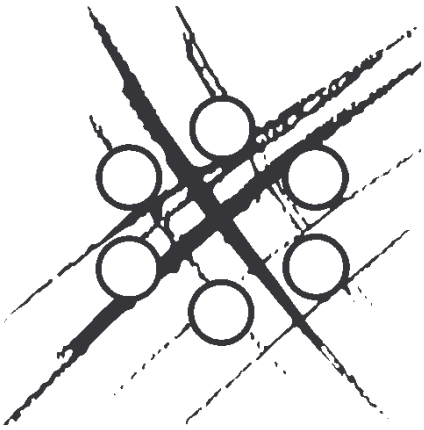
DIRECTION DE L'ÉNERGIE NUCLEAIRE
DIRECTION DELEGUEE AUX ACTIVITES NUCLEAIRES DE SACLAY
DEPARTEMENT DE MODELISATION DES SYSTEMES ET STRUCTURES
SERVICE FLUIDES NUMERIQUES, MODELISATION ET ETUDES



Le couplage de codes parallèles dans la plateforme Salomé

DEN/DANS/DM2S/SFME/LGLS/RT/10-004/A

Bernard Sécher

Commissariat à l'Energie Atomique et aux énergies alternatives
DEN/DANS/DM2S/SFME/LGLS
Centre de Saclay BAT 454 - PC 47
91191 Gif/Yvette cedex - France
Tél. : 01 69 08 91 10 Fax : 01 69 08 96 96 Courriel : evelyne.macanda@cea.fr

Établissement public à caractère industriel et commercial
RCS Paris B 775 685 019

Le couplage de codes parallèles dans la plateforme Salomé

NIVEAU DE CONFIDENTIALITE				
DO	DR	CCEA	CD	SD
X				

PARTENAIRES/CLIENTS	REFERENCE DE L'ACCORD OU DU CONTRAT	TYPE D'ACTION
EDF	NEPAL	

REFERENCES INTERNES CEA			
DIRECTION D'OBJECTIFS	PROGRAMME	PROJET	EOTP
DISN	SIMULATION	PICI2	A-PICI2-06-01-01

S'IL S'AGIT DU LIVRABLE D'UN JALON :

JALON	INTITULE DU JALON
	SO

S'agit-il d'une synthèse ? ☐ oui ☒non

SUIVI DES VERSIONS			
INDICE	DATE	NATURE DE L'EVOLUTION	PAGES, CHAPITRES
A	22/03/2010	Document initial	

	NOM	FONCTION	VISA	DATE
REDACTEUR	Bernard Sécher			22/03/10
VERIFICATEUR	Anthony Geay			
VERIFICATEUR	Nicolas Crouzet			
APPROBATEUR	Vincent Bergeaud	Chef de laboratoire		
AUTRE VISA				
ÉMETTEUR	Daniel Caruge	CHEF DE SERVICE		

Le couplage de codes parallèles dans la plateforme Salomé

MOTS CLEFS

Salomé – Composant - Couplage - Codes parallèles – Corba – MPI2

RESUME / CONCLUSIONS

L'objectif de ce document est de spécifier le mode de fonctionnement et l'interface de composants parallèles dans Salomé en vue d'un couplage de codes qui échangent des champs éventuellement sur des maillages distincts. Cette action s'inscrit dans le cadre de différents travaux d'ores et déjà entrepris par le CEA et ses partenaires (projets PAL et EHPOC) pour constituer une chaîne de calcul parallèle compatible avec le traitement de grands volumes de données. Ces composants parallèles Salomé sont hébergés dans des containers MPI déjà disponibles dans Salomé. Dans le cadre de ce travail on met en place un mécanisme de couplage pour des composants dont l'interface peut être conforme à l'API ICOCO définie pour les besoins du projet Neptune. Les échanges de champs distribués entre les codes peuvent nécessiter des interpolations qui sont réalisées par des algorithmes implémentés dans le package INTERP_KERNEL intégré dans le module MED de Salomé. Les échanges des données sont réalisés via MPI, par la bibliothèque ParaMEDMEM du module MED de Salomé. Le mécanisme est ensuite validé sur un cas de couplage de codes parallèles réels à l'aide du code Triou.

La construction du composant parallèle Salomé n'est pas simple et demande une certaine compétence technique qu'il n'est pas possible de demander aux utilisateurs d'acquérir. Il est donc indispensable de mettre en place un outil automatique chargé de générer ce composant à partir de son API purement C++. Cet outil existe déjà, il s'agit de HXX2SALOME. Il est par contre indispensable de le faire évoluer afin de tenir compte du parallélisme. Une fois cet outil étendu, chaque utilisateur de la plateforme Salomé bénéficiera d'un outil performant capable de générer automatiquement un composant parallèle Salomé à partir de son implémentation purement C++.

Diffusion initiale interne CEA

LGLS
D. Caruge + chefs labo SFME

G. Fauchet	CEA/Grenoble
F. Perdu	CEA/Grenoble
Ph. Emonot	CEA/Grenoble
C. Chaigneau	
Ph. Fillion	DM2S/SFME/LETR
Ch. Calvin	DM2S/SERMA
S. Salmons	DM2S/SERMA
N. Bouhamou	DEC/SESC
B. Michel	DEC/SESC

Diffusion initiale externe CEA

Ch. Caremoli	EDF/Sinetics
P. Rascle	EDF/Sinetics
A. Ribes	EDF/Sinetics
E. Fayolle	EDF/Sinetics
V. Lefebvre	EDF/Sinetics

SOMMAIRE

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, 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 multi-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 [2,3].

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. Malheureusement, la norme Corba actuelle (V2), ne prend pas en compte le parallélisme à mémoire distribuée. Quelques rares tentatives ont été faites pour réaliser des ORB parallèles. Ainsi le produit PACO++ [4] développé au laboratoire de l'IRISA à Rennes, et à EDF à Clamart. Au CEA nous avons opté pour une seconde voie, basée sur l'intégration de composants parallèles à mémoire distribuée dans SALOME en utilisant un intergiciel Corba (ORB) séquentiel [2].

On se propose de présenter d'abord la méthode utilisée dans cette étude. Les principaux concepts sont définis dans un premier temps, puis on présente un cas d'utilisation basé sur le couplage de codes parallèles TRIOU-TRIOU.

Dans le cadre de ce travail on met en place un mécanisme de couplage pour des composants dont l'interface peut être conforme à l'API ICoCo définie pour les besoins du projet Neptune [5]. Les échanges de champs distribués entre les codes peuvent nécessiter des interpolations qui sont réalisées par des algorithmes implémentés dans le package INTERP_KERNEL intégré dans le module MED de Salomé. Les échanges des données sont réalisés via MPI, par la bibliothèque ParaMEDMEM [6,7] du module MED de Salomé.

