est donc entièrement à la charge du composant. Ainsi, seul le processus 0 du composant parallèle est visible depuis le client. Une fois activé par le client, le processus 0 activera, via CORBA, les autres processus constituant le composant parallèle. Du point de vue de CORBA, le processus 0 d'un composant parallèle se comporte en client de tous les autres processus. En ce qui concerne l'échange et la redistribution de données parallèles entre composants parallèles, chaque processus qui veut récupérer sa partie locale de la donnée, devient le client selon CORBA, de chacun des processus du composant parallèle qui abrite la donnée distribuée. Il y a ainsi communication directe de processus à processus, ce qui évite tout goulot d'étranglement pour l'échange de données. Cette méthode est la méthode classique utilisée par les personnes intéressées par le couplage de code parallèles dans une architecture répartie basée sur le « middleware » CORBA.

Les objectifs de cette étude sont illustrés par un cas d'utilisation qui est décris au chapitre 9.

Du point de vue du client, l'invocation d'un service d'un composant parallèle est identique à l'invocation d'un composant séquentiel. Seul le service sur le process 0 est activé par le client. C'est ce service qui active le service équivalent sur les autres process pour mettre en œuvre le parallélisme.




Les échanges des données parallèles restent à la charge du client de façon à ce que la redistribution se fasse de façon optimale directement du process serveur au process client qui héberge la donnée.



# 4 Définitions

Un composant SALOME est une « entité logicielle » de base qui offre un certain nombre de services. L'ensemble de ces services est défini dans le domaine public. Ils sont donc accessibles depuis n'importe quel autre logiciel client, qu'il soit localisé sur la même machine ou distant. Ils permettent également aux différents composants d'échanger entre eux des données. Dans SALOME, un composant est concrétisé par une ou plusieurs librairies dynamiques.

Le « middleware » est la couche logicielle chargée des communications entre les composants dans une architecture répartie (ex : CORBA) [2], [3]. Il définit notamment un service de nommage qui donne un accès à la référence de chacun des objets instanciés et identifiés.

Un « container » est un gestionnaire de composants sur une machine donnée (station de travail, serveur multi-processeurs, machine vectorielle, machine massivement parallèle comme le Compaq SC256 de Grenoble, cluster de PC/linux, …). C'est lui qui réalise l'instanciation et la destruction de composants sur la machine qui l'abrite. Dans SALOME, le container est concrétisé par un exécutable qui est un serveur CORBA.

Le principe de fonctionnement est le suivant : on suppose qu'un container est lancé sur une machine donnée. C'est un serveur CORBA dont la référence est enregistrée dans le service de nommage. Lorsqu'un client veut utiliser un service d'un composant donné sur cette machine, il demande au container d'instancier le composant. Le container charge en dynamique la ou les librairies correspondant au composant. Il lui affecte une référence CORBA et l'enregistre au service de nommage. Le client récupère auprès du service de nommage la référence du composant et peut activer le service voulu.

# 5 L'objet parallèle générique

Pour introduire dans SALOME le parallélisme, il s'agit de définir la notion de composant parallèle. Le cycle de vie (instanciation, arrêt, reprise, destruction, …) de ces composants parallèles sera géré par un container parallèle. Les composants parallèles fournissent à leurs clients des données, qui sont des objets parallèles. On vient de définir trois « entités parallèles » distinctes : les containers, les composants et les données. Pour leur réalisation, il nous faut définir un objet parallèle générique.

L'intérêt de la définition d'un objet parallèle générique est de spécifier les attributs et services communs aux trois entités parallèles précédemment définies. En fait, un objet parallèle est une collection d'objets. En effet le container parallèle MPI est une collection de processus qui exécutent le même programme. Donc chacun de ces processus est un serveur CORBA.

## *5.1 Définition de l'interface CORBA*

Un objet parallèle générique est une liste de références CORBA, chaque référence correspondant à l'identifiant de l'objet sur un processus donné. Au niveau du langage IDL de CORBA, on définit un type « IORTab » qui est une séquence d'objets génériques CORBA. L'objet parallèle générique possède un unique attribut « IORTab ». Ainsi le container parallèle hérite d'une part du container générique SALOME et d'autre part de l'objet générique parallèle.

```
#ifndef _SALOME_MPIOBJECT_IDL_
#define _SALOME_MPIOBJECT_IDL_

module Engines
{
  typedef sequence<Object> IORTab;
  interface MPIObject
  {
    attribute IORTab tior;
  } ;
} ;

#endif
```

