Coupleur OpenPALM version 4.3.0, Manuel utilisateur et de formation

T. Morel

Coupleur OpenPALM
version 4.3.0

Manuel utilisateur et de formation

Thierry Morel [1], Florent Duchaine [1], Anthony Thévenin [1], Andrea Piacentini [1] , Moritz Kirmse [1] et Eric Quémerais [2]

Avril 2019

TR-CMGC-19-69

1: CERFACS, Global Change and Climate Modeling Team, 42 avenue G. Coriolis, 31 057 Toulouse Cedex 01, France
2 : Department of Fundamental and Applied Energetics (DEFA), ONERA, 29 Avenue de la Division Leclerc, 92 322 Châtillon Cedex, France

SOMMAIRE

Introduction

Le coupleur OpenPALM est issu du coupleur PALM développé au CERFACS depuis 1998 et de la bibliothèque d'interpolation CWIPI développée à l'ONERA DSNA/ELCI. C'est un logiciel libre distribué sous licence LGPL, il est co-développé par le CERFACS et l'ONERA depuis janvier 2011.

Son originalité tient à ses possibilités de définition d'algorithmes complexes autour des codes à coupler, à son efficacité et sa souplesse pour transférer des données d'un code à l'autre. La technologie MPI sur laquelle il s'appuie pour le lancement de processus et l'échange de données entre les codes en fait un logiciel portable et optimisé sur toutes machines du monde unix/linux.

OpenPALM présente de nombreuses fonctionnalités de couplage allant jusqu'à des fonctionnalités d'interpolation géométrique de tous types de maillages.

Ce manuel, organisé sous forme de sessions de formation, montre pas à pas toutes les fonctionnalités offertes par le coupleur OpenPALM. La prise en main du logiciel est facilitée par l'interface graphique PrePALM.

1 Session 1 : Prise en main de l'interface graphique

1.1 Introduction

La mise au point d'une application PALM passe nécessairement par l'interface graphique **PrePALM**. Il est très important d'en maîtriser toutes les subtilités pour tirer parti au maximum des fonctionnalités du coupleur. On passe en général plus de temps dans l'outil **PrePALM** que dans la modification du code source des programmes à coupler. La raison principale vient du fait que l'algorithme du couplage est entièrement décrit dans l'interface graphique et que la plupart des fonctionnalités du coupleur sont définies via l'interface graphique qui contrôle la cohérence des données saisies.

1.2 Lancement de PrePALM

Une fois le logiciel **PrePALM** installé, chaque utilisateur doit définir un alias qui permet de définir le chemin du logiciel et le nom de la commande **PrePALM,** par exemple en tcsh :

```
alias prepalm  setenv PREPALMMPDIR chemin_de_l'install ; $PREPALMMPDIR/PrePALM_MP.tcl !* &
```

Ou en bash :

```
function prepalm {
export PREPALMMPDIR=install-path
$PREPALMMPDIR/prepalm_MP.tcl $* &
}
```

Cette commande trouve naturellement sa place dans le fichier de configuration de l'utilisateur (.cshrc, .bashrc,… selon le système utilisé) ou dans un script lancé par celui-ci. Dans ce même fichier, on peut également définir la variable d'environnement PREPALMEDITOR pour éditer des fichiers depuis l'interface graphique. Si cette variable n'est pas initialisée PrePALM utilisera l'éditeur vi . Si par exemple on est plus familier avec l'éditeur *emacs*, il suffit de déclarer :