2 Définitions

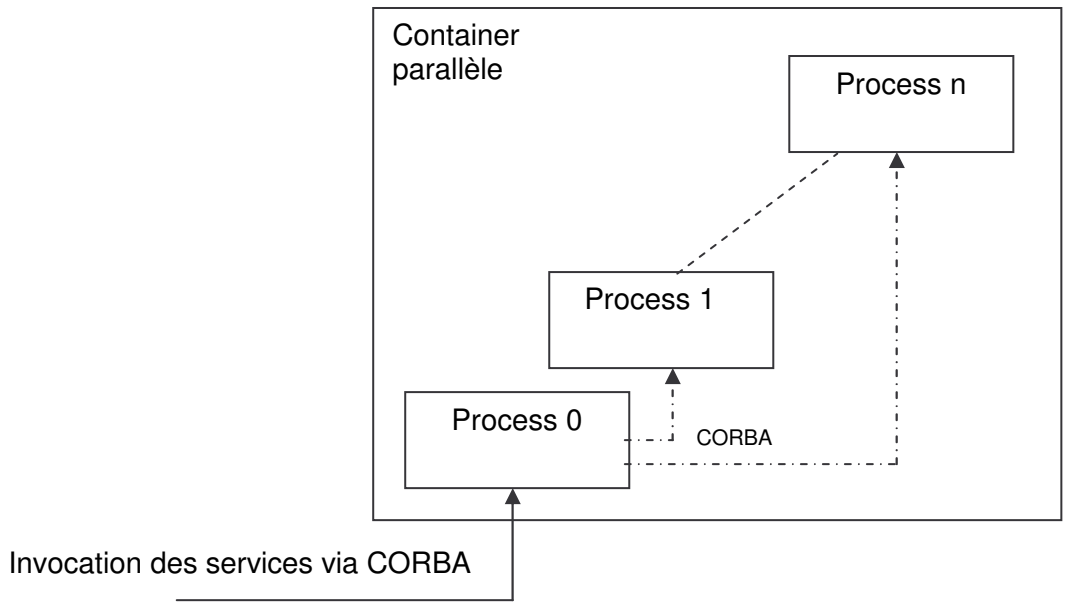
Ce document traite de la communication entre composants parallèles pouvant être distribués sur des nombres de processus différents. Ces composants vont échanger des « données distribuées » directement de processus à processus sans passer par un serveur unique qui deviendrait un goulot d'étranglement pour le transfert de ces données. Il y a donc redistribution des données. Dans le cas où les deux composants manipulent des données de nature différente (discrétisation différente du problème traité), l'échange des données distribuées passe par une phase d'interpolation. Nous utilisons pour cela les fonctionnalités de l'outil MED mémoire parallèle ParaMEDMEM [6,7] utilisant les algorithmes de l'outil d'interpolation INTERP_KERNEL.

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.

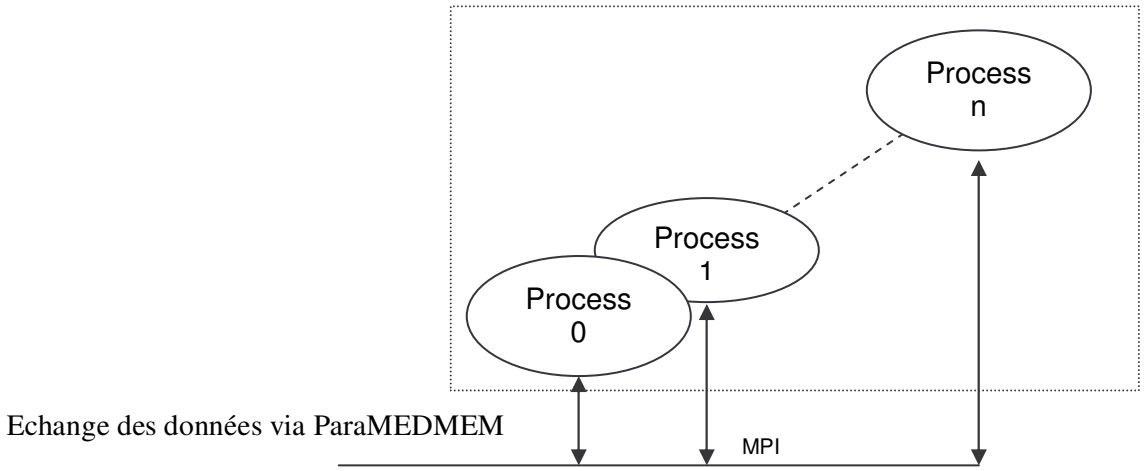
L'approche présentée dans ce document consiste à utiliser un ORB séquentiel pour coupler des composants parallèles. La gestion du parallélisme d'invocation de service Corba est donc entièrement à la charge du composant. Ainsi, seul le processus 0 du composant parallèle est visible depuis le client (superviseur Salomé, script python, …). Une fois activé par le client, le processus 0 invoquera, 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, nous utilisons l'intergiciel MPI à travers ParaMEDMEM. Pour cela nous créons un nouveau communicateur global sur l'ensemble des processus des deux programmes parallèles impliqués en utilisant des fonctionnalités de la norme MPI2, puis nous mettons en œuvre les fonctionnalités de ParaMEDMEM. Il y a ainsi communication directe de processus à processus, ce qui évite tout goulot d'étranglement pour l'échange de données.

Les objectifs de cette étude sont illustrés par un cas d'utilisation qui est décrit au chapitre 4. Il s'agit du couplage de deux instances du code parallèle Triou.

.



Les échanges des données parallèles sont gérés par ParaMEDMEM de façon à ce que la redistribution se fasse de façon optimale directement du processus serveur au processus client avec interpolation éventuelle.



Un composant SALOME est une « entité logicielle » de base qui offre un certain nombre de services. Il est donc accessible depuis n'importe quel client, qu'il soit localisé sur la même machine ou distant. Les différents composants peuvent également échanger des données entre eux. Dans SALOME, un composant est concrétisé par une librairie dynamique et un catalogue.

L'« intergiciel » est la couche logicielle chargée des communications entre les composants dans une architecture répartie. On utilise ici Corba qui définit notamment un service de nommage qui donne un accès à la référence de chacun des objets instanciés et identifiés, mais également MPI.

Un « container » est un gestionnaire de composants sur une machine donnée (station de travail, serveur multi-processeurs, machine vectorielle, machine massivement, …). C'est lui qui réalise l'instanciation et la destruction de composants sur la machine qui l'héberge. 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 librairie correspondant au composant et crée une instance. Il lui affecte une référence Corba et l'enregistre au service de nommage. Le client (superviseur Salomé, script python, …) récupère auprès du service de nommage la référence du composant et peut activer le service voulu.

2.1 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 implémentation, il est utile de définir un objet parallèle générique afin de mutualiser les fonctionnalités communes.

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.

2.1.1 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.

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

2.1.2 Définition de l'implémentation

L'implémentation de l'objet générique parallèle doit définir au moins deux fonctions : la première donne à l'objet parallèle sa liste de références Corba (fonction set), et la deuxième 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,
- une fonction d'échange des références Corba entre processus, via MPI, afin de constituer ladite liste,
- les fonctions de connexion et déconnexion MPI2 de deux programmes parallèles indépendants, avec la liste des communicateurs MPI associés.

```
class MPIObject_i: public virtual POA_Engines::MPIObject
{
 public:
  MPIObject_i();
  ~MPIObject_i();

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

 protected:
  // Numero du processus
  int _numproc;
  // Nombre de processus
  int _nbproc;
  // IOR des objets paralleles sur tous les process mpi
  Engines::IORTab* _tior;
  // Echange des IOR de l'objet entre process
  void BCastIOR(CORBA::ORB_ptr orb,Engines::MPIObject_ptr pobj,bool amiCont);
#ifdef HAVE_MPI2
  // MPI2 connection
  void remoteMPI2Connect(std::string service);
  // MPI2 disconnection
  void remoteMPI2Disconnect(std::string service);
#endif
  std::map<std::string, MPI_Comm> _gcom;

private:
  std::map<std::string, MPI_Comm> _icom;
  std::map<std::string, bool> _srv;
  std::map<std::string, std::string> _port_name;

} ;
```

2.2 Le container parallèle

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

2.2.1 Définition de l'interface Corba

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

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

2.2.2 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 via 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 invoque un service d'un container parallèle, invoque en fait uniquement le service correspondant du processus 0. C'est le service activé du processus 0, qui va ensuite invoquer 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 invoqué 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 invoque le service de destruction d'un composant du container situé sur le processus 0. Ce service de destruction invoque alors le même service sur les autres processus, puisqu'il possède la liste des références de ces objets Corba.