### *5.2 Définition de l'implémentation*

L'implémentation de l'objet générique parallèle doit définir au moins deux fonctions : la première qui donne à l'objet parallèle sa liste de références CORBA (fonction set), et la deuxième qui permet à un client de récupérer la liste de références CORBA d'un objet parallèle quelconque (fonction get). A ceci , il est nécessaire de rajouter quelques attributs et services nécessaires au bon fonctionnement de l'objet parallèle générique : le nombre de processus, le numéro du processus courant, la liste des références CORBA de la collection d'objets formant l'objet parallèle, et une fonction d'échange des références CORBA entre processus, via MPI, afin de constituer la dite liste.

```
#ifndef _SALOME_MPIOBJECT_I_H_
#define _SALOME_MPIOBJECT_I_H_

#include "SALOMEconfig.h"
#include CORBA_SERVER_HEADER(MPIObject)

class MPIObject_i: public POA_Engines::MPIObject
{
 public:
  MPIObject_i();
  MPIObject_i(int nbproc, int numproc);
  ~MPIObject_i();

  Engines::IORTab* tior();
  void tior(const Engines::IORTab& ior);

 protected:
  // Number of current process
  int _numproc;
  // Process number
  int _nbproc;
  // List of IOR of each object on all mpi process
  Engines::IORTab* _tior;
  // Broadcast of IOR between each process to build IOR list
  void BCastIOR(CORBA::ORB_ptr orb,Engines::MPIObject_var pobj,bool
amiCont);
} ;

#endif
```

# 6 Le container parallèle

Le container parallèle hérite du container générique SALOME et de l'objet parallèle générique.

## *6.1 Définition de l'interface CORBA*

Il n'est pas nécessaire de spécifier de nouveaux attributs ou services. Ils existent déjà dans les classes parentes.

```
#ifndef _SALOME_MPICONTAINER_IDL_
#define _SALOME_MPICONTAINER_IDL_

#include "SALOME_Component.idl"
#include "MPIObject.idl"

module Engines
{
  interface MPIContainer:Container,MPIObject
  {
  } ;
} ;

#endif
```

### *6.2 Définition de l'implémentation*

Dans l'implémentation du container parallèle, il n'est pas nécessaire non plus, de rajouter de nouveaux attributs ou services. Par contre il est nécessaire de surcharger les différentes méthodes définies dans le container générique SALOME pour prendre en compte le parallélisme.

Ainsi dans le constructeur du container parallèle, seul l'objet du processus 0 s'enregistre au service de nommage. Les autres processus se contentent d'envoyer leur référence CORBA au processus 0 avec la fonction de l'objet parallèle générique : BCastIOR() pour constituer la liste IORTab. Ainsi, le container parallèle est vu par l'extérieur comme un container séquentiel. Un client qui active un service d'un container parallèle, active en fait uniquement le service correspondant au processus 0. C'est le service activé du processus 0, qui va ensuite activer les services correspondants des autres processus. Il pourra le faire via le mécanisme de CORBA, puisqu'il possède la liste des références CORBA des autres processus. Un service quelconque du processus 0 devient ainsi le client pour le même service des autres processus.

Ainsi, lorsque le container parallèle est activé pour instancier un composant parallèle, il va d'abord demander aux autres processus d'instancier le composant, puis instancier lui même ce composant. Ainsi une collection de composants identiques, identifiés par leur référence CORBA, est chargée en mémoire sur l'ensemble des processus constituant le container parallèle. Ces références CORBA sont échangées entre processus, via MPI, pour constituer la liste « IORTab » correspondant au composant parallèle ainsi formé. Seule la référence CORBA du composant chargé sur le processus 0 est enregistrée au service de nommage et renvoyée au client.

Il en va de même pour détruire un composant parallèle. Le client active le service de destruction d'un composant du container situé sur le processus 0. Ce service de destruction active alors le même service sur les autres processus, puisqu'il possède la liste des références de ces objets CORBA.

# 7 Le composant parallèle

Le composant parallèle hérite du composant générique SALOME et de l'objet parallèle générique.

L'interface CORBA d'un composant parallèle est bien sûr spécifique à la nature du composant. Cependant, on peut tirer certaines généralités liées à la nature du parallélisme. Ainsi, lorsque un service donné d'un composant parallèle est activé par un client, ce dernier connaît uniquement la référence CORBA du composant situé sur le processus 0.

Donc, seul le service correspondant au processus 0 s'exécute. Il faut que ce service active le même service du composant situé sur l'ensemble des autres processus. Il peut le faire, car il possède la liste des références CORBA de ces composants. De plus, pour que cet ensemble de services identiques, s'exécutent en parallèle, le processus 0 doit les activer en mode asynchrone : c'est à dire activer le service des autres processus et continuer sa propre exécution sans attendre que les services invoqués ne soit terminés.