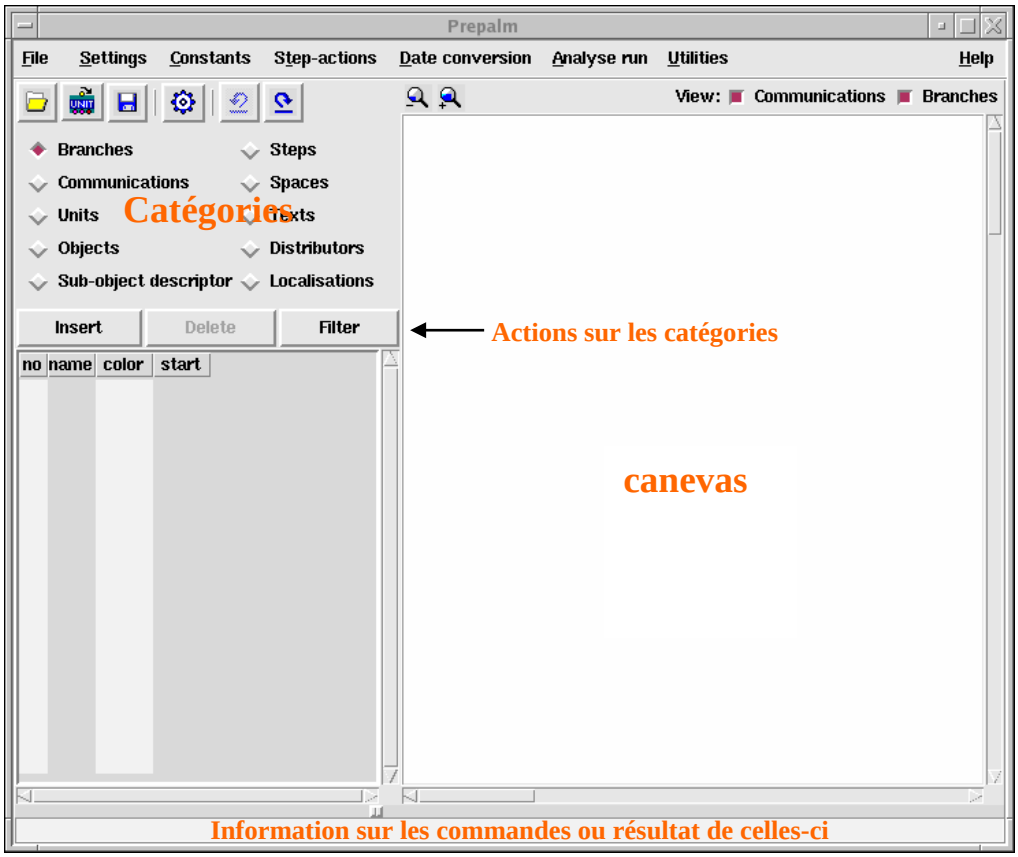
`setenv PREPALMEDITOR emacs` ou `export PREPALMEDITOR=emacs`

L'alias étant défini, pour lancer l'interface graphique, il suffit d'entrer la commande :

> prepalm

Exercez-vous !

- Positionnez vous dans le répertoire `cours_palm/session_1`
- Lancez l'interface graphique **PrePALM**



Interface graphique PrePALM

1.3 Création d'une branche de calcul

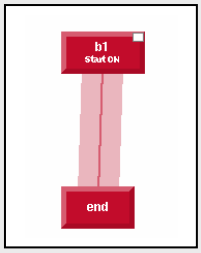
La première opération que nous allons réaliser avec PrePALM va consister à insérer une branche de calcul. Les branches servent à programmer le lancement des différentes unités de traitement de PALM. Nous reviendrons plus loin sur les unités PALM.
PALM est un coupleur dynamique, l'enchaînement des différentes actions élémentaires suit la logique d'un langage de programmation, avec des déclarations de variables, des instructions et des structures de contrôle (boucles et tests), tout ceci est défini dans les branches de calcul.

Au boulot !

- Sélectionnez la catégorie **Branches**
- Cliquez sur le bouton Insert

Une fenêtre apparaît

- Donnez un nom à votre branche, par exemple b1, validez
- Dans le canevas vous devez voir apparaître ceci :



Le rectangle supérieur matérialise le début de la branche, celui du bas la fin de la branche, le trait épais du milieu servira à matérialiser le cheminement entre les unités.

Faites-le !

- Double click sur un des deux rectangles
- Modifiez la couleur de la branche : on vous laisse deviner comment
- Click droit sur l'un des deux rectangles pour les déplacer individuellement
- Click droit sur le gros trait pour déplacer toute la branche
- Click dans le petit rectangle blanc pour fermer/ouvrir la branche

Important : si vous ne l'avez pas remarqué, une aide contextuelle sur les actions dans le canevas s'affiche tout en bas de l'interface graphique

- Déplacez la souris sur les différentes zones de PrePALM pour voir apparaître les messages d'aide

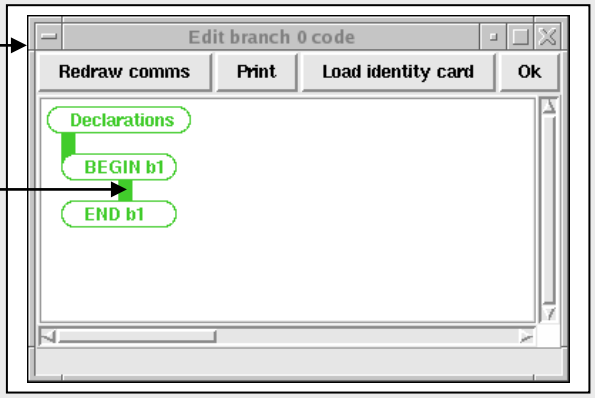
1.4 Edition du code de branche

Nous allons maintenant créer notre première application PALM, le traditionnel « bonjour », pour cela nous allons insérer une région Fortran dans le code de branche.

C'est parti !