2.3 Le composant parallèle générique

L'objet ParaMEDMEMComponent est un objet générique qui représente un composant parallèle MPI qui échange ses données distribuées via ParaMEDMEM. Il est défini par un fichier idl dont l'interface dérive des deux objets du Kernel de Salomé: Component et MPIObject.

```
module SALOME_MED
{
  interface ParaMEDMEMComponent:Engines::Component,Engines::MPIObject
  {
    void setInterpolationOptions(in string coupling,
                         in long print_level,
                         in string intersection_type,
                         in double precision,
                         in double median_plane,
                         in boolean do_rotate,
                         in double bounding_box_adjustment,
                         in double bounding_box_adjustment_abs,
                         in double max_distance_for_3Dsurf_intersect,
                         in long orientation,
                         in boolean measure_abs,
                         in string splitting_policy,
                         in boolean P1P0_bary_method) raises
(SALOME::SALOME_Exception);
    void initializeCoupling(in string coupling) raises (SALOME::SALOME_Exception);
    void terminateCoupling(in string coupling) raises (SALOME::SALOME_Exception);
  };
};
```

Le servant C++ de cet objet dérive bien sûr des objets Engines_Component_i et MPIObject_i.

Un composant parallèle peut échanger des données avec un ou plusieurs autres composants parallèles. Pour chaque couplage parallèle, il faut définir deux groupes de communicateurs : source et target et un canal d'échange de données (DEC) [6, 7]. Un couplage est défini par une chaîne de caractères, les groupes de communicateurs et les canaux d'échanges de données sont stockés dans des objets map, identifiés par le nom du couplage correspondant.

Ainsi, le servant ParaMEDMEMComponent_i possède deux attributs pour les communicateurs source et target et un attribut pour les canaux d'échange de données.

```
namespace ParaMEDMEM
{
  class ParaMEDMEMComponent_i : public virtual POA_SALOME_MED::ParaMEDMEMComponent,
public Engines_Component_i, public MPIObject_i
  {

  public:
    // Constructor
    ParaMEDMEMComponent_i();
    ParaMEDMEMComponent_i(CORBA::ORB_ptr orb,
                          PortableServer::POA_ptr poa,
                          PortableServer::ObjectId * contId,
                          const char *instanceName,
                          const char *interfaceName,
                          bool regist);
```

```
    // Destructor
    ~ParaMEDMEMComponent_i();
    void setInterpolationOptions(const char * coupling,
                                 CORBA::Long print_level,
                                 const char * intersection_type,
                                 CORBA::Double precision,
                                 CORBA::Double median_plane,
                                 CORBA::Boolean do_rotate,
                                 CORBA::Double bounding_box_adjustment,
                                 CORBA::Double bounding_box_adjustment_abs,
                                 CORBA::Double max_distance_for_3Dsurf_intersect,
                                 CORBA::Long orientation,
                                 CORBA::Boolean measure_abs,
                                 const char * splitting_policy,
                                 CORBA::Boolean P1P0_bary_method )
throw(SALOME::SALOME_Exception);
    void initializeCoupling(const char * coupling) throw(SALOME::SALOME_Exception);
    void terminateCoupling(const char * coupling) throw(SALOME::SALOME_Exception);
    void _getOutputField(const char * coupling, MEDCouplingFieldDouble* field);

  protected:
    void _setInputField(const char * coupling,
SALOME_MED::MPIMEDCouplingFieldDoubleCorbaInterface_ptr fieldptr,
MEDCouplingFieldDouble* field);

  private:
    CommInterface* _interface;
    std::map<std::string,InterpKernelDEC*> _dec;
    std::map<std::string,MPIProcessorGroup*> _source, _target;
    std::map<std::string,ProcessorGroup*> _comm_group;
    std::map<std::string,INTERP_KERNEL::InterpolationOptions*> _dec_options;
  };
};
```

La méthode initializeCoupling() prend en entrée un argument chaîne de caractères pour définir le couplage à initialiser. Cette méthode commence par réaliser un rendez-vous MPI2 à l'aide de la méthode du Kernel : remoteMPI2Connect() héritée de la classe MPIObject. Cette méthode crée un communicateur global _gcom stocké dans un objet map de la classe MPIObject du Kernel. Ensuite la méthode initializeCoupling() crée les groupes de processeurs source et target qu'il enregistre dans sa map.

Deux méthodes d'échange de données _setInputField() et _getOutputField() prennent en entrée les arguments suivants : la chaîne de caractères qui identifie le couplage et un pointeur MEDCouplingFieldDouble. La méthode _setInputField() prend un troisième argument qui est le pointeur CORBA du champ parallèle côté serveur.

Dans le cas d'un premier appel à ces deux méthodes, il faut créer un canal d'échange de données et l'enregistrer dans la map correspondante, attacher le champ à ce DEC qui se charge de calculer la matrice d'interpolation. Ensuite dans tous les appels suivants, on échange les données en utilisant le canal existant.

La méthode terminateCoupling() prend en entrée une chaîne de caractères qui identifie le couplage à finaliser. Elle appelle la méthode de fermeture de la connexion MPI2, détruit les objets groupes de processeurs et canal d'échange des données, et enlève ces objets des maps correspondantes.

2.4 Le composant parallèle

Le composant parallèle hérite du composant générique ParaMEDMEMComponent (voir paragraphe 2.3). 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 invoqué 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 invoque 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'exécute en parallèle, le processus 0 doit les invoquer en mode asynchrone : c'est à dire invoquer le service des autres processus et continuer sa propre exécution sans attendre que les services invoqués ne soit terminés. Ainsi les autres processus sont invoqués par le processus 0 dans des threads.

2.5 Les données parallèles

Les données parallèles échangées par des composants parallèles sont essentiellement des champs distribués. Chacun des processus héberge un objet de type MEDCouplingField. L'échange et la redistribution avec une interpolation éventuelle des valeurs de ce champ distribué entre deux composants parallèles sont réalisés par l'outil ParaMEDMEM [6, 7].

3 L'interface ICoCo

L'interface ICoCo propose une interface en vue de couplages multi-échelles ou multi-disciplines de codes. Elle propose une liste de services standards qu'un code doit fournir pour être facilement couplé. Elle fixe des règles, des spécifications sur le comportement que doivent avoir ces méthodes et sur la façon de les appeler.

Cette interface repose sur la notion de problème et de champ. L'ensemble des informations échangées dans le cadre des couplages peut être représenté par des champs. Un problème est un objet informatique dont la fonction est de calculer des champs sur un domaine d'espace et de temps. Il présente une interface standardisée.