Ainsi, un service d'un composant parallèle doit nécessairement être asynchrone (non bloquant). Or, cela n'est pas possible, si le service en question doit retourner au client une (ou plusieurs) donnée. Dans ce dernier cas, l'interface CORBA doit nécessairement définir deux services différents pour réaliser la même fonction. Le premier est synchrone (bloquant) et retourne au client la donnée parallèle. Il s'exécute seulement sur le processus 0 du serveur parallèle. Le second est asynchrone (non bloquant) : il s'exécute sur tous les autres processus et est activé uniquement par la fonction synchrone du processus 0. Si le service retourne une donnée au client, celle-ci est une donnée parallèle. C'est à dire que chaque processus crée sa propre donnée CORBA et envoie, via MPI, au processus 0 la référence CORBA de cette donnée. Une fois que le processus 0 , a reçu l'ensemble des références CORBA correspondant à la donnée parallèle (via la fonction BCastIOR()), il peut la renvoyer au client.

# 8 La donnée parallèle

La donnée parallèle hérite de l'objet parallèle générique. C'est une collection d'objets identiques distribués sur différents processus. Son interface CORBA est spécifique au type de la donnée, par exemple une séquence de doubles flottants pour un vecteur. Cependant, il est nécessaire de spécifier certains services qui permettront au client d'avoir des renseignements sur la distribution de l'objet parallèle situé sur le serveur. Notamment, il est important de pouvoir récupérer la taille totale de l'objet et sa distribution sur les différents processus contenant le composant parallèle qui abrite la donnée.

Le client qui veut récupérer une donnée parallèle, peut être lui même parallèle, mais avec un nombre de processus différents du serveur. Lors du transfert des données, il y aura donc redistribution de l'objet. Il faut que ce mécanisme soit optimal, c'est à dire que la donnée ne transite qu'une seule fois sur le réseau et que chaque processus client tire en parallèle les données qu'il a besoin en local et uniquement celles-ci. Il ne s'agit pas d'effectuer une redistribution sur les différents processus du client après le transfert de la donnée parallèle. Ceci est faisable puisque le client à accès à la taille totale de l'objet et sa distribution sur le serveur. Chaque processus du client sera donc capable de demander à chaque processus du serveur les données locales dont il a besoin et uniquement celles-ci.

On pourrait imaginer une autre façon de transférer les données : au lieu que ce soit le client qui sélectionne les valeurs dont il a besoin en fonction de leur distribution sur le serveur, on aurait pu faire que chaque processus client demande à chaque processus serveur les données locales dans la distribution cliente. Chaque processus serveur lui aurait renvoyé uniquement celles qui lui sont locales. Dans ce dernier cas, le client n'a pas besoin de demander au préalable au serveur de lui envoyer la distribution des données à transférer.

# 9 Cas d'utilisation

L'objectif de l'exercice est le suivant : il s'agit de lancer deux containers parallèles. Le premier container parallèle est activé sur n machines (PC/linux) différentes. Le second container est activé sur p autres machines (PC/linux) distinctes des machines abritant le premier container. On suppose, bien entendu, que le service de nommage CORBA est lancé préalablement. Un code client demande ensuite d'instancier sur le premier container un composant Numerical Platon [8] dont le but est de charger en mémoire une matrice depuis un fichier, un composant Numerical Platon dont le but est de charger en mémoire un vecteur depuis un fichier, un autre composant Numerical Platon dont le but est de récupérer un vecteur et de le stocker dans un fichier. Ces trois objets Numerical Platon sont donc distribués sur les n processus constituant le premier container. Le même code client demande ensuite au second container d'instancier un composant Numerical Platon dont le but est de résoudre un système linéaire. Ce composant a donc un unique service Solve() qui prend en entrée un matrice et un vecteur et redonne en sortie le vecteur solution. Ces trois objets sont des données CORBA parallèles. Une fois activé, le service Solve() travaille sur des objets Numerical Platon distribués sur p machines. La solution est ensuite renvoyée sur le premier container (et donc redistribuée sur n machines), puis stockée dans un fichier.

Les objets CORBA utilisés dans cet exercice sont donc des vecteurs : séquences de double flottants et une matrice creuse au format Compressed Sparse Row (CSR). Ces types de base sont définis dans un fichier d'include : « TypeData.idl ».

```
#ifndef _SALOME_TYPEDATA_IDL_
#define _SALOME_TYPEDATA_IDL_

module Engines
{

  typedef sequence<double> DoubleVec ;
  typedef sequence<long> IntVec;

  typedef struct CSR {
    unsigned long nbpos;
    unsigned long nbval;
    IntVec pos;
    IntVec col;
    DoubleVec data;
  } CSRMatStruct;

} ;

#endif
```

### *9.1 Le composant et la donnée vecteur parallèle*

Le composant parallèle vecteur Numerical Platon est censé être représentatif d'un code de calcul qui génère un résultat de type objet vecteur parallèle. Il faut bien faire la distinction entre le composant et la donnée, même si tous les deux sont représentés dans l'interface CORBA comme une « interface ». Si la donnée vecteur est une interface et non seulement une séquence de double flottants, c'est pour permettre d'une part de transmettre uniquement sa référence CORBA en argument de service et non pas l'ensemble du vecteur, et d'autre part de le définir en tant que donnée parallèle : c'est à dire que l'interface PVec hérite de l'interface générique MPIObject.