- Double click sur trait épais de la branche

La fenêtre Edit branch code apparaît →



- Click ici →

- Sélection de

- Saisie de l'instruction F90

1.5 Préparation des options de compilations

PrePALM se charge lui-même de créer l'application PALM, y compris le fichier Makefile de l'application. Pour cela il a besoin de connaître certaines infortmations comme l'endroit où est installée la bibliothèque PALM, le nom des compilateurs F90, F77, C et C++, les options de compilation.

Pour des raisons de portabilité des applications, le Makefile inclut un fichier (**Make.include**) qui dépend de la machine sur laquelle on travaille. L'image de ce fichier Make.include peut être éditée dans l'interface graphique.

Faisons le !

- Menu Settings => Palm Makefile options edit
- Remplissez les options qui correspondent à votre environnement (compilateur, mpi, etc)
- Sauvegardez ces options dans un fichier .mak
- Click sur le bouton "save as default" pour que PrePALM les retienne la prochaine fois.

Il est temps de sauvegarder notre fichier **PrePALM**. Les fichiers PrePALM ont une extension en .ppl (comme **P**re**P**ALM **L**anguage).

Sauvegardez !

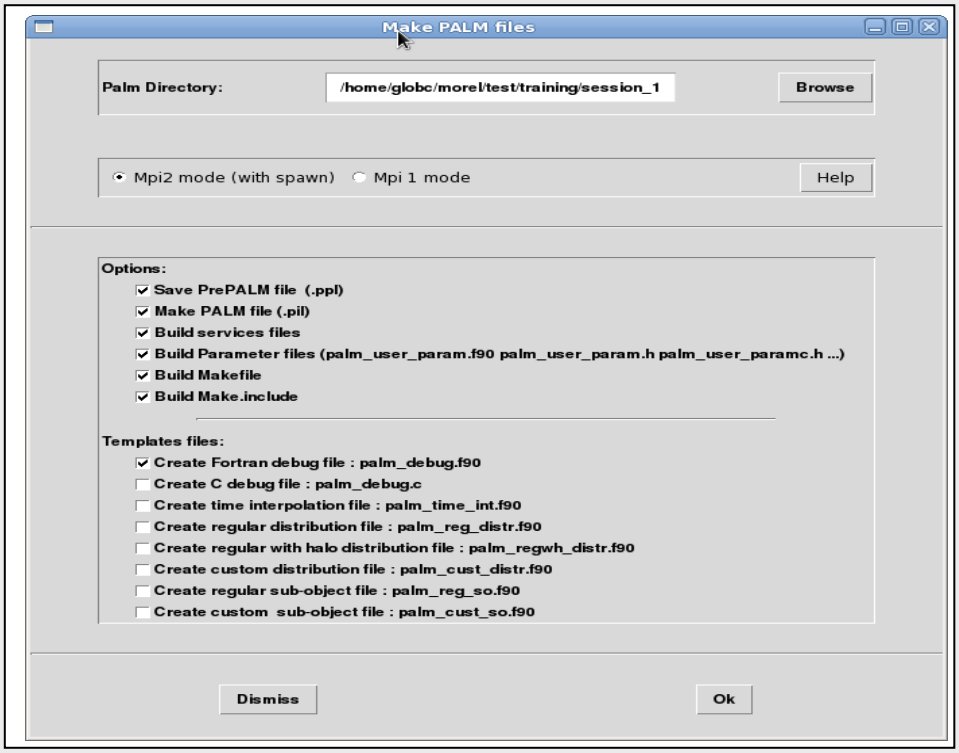
- Menu File => Save PrePALM file as (ppl format) ou icône
- Positionnez-vous dans le répertoire `training/session_1`
- Donnez un nom de fichier, session_1 par exemple

1.6 Création des fichiers nécessaires pour PALM

On peut maintenant créer les fichiers nécessaires à l'application PALM

Faites travailler PrePALM !

- Menu File => Make Palm files ou icône Cochez les cases comme ci-dessous :



- Click sur Ok

Arrêtons-nous un instant sur les fichiers crées, dans le répertoire **session_1**.
Une commande ls donne :

```
Makefile                    palm_init.c
Make.include                palm_user_paramc.h
palm_debug.f90              palm_user_param.f90
palm_driver_servicesc.c      palm_user_param.h
palm_driver_services.f90    session_1.pil
```

palm_entities_services.f90 session_1.ppl

En plus du fichier `session_1.ppl`, PrePALM a créé des fichiers de service pour PALM, décrivons en quelques-uns :

➥ **`Makefile`** va nous permettre de compiler l'application avec la commande make
➥ **`Make.include`** est un fichier inclus par Makefile qui contient tout ce qui est spécifique à notre machine, lorsque vous travaillerez sur une autre machine, il faudra modifier ce fichier.
➥ **`palm_init.c`, `palm_driver_servicesc.c`, `palm_driver_services.f90`, `palm_entities_services.f90`** et **`palm_debug.f90`** sont des fichiers de service nécessaires pour compiler l'application.
➥ **`palm_user_param.h`** et **`palm_user_paramc.h`** sont des fichiers qui peuvent contenir des constantes, leur utilité sera détaillée ultérieurement.
➥ **`session_1.pil`** est le fichier lu en entrée du coupleur, il contient les informations compilées du fichier .ppl pour être lisibles par le coupleur.