4 Cas d'utilisation : le couplage TRIOU-TRIOU

Il s'agit de montrer comment à partir d'un exemple de composant C++ [8] parallèle SPMD, on réalise un composant parallèle dans Salomé. Le composant C++ parallèle est définit par une interface représentée par une classe C++. Cette classe est conforme à l'interface ICoCo [5] qui définit un standard pour faciliter le couplage de codes (voir paragraphe 3). L'objectif de ce paragraphe est de montrer comment l'interface Salomé du composant parallèle est bâtie à partir de l'objet purement C++. On va ainsi construire un composant Salomé qui échange ses données distribuées (les champs) via une référence Corba qui passe d'un code parallèle à un autre. Ce processus est essentiellement pédagogique sachant que le composant Salomé parallèle sera généré automatiquement à terme, par l'outil HXX2SALOME à partir du composant C++.

4.1 L'interface du composant C++ TRIOU

Voici l'interface du composant C++ TRIOU :

```
namespace ParaMEDMEM {
    class MEDCouplingFieldDouble;
}

namespace ICoCo {

  class Init_Params;

  class Problem {

  public :

    // interface Problem
    Problem();

    Problem(void* data);
    void set_data(void* data);
    void set_data_file(const std::string& file);
    virtual ~Problem()  ;
    virtual bool initialize();
    virtual void terminate();
    virtual double presentTime() const;
    virtual double computeTimeStep(bool& stop) const;
    virtual bool initTimeStep(double dt);
    virtual bool solveTimeStep();
    virtual void validateTimeStep();
    virtual bool isStationary() const;
    virtual void abortTimeStep();
    virtual bool iterateTimeStep(bool& converged);

    // interface salome
    ParaMEDMEM::MEDCouplingFieldDouble* getOutputField(const std::string& name)
const ;
```

```
    ParaMEDMEM::MEDCouplingFieldDouble* getInputFieldTemplate(const std::string&
name) const;
    void setInputField(const std::string& name, const
ParaMEDMEM::MEDCouplingFieldDouble* afield);
  protected :

    Init_Params* my_params;
    Probleme_U* pb;
    mon_main* p;

  };
}
```

4.2 L'interface Corba du composant Salomé parallèle

Voici l'interface Corba du composant Salomé parallèle TRIOU :

```
module TRIOU
{
  interface TRIOU_Gen:SALOME_MED::ParaMEDMEMComponent
  {
    void configure(in string work_directory) raises (SALOME::SALOME_Exception);
    void set_data(in string data_file,in string problem_name) raises
(SALOME::SALOME_Exception);
    void initialize() raises (SALOME::SALOME_Exception);
    double computeTimeStep(out boolean stop) raises (SALOME::SALOME_Exception);
    boolean initTimeStep(in double dt) raises (SALOME::SALOME_Exception);
    boolean solveTimeStep() raises (SALOME::SALOME_Exception);
    SALOME_MED::MPIMEDCouplingFieldDoubleCorbaInterface getOutputField(in string
name) raises (SALOME::SALOME_Exception);
    void setInputField(in string name,in
SALOME_MED::MPIMEDCouplingFieldDoubleCorbaInterface field, in string coupling)
raises (SALOME::SALOME_Exception);
    void abortTimeStep() raises (SALOME::SALOME_Exception);
    void validateTimeStep() raises (SALOME::SALOME_Exception);
    boolean isStationary() raises (SALOME::SALOME_Exception);
    void terminate() raises (SALOME::SALOME_Exception);
  };
};
```

L'interface TRIOU_Gen hérite de l'interface ParaMEDMEMComponent définie au paragraphe 2.3. Ainsi le composant Salomé parallèle bénéficie des fonctionnalités parallèles de base : connection (et déconnection) MPI2 à un autre composant Salomé parallèle, instanciation de Canaux d'Echange de Données de l'outil ParaMEDMEM et échange des champs distribués avec interpolation éventuelle selon le cas d'utilisation.

On a rajouté dans l'interface Corba une méthode configure() qui prend en argument d'entrée le répertoire courant dans lequel va s'exécuter le code parallèle. Ceci permet de pallier le fait que le code parallèle TRIOU doit s'exécuter obligatoirement dans un répertoire de données spécifique. Lorsque le code parallèle est exécuté en dehors de Salomé, il doit être lancé obligatoirement dans le répertoire de données. Lorsqu'il est lancé dans Salomé, il est nécessaire de faire ce changement de répertoire, ce qui évite d'obliger de lancer la plateforme Salomé dans ce répertoire de données.

A chaque méthode du composant C++ correspond une méthode du composant Salomé avec les mêmes arguments aux exceptions suivantes :

- La méthode set_data du composant C++ prend un pointeur void* en entrée qui correspond en fait à l'adresse d'une structure Init_Params. Cette structure a quatre champs : le nom du fichier de données à prendre en entrée, le nom du problème à résoudre, le communicateur MPI et un booléen qui indique si on travaille en parallèle ou en séquentiel. Dans le cas de l'intégration du composant parallèle dans la plateforme Salomé, les deux derniers champs sont imposés : le booléen est mis en mode parallèle et le communicateur est mis à la valeur MPI_COMM_WORLD. Seuls les deux premiers champs sont variables. On retrouve ainsi ces deux champs comme arguments de la méthode de l'interface Corba.

- Le pointeur sur le champ C++ MEDCouplingFieldDouble échangé dans les méthodes getOutputField() et setInputField() est remplacé par la référence Corba correspondant à ce champ : MPIMEDCouplingFieldDoubleCorbaInterface.
- La méthode getInputFieldTemplate() du composant C++ n'apparait pas dans l'interface Corba du composant Salomé. En effet cette méthode ne sert qu'à donner le pointeur C++ correspondant au champ reçu par le composant. Cette méthode n'est donc pas explicitée au niveau de l'interface Corba, mais est appelée en interne de la méthode setInputField() de l'interface Corba.
- La méthode setInputField() de l'interface Corba prend un argument supplémentaire par rapport à la même méthode du composant C++. Cet argument est la chaîne de caractère « coupling » qui identifie le couplage entre deux codes parallèles. Il permet de spécifier au composant parallèle Salomé le Canal d'Echange de Données qu'il doit utiliser pour réaliser l'échange du champ distribué (voir paragraphe 2.3).

4.3 L'objet MPIMEDCouplingFieldDoubleCorbaInterface

Cet objet représente l'interface Corba du champ distribué hébergé par un composant parallèle. Il s'agit d'une collection d'objets Corba.

```
module SALOME_MED
{
  interface
MPIMEDCouplingFieldDoubleCorbaInterface:ParaMEDCouplingFieldDoubleCorbaInterface
  {
    void getDataByMPI(in string coupling) raises (SALOME::SALOME_Exception);
  };
```

Il hérite de la classe ParaMEDCouplingFieldDoubleCorbaInterface. Il a la méthode supplémentaire getDataByMPI() qui permet au client de demander au serveur de lui envoyer les valeurs du champ par MPI.