#### 9.1.1 Définition de l'interface CORBA

Dans l'exemple ci-dessous, on définit deux interfaces. Ces deux interfaces sont parallèles car elles héritent de l'interface générique parallèle : MPIObject.

La première interface nommée PVec représente la donnée vecteur parallèle. Elle contient un attribut de type séquence de double flottants. Cette séquence représente les données locales à un processeur, et constitue donc une partie seulement de l'objet vecteur distribué. Cette interface possède de plus une fonction Size() qui renvoie la taille totale du vecteur sur l'ensemble des processeurs, et une fonction LVec(start,end) qui renvoie les indices de début et de fin correspondant à la partie locale du vecteur sur chaque processeur « serveur » parmi l'ensemble du vecteur. Cette partie locale est constituée d'un ensemble contigu d'indices. Enfin une fonction nvec(start,end) permet à un processus « client » de récupérer une partie seulement des données locales au processeur « serveur ». Cette dernière fonction est importante, car elle permet à chaque processus d'un « client » parallèle, de récupérer sur les processus concernés du « serveur » parallèle, uniquement ses propres données locales. Les transferts d'une donnée parallèle est donc optimisée sur le réseau : chaque composante du vecteur parallèle ne transite qu'une seule fois sur le réseau et arrive directement sur le processus « client » qui l'héberge en local. Il n'y a pas de goulot d'étranglement logiciel car le transfert se fait en parallèle sur chacun des couples processus « client/serveur » concernés, et il n'y a pas redistribution des données, une fois que celles-ci sont arrivées sur les processus « client ». Le seul goulot d'étranglement est le débit du réseau utilisé.

La seconde interface nommée NPVecComponent représente le code de calcul qui génère ou qui récupère une donnée de type vecteur parallèle. Elle possède un attribut de type PVec : c'est la donnée parallèle échangée avec les autres composants, un service SetFileName(filename) qui permet de spécifier où se trouve la donnée persistante, un service ReadDataFromFile() qui permet de charger la donnée en mémoire et un service SaveDataToFile() qui permet de sauvegarder une donnée en mémoire dans un fichier. Ces deux derniers services doivent pouvoir s'exécuter en parallèle : ils sont donc asynchrones (type CORBA oneway). De plus l'attribut PVec va générer deux fonctions de lecture et écriture synchrones. Pour pouvoir exécuter ces services en parallèles, il est nécessaire de définir deux autres services identiques mais asynchrones : get_dvec() et put_dvec().

```
#ifndef _SALOME_NPVECCOMPONENT_IDL_
#define _SALOME_NPVECCOMPONENT_IDL_

#include "SALOME_Component.idl"
#include "TypeData.idl"
#include "MPIObject.idl"

module Engines
{
  // Definition de la donnee vecteur parallele
  interface PVec : MPIObject
  {
    readonly attribute DoubleVec vec;

    unsigned long Size();
    void LVec(out unsigned long start, out unsigned long end);
    DoubleVec nvec(in unsigned long start, in unsigned long end);
  };

  // Definition du composant vecteur NP parallele
  interface NPVecComponent:Component,MPIObject
  {
    // version synchrone des lecture/écriture du vecteur parallele
    attribute PVec dvec;
    // version asynchrone des lecture/écriture du vecteur parallele
    oneway void get_dvec(in string id_callback);
    oneway void put_dvec(in PVec vec,in string id_callback);

    void SetFileName(in string filename);

    oneway void ReadDataFromFile(in string id_callback);
    oneway void SaveDataToFile(in string id_callback);

  } ;
} ;

#endif
```

### 9.1.2 Définition de l'implémentation

L'implémentation de la donnée vecteur parallèle doit donc définir une fonction Size() qui renvoie la taille totale du vecteur distribué, une fonction LVec(start,end) qui renvoie la distribution locale à un processeur du vecteur parallèle, une fonction vec() qui renvoie les données locales du vecteur à un processeur. De plus, pour permettre un transfert optimal des données parallèles d'un composant à un autre lorsque ceux-ci se trouvent dans des containers différents et suivant un nombre de processus différents, il est nécessaire d'implémenter une fonction nvec(start,end) qui renvoie une partie seulement des données locales du vecteur à un processeur.

L'implémentation du composant vecteur Numerical Platon est représentative d'un code de calcul qui génère un objet de type vecteur parallèle, ou qui prend en entrée d'un algorithme un vecteur parallèle. Il doit donc définir une fonction de lecture d'un vecteur depuis un fichier : ReadDataFromFile(), d'écriture d'un vecteur dans un fichier : SaveDataToFile(), une fonction qui renvoie une référence CORBA sur un objet vecteur parallèle dvec(), une fonction qui prend en entrée une référence CORBA sur un objet vecteur parallèle dvec(vec), ainsi que leur équivalent en asynchrone : get_dvec() et put_dvec().

```
#ifndef _NPVECCOMPONENT_
#define _NPVECCOMPONENT_

#include "SALOMEconfig.h"
#include CORBA_SERVER_HEADER(NPVecComponent)
#include "SALOME_Component_i.hxx"
#include "MPIObject_i.h"
#include "np_vector.hh"

class PVec_i: public POA_Engines::PVec,
            public MPIObject_i

{
public:
  // Constructors
  PVec_i(int nbproc, int numproc, int *lim, int size, double *data) ;
  // Destructor
  ~PVec_i() ;

  Engines::DoubleVec* vec();
  CORBA::ULong Size() { return (CORBA::ULong)_size; };
  void LVec(CORBA::ULong& start, CORBA::ULong& end) { start = _lim[0];
                                                      end = _lim[1]; };
  Engines::DoubleVec* nvec(CORBA::ULong start, CORBA::ULong end);

 protected:
  Engines::DoubleVec* _vec;
  int _lim[2];
  int _size;
  double *_data;

} ;
```

```
class NPVecComponent_i: public POA_Engines::NPVecComponent,
                  public Engines_Component_i,
                  public MPIObject_i

{
public:
  // Constructors
  NPVecComponent_i() ;
  NPVecComponent_i( int nbproc, int numproc,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName);
  NPVecComponent_i( int nbproc, int numproc,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName,
                int flag);
  NPVecComponent_i( int nbproc, int numproc, NPvector *vec,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName);
  // Destructor
  ~NPVecComponent_i() ;

  void SetFileName(const char*fileName);
  void ReadDataFromFile(const char* id_callback) ;

  void SaveDataToFile(const char* id_callback) ;

  NPvector *GetData( void );
  Engines::PVec_ptr dvec();
  void dvec(Engines::PVec_ptr vec);
  void get_dvec(const char* id_callback);
  void put_dvec(Engines::PVec_ptr vec,const char* id_callback);

protected:
  // filename
  string _fileName;
  // NP vector pointer
  NPvector *_vec;
  // Get Local Data
  Engines::PVec_ptr GetLocalData(void);
} ;

#endif
```

### *9.2 Le composant et la donnée matrice parallèle*

Le composant parallèle matrice Numerical Platon est censé être représentatif d'un code de calcul qui génère un résultat de type objet matrice parallèle. Il faut bien faire la distinction entre le composant et la donnée, même si tous les deux sont représentés dans l'interface CORBA comme une « interface ». Si la donnée matrice est une interface et non seulement une structure CSR, c'est pour permettre d'une part de transmettre uniquement sa référence CORBA en argument de service et non pas l'ensemble de la matrice, et d'autre part de le définir en tant que donnée parallèle : c'est à dire que l'interface PMat hérite de l'interface générique MPIObject.

## 9.2.1 Définition de l'interface CORBA

Dans l'exemple ci-dessous, on définit deux interfaces. Ces deux interfaces sont parallèles car elles héritent de l'interface générique parallèle : MPIObject.

La première interface nommée PMat représente la donnée matrice parallèle. Elle contient un attribut de type structure CSR définie dans le fichier « TypeData.idl ». Cette structure représente les données locales à un processeur, et constitue donc une partie seulement de l'objet matrice distribué. Cette interface possède de plus une fonction NbRows() qui renvoie le nombre total de lignes de la matrice sur l'ensemble des processeurs, une fonction NbCols() qui renvoie le nombre total de colonnes de la matrice sur l'ensemble des processeurs et une fonction LMat(start,end) qui renvoie les indices de ligne de début et de fin correspondant à la partie locale de la matrice sur chaque processeur « serveur » parmi l'ensemble de la matrice (on suppose dans cet exemple que la matrice est distribuée sur les différents processeurs suivant des blocs de lignes entières). Enfin une fonction ncsrmat(start,end) permet à un processus « client » de récupérer une partie seulement des données locales au processeur « serveur ». Cette dernière fonction est importante, car elle permet à chaque processus d'un « client » parallèle, de récupérer sur les processus concernés du « serveur » parallèle, uniquement ses propres données locales. Les transferts d'une donnée parallèle est donc optimisée sur le réseau : chaque composante de la matrice parallèle ne transite qu'une seule fois sur le réseau et arrive directement sur le processus « client » qui l'héberge en local. Il n'y a pas de goulot d'étranglement logiciel car le transfert se fait en parallèle sur chacun des couples processus « client/serveur » concernés, et il n'y a pas redistribution des données, une fois que celles-ci sont arrivées sur les processus « client ». Le seul goulot d'étranglement est le débit du réseau utilisé.