L'objet ParaMEDCouplingFieldDoubleCorbaInterface hérite de l'objet MEDCouplingFieldDoubleCorbaInterface qui permet de transférer les valeurs d'un champ séquentiel par Corba, et de l'objet MPIObject présenté au paragraphe 2.1 qui représente un objet parallèle générique. C'est donc également une collection d'objets Corba.

```
module SALOME_MED
{
  interface ParaMEDCouplingUMeshCorbaInterface : MEDCouplingUMeshCorbaInterface,
Engines::MPIObject
  {
  };

  interface ParaMEDCouplingFieldDoubleCorbaInterface :
MEDCouplingFieldDoubleCorbaInterface, Engines::MPIObject
  {
  };
};
```

4.4 L'implémentation du composant Salomé parallèle

4.4.1 Mise en œuvre du parallélisme

C'est le rôle du processus 0 d'invoquer via Corba la même méthode sur les autres processus. Comme cette méthode doit s'exécuter en parallèle sur tous les processus, le processus 0 doit invoquer les autres processus de manière asynchrone. Ceci est fait par l'intermédiaire de threads. Corba propose bien un mode asynchrone via le type « oneway », mais celui-ci n'est pas standard d'une implémentation à l'autre et ne permet pas de gérer les exceptions. Ainsi la structure de chacune des méthodes d'un servant parallèle est la suivante :

- Si je suis le processus 0 j'invoque la même méthode sur les autres processus par l'intermédiaire de threads. Tous les arguments d'entrée de la méthode doivent être propagés aux autres processus.
- J'exécute la méthode
- Si je suis le processus 0 je termine proprement chacun des threads en propageant les éventuelles erreurs venues des autres processus, vers le client
- Si je suis le processus 0, je renvoie la donnée de sortie éventuelle au client

L'exemple ci-dessous concernant la méthode computeTimeStep() illustre la méthode. Ainsi en plus des arguments d'entrée, on envoie au thread deux arguments qui sont le numéro de processus : ip et le tableau des références corba correspondant au composant parallèle Salomé sur les autres processus : _tior. Ces données permettront au thread d'activer la méthode computeTimeStep() du composant sur le processus ip.

```
typedef struct
{
  bool exception;
  std::string msg;
} except_st;

typedef struct
{
  int ip;
  bool stop;
  Engines::IORTab *tior;
} thread_str;

double Triou_i::computeTimeStep(bool & stop) throw(SALOME::SALOME_Exception)
{
  except_st *est;
  void *ret_th;
  pthread_t *th;

  if(_numproc == 0)
    {
// invocation de la meme methode sur les autres processeurs dans des threads
```

```
      th = new pthread_t[_nbproc];
      for(int ip=1;ip<_nbproc;ip++)
        {
          thread_str *st = new thread_str;
          st->ip = ip;
          st->tior = _tior;
          st->stop = stop;
          pthread_create(&(th[ip]),NULL,th_computetimestep,(void*)st);
        }
    }

  double ret;
  try
    {
// invocation de la methode du composant C++
      ret = _triou->computeTimeStep(stop);
    }
  catch(const std::exception &ex)
    {
      MESSAGE(ex.what());
      THROW_SALOME_CORBA_EXCEPTION(ex.what(),SALOME::INTERNAL_ERROR);
    }

  if(_numproc == 0)
    {
// A la fin des threads on leve une exception Corba si une exception est apparue
// dans un thread
      for(int ip=1;ip<_nbproc;ip++)
        {
          pthread_join(th[ip],&ret_th);
          est = (except_st*)ret_th;
          if(est->exception)
            {
              ostringstream msg;
              msg << "[" << ip << "] " << est->msg;

THROW_SALOME_CORBA_EXCEPTION(msg.str().c_str(),SALOME::INTERNAL_ERROR);
            }
          delete est;
        }
      delete[] th;
    }

  return ret;
}
```

4.4.2 Données retournées

Si la méthode renvoie une donnée, celle-ci peut-être un champ distribué. Le processus 0 renvoie alors la référence Corba de l'objet parallèle. Si c'est une donnée simple comme un réel, un entier ou un booléen, le composant C++ doit faire une opération de réduction afin que chaque processus renvoie une valeur identique et cohérente.

4.4.3 Gestion des erreurs

La fonction qui s'exécute dans chacun des threads est la suivante :

```
void *th_computetimestep(void *s)
{
  ostringstream msg;
  thread_str *st = (thread_str*)s;
  except_st *est = new except_st;
  est->exception = false;

  try
    {
// invocation de la methode sur un autre processeur par activation de l'objet
// Corba correspondant a ce processeur
      TRIOU::TRIOU_Gen_var compo=TRIOU::TRIOU_Gen::_narrow((*(st->tior))[st->ip]);
      compo->computeTimeStep(st->stop);
    }
// en cas d'exception, l'erreur est renvoyee au processeur 0
  catch(const SALOME::SALOME_Exception &ex)
    {
      est->exception = true;
      est->msg = ex.details.text;
    }
  catch(const CORBA::Exception &ex)
    {
      est->exception = true;
      msg << "CORBA::Exception: " << ex;
      est->msg = msg.str();
    }
  delete st;
  return((void*)est);
}
```

Ainsi la méthode commence par déterminer la référence Corba du composant parallèle Salomé sur le processus ip. Elle peut ensuite activer la méthode computeTimeStep() sur ce processus en lui passant les arguments d'entrée. Pour la gestion des erreurs une structure « except_st » est utilisée :

```
typedef struct
{
  bool exception;
  string msg;
} except_st;
```

Cette structure contient deux champs : un booléen qui indique s'il y a eu une erreur et une chaîne de caractères contenant le message d'erreur éventuel. Ensuite la structure d'erreur est renvoyée au processus 0 qui a lancé le thread. Ce dernier peut alors tester une erreur éventuelle et lancer une exception Corba vers le client si nécessaire.

```
  if(_numproc == 0)
    {
      for(int ip=1;ip<_nbproc;ip++)
```

```
        {
          pthread_join(th[ip],&ret_th);
          est = (except_st*)ret_th;
          if(est->exception)
            {
              msg << "[" << ip << "] " << est->msg;
              THROW_SALOME_CORBA_EXCEPTION( msg.str().c_str(),
                                            SALOME::INTERNAL_ERROR);
            }
          delete est;
        }
      delete[] th;
    }
}
```

4.4.4 Mise à dispostion du champ distribué par le code serveur

Le composant Corba Salomé commence par appeler la méthode du composant C++ qui renvoie un objet de type MEDCouplingFieldDouble. Ensuite il instancie un objet Corba champ parallèle de type MPIMEDCouplingFieldDoubleServant (voir paragraphe 4.3), il décremente le compteur de référence du champ MEDCoupling et crée la référence Corba qui sera envoyée au client.

```
  try
    {
// recuperation du champ a partir du composant C++
      ParaMEDMEM::MEDCouplingFieldDouble *field = _triou->getOutputField(name);
      if( field )
        {
// instanciation d'un objet champ Corba parallele
          ParaMEDMEM::MPIMEDCouplingFieldDoubleServant *m=new
ParaMEDMEM::MPIMEDCouplingFieldDoubleServant(_orb,this,field);
          field->decrRef();
          SALOME_MED::MPIMEDCouplingFieldDoubleCorbaInterface_ptr ret=m->_this();
        }
      else
        {
          msg << "No Field of name " << name;
          THROW_SALOME_CORBA_EXCEPTION(msg.str().c_str(),SALOME::INTERNAL_ERROR);
        }
    }
  catch(const std::exception &ex)
    {
      THROW_SALOME_CORBA_EXCEPTION(ex.what(),SALOME::INTERNAL_ERROR);
    }
```

4.4.5 Récupération du champ distribué par le code client

Le composant Corba Salomé commence par récupérer le pointeur C++ dans lequel il doit stocker le champ lu, par l'intermédiaire de la méthode getInputFieldTemplate() (voir paragraphe 4.2). Puis il active la méthode _setInputField() du composant parallèle générique ParaMEDMEMComponent (voir paragraphe 2.3). C'est cette méthode qui va exécuter l'échange des données distribuées du champ parallèle par l'outil ParaMEDMEM via MPI. Ensuite le pointeur du champ lu est donné au composant C++ et le compteur de référence du champ MEDCoupling est décrémenté.

```
  try
    {
// recuperation du pointeur dans lequel sera stocke le champ dans le composant C++
      ParaMEDMEM::MEDCouplingFieldDouble *field = _triou->getInputFieldTemplate(
                                                                        name);
      if( field )
        {
// appel de la methode du composant generique ParaMEDMEMComponent qui echange les
// valeurs du champ via MPI
          _setInputField(coupling,fieldptr,field);
// Le pointeur qui contient le champ est ensuite donne au composant C++
          _triou->setInputField(name,field);
          field->decrRef();
        }
      else
        {
          msg << "No Field of name " << name;
          THROW_SALOME_CORBA_EXCEPTION(msg.str().c_str(),SALOME::INTERNAL_ERROR);
        }
    }
  catch(const std::exception &ex)
    {
      THROW_SALOME_CORBA_EXCEPTION(ex.what(),SALOME::INTERNAL_ERROR);
    }
```

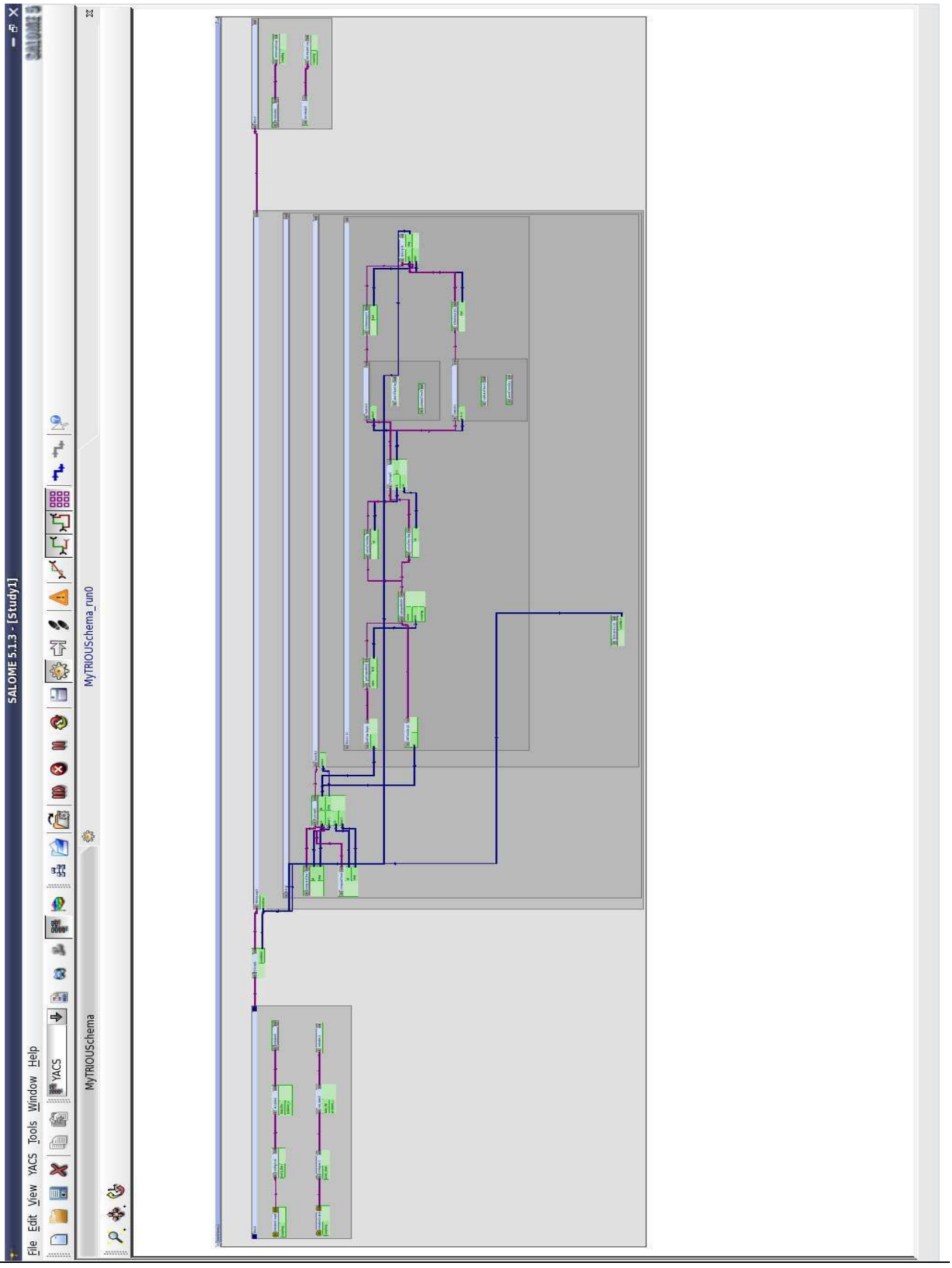
4.4.6 Le catalogue du composant parallèle

Le catalogue du composant doit contenir l'ensemble des méthodes du composant Salomé, ainsi que l'ensemble des méthodes de la classe dont il hérite, à savoir le composant générique parallèle : ParaMEDMEMComponent. Ainsi il ne faut pas oublier les méthodes suivantes :

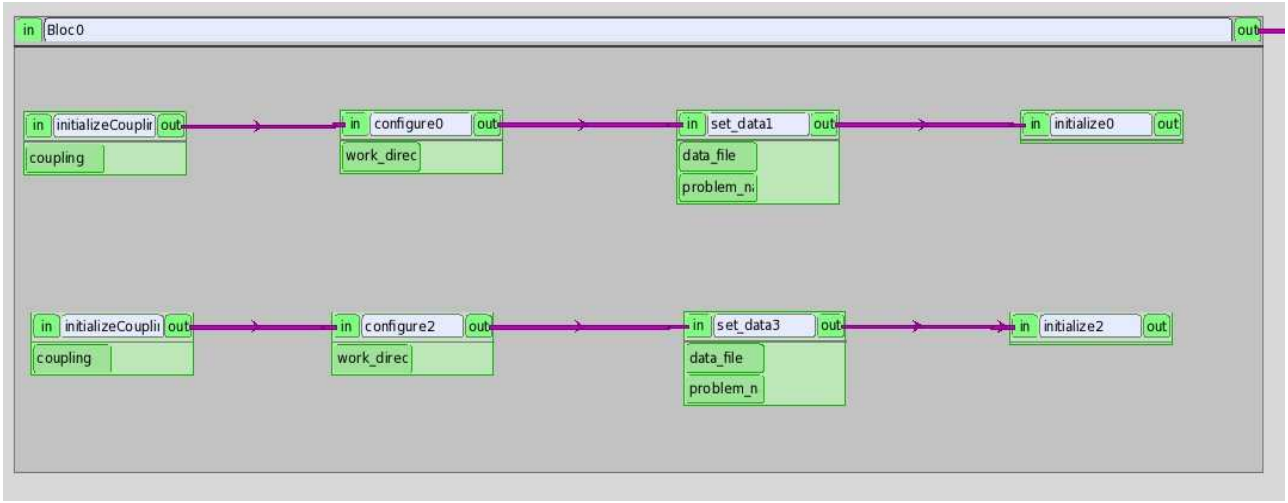
- initializeCoupling()
- setInterpolationOptions()
- terminateCoupling()