La seconde interface nommée NPMatComponent représente le code de calcul qui génère une donnée de type matrice parallèle. Elle possède un attribut de type PMat : c'est la donnée parallèle échangée avec les autres composants, un service SetFileName(filename) qui permet de spécifier où se trouve la donnée persistante, un service ReadDataFromFile() qui permet de charger la donnée en mémoire et un service SaveDataToFile() qui permet de sauvegarder une donnée en mémoire dans un fichier. . Ces deux derniers services doivent pouvoir s'exécuter en parallèle : ils sont donc asynchrones (type CORBA oneway). De plus l'attribut « readonly » PCSRMat va générer une fonction de lecture synchrone. Pour pouvoir l'exécuter en parallèle, il est nécessaire de définir un autre service identique mais asynchrone : get_dmat.

```
#ifndef _SALOME_NPMATCOMPONENT_IDL_
#define _SALOME_NPMATCOMPONENT_IDL_

#include "SALOME_Component.idl"
#include "TypeData.idl"
#include "MPIObject.idl"

module Engines
{
  // Definition de la donnee matrice parallele
  interface PCSRMat : MPIObject
  {
    readonly attribute CSRMatStruct csrmat;

    unsigned long NbRows();
    unsigned long NbCols();
    void LMat(out unsigned long start, out unsigned long end);
    CSRMatStruct ncsrmat(in unsigned long start, in unsigned long end);
  };

  // Definition du composant matrice NP parallele
  interface NPMatComponent:Component,MPIObject
  {

    // lecture de la donnee matrice parallele
    readonly attribute PCSRMat dmat;
    // version asynchrone de lecture de la matrice parallele
    oneway void get_dmat(in string id_callback);

    void SetFileName(in string filename);

    oneway void ReadDataFromFile(in string id_callback);
    oneway void SaveDataToFile(in string id_callback);

  } ;
} ;

#endif
```

### 9.2.2 Définition de l'implémentation

L'implémentation de la donnée matrice parallèle doit donc définir une fonction NbRows() qui renvoie le nombre total de lignes de la matrice distribuée, une fonction NbCols() qui renvoie le nombre total de colonnes de la matrice distribuée, une fonction LMat(start,end) qui renvoie la distribution locale à un processeur de la matrice parallèle, une fonction csrmat() qui renvoie les données locales de la matrice à un processeur. De plus, pour permettre un transfert optimal des données parallèles d'un composant à un autre lorsque ceux-ci se trouvent dans des containers différents et suivant un nombre de processus différents, il est nécessaire d'implémenter une fonction ncsrmat(start,end) qui renvoie une partie seulement des données locales de la matrice à un processeur.

L'implémentation du composant matrice Numerical Platon est représentative d'un code de calcul qui génère un objet de type matrice parallèle. Il doit donc définir une fonction de lecture d'une matrice depuis un fichier : ReadDataFromFile(), d'écriture d'une matrice dans un fichier : SaveDataToFile(), une fonction qui renvoie une référence CORBA sur un objet matrice parallèle dmat(), ainsi que son équivalent en asynchrone : get_dmat().

```
#ifndef _NPMATCOMPONENT_
#define _NPMATCOMPONENT_

#include "SALOMEconfig.h"
#include CORBA_SERVER_HEADER(NPMatComponent)
#include "SALOME_Component_i.hxx"
#include "MPIObject_i.h"
#include "np_matrix.hh"

class PCSRMat_i: public POA_Engines::PCSRMat,
              public MPIObject_i

{
public:
  // Constructors
  PCSRMat_i(int nbproc, int numproc, int *lim, int nbrows, int nbcols,
NP_csr *csr) ;
  // Destructor
  ~PCSRMat_i() ;

  Engines::CSRMatStruct* csrmat();
  CORBA::ULong NbRows() { return (CORBA::ULong)_nbrows; };
  CORBA::ULong NbCols() { return (CORBA::ULong)_nbcols; };
  void LMat(CORBA::ULong& start, CORBA::ULong& end) { start = _lim[0];
                                                      end = _lim[1]; };
  Engines::CSRMatStruct* ncsrmat(CORBA::ULong start, CORBA::ULong end);

 protected:
  Engines::CSRMatStruct_var _csrmat;
  int _lim[2];
  int _nbrows;
  int _nbcols;
  NP_csr *_csr;
```

```
} ;

class NPMatComponent_i: public POA_Engines::NPMatComponent,
                  public Engines_Component_i,
                  public MPIObject_i

{
public:
  // Constructors
  NPMatComponent_i() ;
  NPMatComponent_i( int nbproc, int numproc,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName);
  NPMatComponent_i( int nbproc, int numproc,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName,
                int flag);
  NPMatComponent_i( int nbproc, int numproc, NPmatrix *mat,
                CORBA::ORB_ptr orb,
                PortableServer::POA_ptr poa,
                PortableServer::ObjectId * contId,
                const char *instanceName,
                const char *interfaceName);
  // Destructor
  ~NPMatComponent_i() ;

  void SetFileName(const char*fileName);
  void ReadDataFromFile(const char* id_callback) ;

  void SaveDataToFile(const char* id_callback) ;

  NPmatrix *GetData( void );
  Engines::PCSRMat_ptr dmat();
  void get_dmat(const char* id_callback);

protected:
  // filename
  string _fileName;
  // NP matrix pointer
  NPmatrix *_mat;
  // Get Local Data
  Engines::PCSRMat_ptr GetLocalData(void);
} ;

#endif
```

### *9.3 Le composant solveur et la donnée vecteur résultat parallèle*

Le composant parallèle solveur Numerical Platon est censé être représentatif d'un code de calcul qui résoud un système linéaire en parallèle.

## 9.3.1 Définition de l'interface CORBA

Ce composant possède un service Solve() qui prend en entrée une donnée matrice parallèle et une donnée vecteur parallèle. Il donne en sortie une donnée vecteur parallèle. Il ne peut donc pas être asynchrone. Pour que la résolution du système linéaire s'effectue en parallèle, il est nécessaire de définir un second service asynchrone de type « oneway » : c'est le service SPSolve(). Le client qui veut invoquer le service de résolution du système linéaire fera appel au service Solve() sur le processus 0 du serveur qui abrite le composant solveur. Puis, le service Solve() activera le service SPSolve() sur tous les autres processus. De cette manière la résolution du système sera bien parallélisée.

```
#ifndef _SALOME_NPSOLVECOMPONENT_IDL_
#define _SALOME_NPSOLVECOMPONENT_IDL_

#include "SALOME_Component.idl"
#include "TypeData.idl"
#include "MPIObject.idl"
#include "NPVecComponent.idl"
#include "NPMatComponent.idl"

module Engines
{
  interface NPSolveComponent:Component,MPIObject
  {
    void Solve( in PCSRMat A, in PVec b, out PVec x );
    oneway void SPSolve( in PCSRMat A, in PVec b, in string id_callback );
  } ;
} ;

#endif
```

### 9.3.2 Définition de l'implémentation

L'implémentation du composant solveur parallèle doit donc définir les méthodes Solve() et SPSolve(). Comme elles sont quasiment identiques, la résolution du système linéaire à été factorisée dans la méthode LSolve().

```
#ifndef __NPSOLVECOMPONENT__
#define __NPSOLVECOMPONENT__

#include "SALOMEconfig.h"
#include CORBA_SERVER_HEADER(NPSolveComponent)
#include "MPIObject_i.h"
#include "SALOME_Component_i.hxx"
#include "np_vector.hh"
#include "np_matrix.hh"
#include "np_precond.hh"
#include "np_solver.hh"

class NPSolveComponent_i : public POA_Engines::NPSolveComponent,
                    public Engines_Component_i,
                    public MPIObject_i
{

 public:
  // Constructor
  NPSolveComponent_i();
  NPSolveComponent_i( int nbproc, int numproc,
                 CORBA::ORB_ptr orb,
                 PortableServer::POA_ptr poa,
                 PortableServer::ObjectId * contId,
                 const char *instanceName,
                 const char *interfaceName);
  NPSolveComponent_i( int nbproc, int numproc,
                 CORBA::ORB_ptr orb,
                 PortableServer::POA_ptr poa,
                 PortableServer::ObjectId * contId,
                 const char *instanceName,
                 const char *interfaceName,
                 int flag);
  // Destructor
  ~NPSolveComponent_i();

  // Solve service
  void Solve( Engines::PCSRMat_ptr A, Engines::PVec_ptr b,
            Engines::PVec_out x );
  void SPSolve( Engines::PCSRMat_ptr A, Engines::PVec_ptr b,
            const char* id_callback );

  // get NP vector Pointer
  NPvector *Vec(void) { return(_x); };

 private:
  NPmatrix *_A;
```

```
  NPvector *_b;
  NPvector *_x;
  int _limx[2];
  // Conversion des entrees de la structure CORBA
  // a la structure d'implementation
  NPmatrix *ReadNPMatrix(Engines::PCSRMat_ptr mat);
  NPvector *ReadNPVector(Engines::PVec_ptr vec);
  // Conversion de la sortie de la structure d'implementation
  // a la structure CORBA
  PVec_i *WriteOutput(void);
  // Affichage du resultat de la resolution du systeme
  void PrintResult( NPsolver *sol );
  // Resolution du systeme
  Engines::PVec_ptr LSolve(Engines::PCSRMat_ptr A, Engines::PVec_ptr b);

};
#endif
```