4.5 Le schéma de couplage

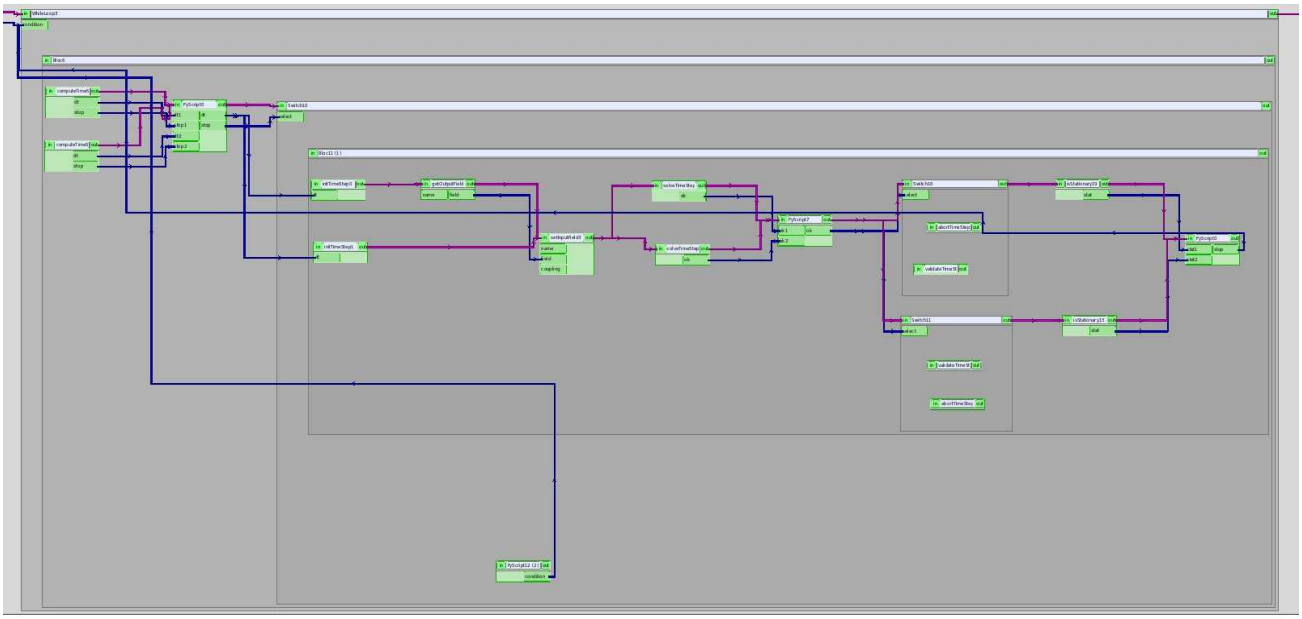
Le schéma de couplage du cas Triou-Triou est le suivant :



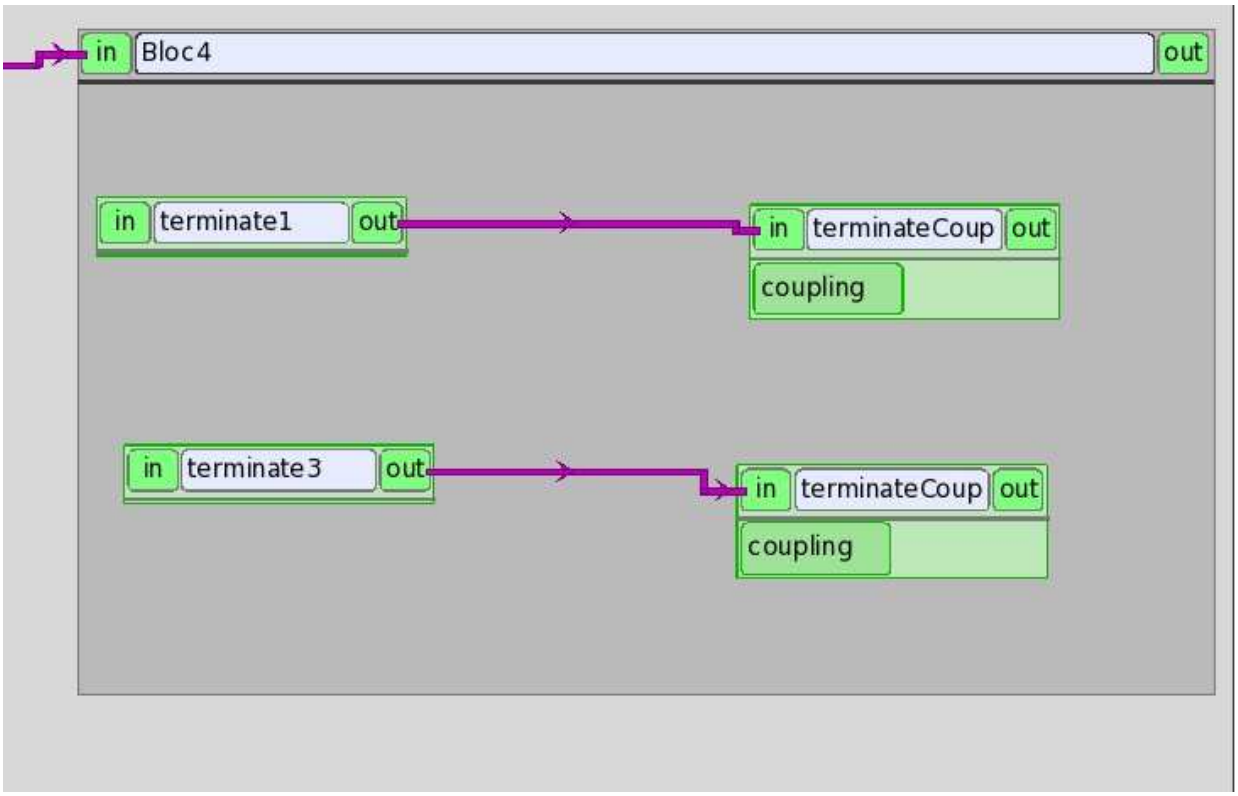
Initialisation du schéma :



Boucle en temps :



Sortie du schéma :