### *9.4 Cas test*

On initialise l'environnement SALOME avec la commande runSession. Puis les deux container parallèles sont lancés sur 2 et 3 machines de type PC/linux respectivement. Ensuite on active l'Interface Homme/Machine de SALOME avec la commande runLoader. Depuis la fenêtre de l'interpréteur de commandes python (TUI), on demande au premier container d'instancier un composant matrice et deux composants vecteur, puis au second container d'instancier un composant solveur. On récupère ainsi la référence CORBA de chacun des composants parallèles. On demande au composant matrice de charger en mémoire une matrice distribuée depuis un fichier, au premier composant vecteur de charger en mémoire un vecteur distribué depuis un fichier. On récupère ainsi la référence CORBA de ces deux objets parallèles. On peut alors activer le service Solve() du composant solveur situé sur le second container parallèle en lui fournissant les références CORBA des deux objets parallèles matrice A et vecteur second membre b. Les données transitent alors via le réseau de manière optimale tout en se redistribuant depuis deux processeurs vers trois processeurs. Le système est résolu en parallèle. On récupère la référence CORBA de l'objet vecteur solution parallèle. Cette référence CORBA est fournie au second composant vecteur du premier container. La donnée vecteur parallèle est donc transférée via le réseau de manière optimale, tout en étant redistribuée depuis trois processeurs vers deux processeurs. On demande enfin au composant vecteur de stocker la solution dans un fichier.

```
import salome
import SALOMEDS

A=salome.lcc.FindOrLoadComponent("data","NPMatComponent")
A.SetFileName("/home/secher/data/Matrice100.xdr")
A.ReadDataFromFile("callback")

b=salome.lcc.FindOrLoadComponent("data","NPVecComponent")
b.SetFileName("/home/secher/data/ScdMembre100.xdr")
b.ReadDataFromFile("callback")

x=salome.lcc.FindOrLoadComponent("data","NPVecComponent")
x.SetFileName("/home/secher/data/Solution100.xdr")

s=salome.lcc.FindOrLoadComponent("solve_system","NPSolveComponent")
xv=s.Solve(A._get_dmat(),b._get_dvec())

x._set_dvec(xv)
x.SaveDataToFile("callback")
```

# 10 Conclusion

La méthode idéale pour coupler des codes parallèles dans un environnement à architecture répartie consistera à utiliser un ORB parallèle. On aura ainsi à la fois la simplicité d'intégration des composants dans la plate-forme SALOME et une bonne performance dans le traitement des algorithmes et les échanges de données. Malheureusement, comme nous l'avons dit au début de ce document, l'ORB parallèle n'en est qu'à l'état de recherche, même si l'OMG a décidé son introduction dans la norme CORBA, et défini un certain nombre de spécifications [9].

Une approche simple consisterait à déléguer au processus maître l'échange des données parallèles. L'intégration d'un composant parallèle dans l'environnement SALOME serait aisé. Par contre, il y aurait un problème de performance non négligeable dans l'échange de données distribuées.

La méthode proposée dans ce document pour introduire la notion de composant parallèle dans la plate-forme SALOME est suffisamment générale pour permettre à deux codes de calculs parallèles chargés sur deux machines différentes, d'échanger des données parallèles tout en optimisant leur transfert sur le réseau, à condition que la discrétisation du problème traité soit identique dans ces deux programmes. L'inconvénient de la méthode expliquée dans ce document pour introduire la notion de composant parallèle dans SALOME à l'aide d'un ORB séquentiel, est que toute la mécanique de gestion du parallélisme est incluse dans le composant parallèle. Elle est donc entièrement à la charge de l'utilisateur qui désire intégrer son code dans la plate-forme SALOME, même si certaines fonctionnalités sont intégrées dans l'objet parallèle générique. Enfin le traitement des exceptions n'a pas été du tout abordé dans cette étude, et demande à être regardé (notamment avec l'utilisation de procédures asynchrones de type oneway pour mettre en œuvre le parallélisme).

La phase suivante consistera à traiter un cas réaliste pour la DEN, de couplage de codes parallèles qui échangeront des maillages et des champs distribués sur différents processeurs.

# 11 Références

[1] http://www.opencascade.org/SALOME/Docs/Index2.html
[2] http://www.corba.org
[3] « CORBA Des concepts à la pratique » - JM. Geib, Ch. Gransart et Ph. Merle, DUNOD – 1999
[4] http://www.omg.org/techprocess/meetings/schedule/Parallel_Processing_RFP.html
[5] http://www.irisa.fr/paris/nanglais/paco.htm
[6] http://www.cs.indiana.edu/hyplan/kksiazek/pardis.html
[7] http://www.cetus-links.org
[8] «Numerical Platon users guide and reference manual» - L. Colombet et B. Sécher CEA/DEN/DM2S/SFME/LGLS/RT/01-001 du 9 mars 2001
[9] http://cgi.omg.org/cgi-bin/doc?orbos/00-08-11
[10] http://lifc.univ-fcomte.fr/~philippe/composants/papiers/JC2001_article_Perez.ps
[11] http://www.irisa.fr/paris/nanglais/padico.htm
[12] Compte rendu de réunion du 20 mars 2002: CEA/DEN/CAD/DTP/SMTH/LDTA/DO/77 du 6 mai 2002