Ce schéma correspond au script C++ suivant :

```
#include <iostream>
#include "math.h"
#include "utilities.h"
#include <unistd.h>
#include "SALOMEconfig.h"
#include CORBA_CLIENT_HEADER(SALOME_Component)
#include CORBA_CLIENT_HEADER(Triou)
#include "SALOME_NamingService.hxx"
#include "SALOME_LifeCycleCORBA.hxx"
#include "OpUtil.hxx"
#include "Utils_ORB_INIT.hxx"
#include "Utils_SINGLETON.hxx"
#include "Utils_SALOME_Exception.hxx"
#include "Utils_CommException.hxx"
#include <string>
#define EPS 1.e-12
using namespace std;

void * initializecoupling(void *);
void * terminatecoupling(void *);

typedef struct {
  string service;
```

```
  string filename;
  TRIOU::TRIOU_Gen_var compo;
} thread_st;

int main (int argc, char * argv[]) {

  Engines::MachineParameters params;
  TRIOU::TRIOU_Gen_var A, B, C;
  void *statusp;
  thread_st *st;

  // Initializing omniORB
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  // Obtain a reference to the root POA
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA") ;
  PortableServer::POA_var poa = PortableServer::POA::_narrow(obj) ;

  // Use Name Service to find container
  SALOME_NamingService *NS=new SALOME_NamingService(orb);

  SALOME_LifeCycleCORBA LCC(NS);

  Engines::Component_var objComponent;

  try {
    // Load MYMPICOMPO Component
    LCC.preSet(params); // default initialisation (empty strings, 0...)
    params.container_name = "Triou1";
    params.OS = "LINUX";
    params.mem_mb = 0;
    params.nb_proc_per_node = 2;
    params.nb_node = 1;
    params.isMPI = true;
    objComponent = LCC.FindOrLoad_Component(params,"TRIOU");
    ASSERT(!CORBA::is_nil(objComponent));
    A = TRIOU::TRIOU_Gen::_narrow(objComponent);
    ASSERT(!CORBA::is_nil(A));

    SCRUTE(A->instanceName());

    // Load MYCOMPO Component
    LCC.preSet(params); // default initialisation (empty strings, 0...)
    params.container_name = "Triou2";
    params.OS = "LINUX";
    params.nb_proc_per_node = 2;
    params.nb_node = 1;
    params.isMPI = true;
    objComponent = LCC.FindOrLoad_Component(params,"TRIOU");
    ASSERT(!CORBA::is_nil(objComponent));
    B = TRIOU::TRIOU_Gen::_narrow(objComponent);
    ASSERT(!CORBA::is_nil(B));

    SCRUTE(B->instanceName());

    pthread_t t1;
    st = new thread_st;
```

```
    st->service = "TRIOU-TRIOU";
    st->compo = A;
    pthread_create(&t1,NULL,&initializecoupling,(void*)st);
    B->initializeCoupling("TRIOU-TRIOU");
    pthread_join(t1,NULL);

    A->configure("/data/tmplgls/secher/Livraison_Composant_Trio_3/cas");
    B->configure("/data/tmplgls/secher/Livraison_Composant_Trio_3/cas");

    A->setInterpolationOptions("TRIOU-TRIOU",0,"Triangulation",1e-
12,0.,true,0.,0.,0.,0,true,"GENERAL_48",false);
    B->setInterpolationOptions("TRIOU-TRIOU",0,"Triangulation",1e-
12,0.,true,0.,0.,0.,0,true,"GENERAL_48",false);

    A->set_data("boite","pb_ch");
    B->set_data("canal","pb_ch");

    A->initialize();
    B->initialize();

    bool stop1=false;
    bool stop2=false;
    bool stop=false;
    bool stat1, stat2, stat;

    while(!stop){

      bool ok1=false;
      bool ok2=false;
      bool ok=false;

      while(!ok && !stop){

        double dt1 = A->computeTimeStep(stop1);
        double dt2 = B->computeTimeStep(stop2);
        double dt = dt1;
        if(dt2 < dt1) dt = dt2;
        if(stop1 || stop2){
          stop1 = true;
          stop2 = true;
          stop = true;
        }

        if(stop)
          break;

        A->initTimeStep(dt);
        B->initTimeStep(dt);

        SALOME_MED::MPIMEDCouplingFieldDoubleCorbaInterface_ptr fvitesse=A-
>getOutputField("VITESSE_ELEM_out");

        B->setInputField("vitesse_entree",fvitesse,"TRIOU-TRIOU");

        ok1 = A->solveTimeStep();
        ok2 = B->solveTimeStep();
        if(ok1 && ok2) ok = true;
```

```
      if(!ok){
        A->abortTimeStep();
        B->abortTimeStep();
      }
      else{
        A->validateTimeStep();
        B->validateTimeStep();
      }

    }

    stat1 = A->isStationary();
    stat2 = B->isStationary();
    if(stat1 && stat2) stop = true;

  }

  A->terminate();
  B->terminate();

  pthread_t t3;
  st = new thread_st;
  st->service = "TRIOU-TRIOU";
  st->compo = A;
  pthread_create(&t3,NULL,&terminatecoupling,(void*)st);
  B->terminateCoupling("TRIOU-TRIOU");
  pthread_join(t3,NULL);

  cout << "test OK" << endl;

  // Clean-up.
  orb->destroy();

}
catch(SALOME::SALOME_Exception& ex) {
  cerr << "Caught a Salome exception: " << ex.details.text << endl;
  cerr << "test KO" << endl;
}
catch(CORBA::COMM_FAILURE) {
  cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
       << "object." << endl;
  cerr << "test KO" << endl;
}
catch(CORBA::SystemException& e) {
  cerr << "Caught a CORBA::SystemException." << endl;
  cerr << "test KO" << endl;
}
catch(CORBA::Exception& e) {
  cerr << "Caught CORBA::Exception." << endl;
  cerr << "test KO" << endl;
}
catch(...) {
  cerr << "Caught unknown exception." << endl;
  cerr << "test KO" << endl;
}
```

```
  return 0;
}

void * initializecoupling(void *arg)
{
  thread_st *st = (thread_st*)arg;
  try {
    st->compo->initializeCoupling(st->service.c_str());
  }
  catch(SALOME::SALOME_Exception& ex) {
    cerr << "Caught a Salome exception in a thread: " << ex.details.text << endl;
  }
  delete st;
}

void * terminatecoupling(void *arg)
{
  thread_st *st = (thread_st*)arg;
  try {
    st->compo->terminateCoupling(st->service.c_str());
  }
  catch(SALOME::SALOME_Exception& ex) {
    cerr << "Caught a Salome exception in a thread: " << ex.details.text << endl;
  }
  delete st;
}
```

5 Conclusion et perspectives

Le couplage Triou/Triou a permis de valider sur un cas réel les mécanismes de couplages de codes parallèles. Comme nous venons de le voir la construction du composant parallèle Salomé n'est pas simple et demande une certaine compétence technique qu'il n'est pas possible de demander à tous les utilisateurs d'acquérir. Il est donc indispensable de mettre en place un outil automatique chargé de générer ce composant à partir de son API purement C++. Cet outil existe déjà, il s'agit de HXX2SALOME. Il est par contre utile de le faire évoluer afin de tenir compte du parallélisme. Une fois cet outil étendu, chaque utilisateur de la plateforme Salomé bénéficiera d'un outil performant capable de générer automatiquement un composant parallèle Salomé à partir de son implémentation purement C++.

Les appels aux méthodes initializeCoupling() et terminateCoupling() étant systématiques pour chacun des couplages à réaliser, on peut imaginer que ces appels ne doivent pas être explicités par l'utilisateur mais appelés automatiquement par le module YACS. Cela permettrait de simplifier le schéma de calcul réalisé par l'utilisateur.

6 Références

[1] http://www.salome-platform.org
[2] Introduction du parallélisme dans l'architecture logicielle du projet PAL/SALOME – Bernard Sécher – SFME/LGLS/RT/02-002/A – 30 mai 2002
[3] Spécifications du couplage de composants parallèles dans Salomé – Bernard Sécher – SFME/LGLS/RT-09-010/A – 10 juin 2009
[4] http://ralyx.inria.fr/2002/Raweb/paris/uid26.html
[5] Proposition d'une interface de couplage en vue des couplages de codes dans Neptune – Fabien Perdu – SSTH/LMDL/2006-010 – 21 avril 2006
[6] Proposition de spécifications de MEDMEM parallèle – Vincent Bergeaud – SFME/LGLS/RT/07-008/1 – 2 juillet 2007
[7] Spécifications détaillées et conception de MEDMEM para – Vincent Bergeaud et Michaël Ndjinga – SFME/LGLS/RT/07-013/A – 10 octobre 2007
[8] Intégration de composants dans l'environnement PAL/Salomé – Marc Tajchman – SFME/LGLS/RT/03-002/A – 6 mars 2003