1.7 Compilation de l'application et exécution

L'interface graphique ne lance pas l'application PALM, celle-ci doit être compilée et exécutée sur la machine parallèle (ou non) de votre choix. Pour ces TP, où nous ne lancerons pas de grosses applications, il est plus facile d'utiliser la même machine (des PC sous Linux) pour faire tourner à la fois l'interface graphique et lancer les calculs.

Pour faire du calcul parallèle, PALM s'appuie sur la norme MPI2, différentes implémentations MPI respectent cette norme, sous Linux c'est le cas d'Openmpi de Mpich ou d'Intelmpi. Selon la librairie utilisée (et sa version) il peut être nécessaire, avant de lancer une application parallèle, de lancer un processus démon.

Y'a plus K!
- make
- mpd& (si nécessaire avec certaines version de mpich ou intelmmpi)
- mpirun –np 1 ./palm_main

Vous devriez alors voir s'afficher quelque chose comme « Bonjour » à l'écran.

Exercice 1
En cliquant sur le trait du code de branche, insérez une boucle (de 1 à 5) autour de l'affichage de bonjour et une condition à l'intérieur de la boucle pour faire apparaître ceci :
Bonjour 1
Bonjour 2
Troisieme bonjour
Bonjour 4
Bonjour 5
Remarque : les boucles et les conditions s'insèrent à partir du code de branche en cliquant sur le trait vertical. On déplace ensuite la fin de boucle en sélectionnant select to move puis insert.

1.8 Options de la commande PrePALM

La commande prepalm possède des arguments facultatifs, des options permettent de lancer l'interface graphique en mode compilation pour créer les fichiers de service. Pour avoir la syntaxe de la commande prepalm, il suffit d'entrer :

```
> prepalm --help
Usage : prepalm [--h] [--c] [--p] [--l file.f90] [--t] [filename.ppl]

Options
   --h,--help:               display this information

   --c,--compile:            compilation mode, make palm files
                             (.pil and service files)

   --p,--pil-only:           (implies --c) make only the .pil file

   --l,--load-constants-file file.f90:
                             replace the values of the constants
                             in the .ppl file with the values loaded from
                             file.f90 (F90 parameter declaration syntax)

   --m,--makefile-options-file file.mak:
                             replace the makefile commands and
                             options in the .ppl with the values loaded from
                             file.mak (F90 mak options file syntax)

   --np,--nb_procs:          enforces the max nb of processors


   --t,--trace-execution:    (implies -c) force trace execution mode in .pil
file
  --mpi1,--mpi1_mode:             (implies --c) forces the MPI1 mode (mutually exclusive
with --mpi2)
  --mpi2,--mpi2_mode:             (implies --c) forces the MPI2 mode (mutually exclusive
with --mpi1)
```

Pour lancer l'interface graphique et lire un fichier .ppl directement vous pouvez faire :
> **`prepalm nom_de_fichier.ppl`** : ouvre le fichier spécifié s'il existe, le crée s'il n'existe pas.

Dans certain cas il est intéressant de lancer PrePALM en mode compilation avec l'option --c pour générer le fichier .pil et/ou les fichiers de service. Ceci est utile dans ces situations :

- si vous avez plusieurs fichiers .ppl dans le même répertoire, correspondant à plusieurs configurations possibles de votre couplage,
- lorsque vous serez suffisamment expert pour éditer directement le fichier .ppl "à la main" pour faires quelques modifications,
- lorsque vous travaillez sur une machine dédiée au calcul sans retour X11, ou si vous voulez générer l'application dans un job batch.

L'activation du mode compilation se fait ainsi :
> **`prepalm non_de_fichier.ppl --c`** : ouvre le fichier spécifié et crée les fichiers de service.

L'option --l permet de remplacer la valeur des constantes de PrePALM par celles contenues dans le fichier spécifié. Les constantes de PrePALM permettent de paramétrer les applications, nous décrirons plus loin comment les utiliser.

L'option --t, permet d'avoir une trace de l'exécution de l'application, son utilisation sera décrite par la suite.

Dans tous les cas, la commande prepalm demande d'avoir un environnement graphique X11. Pour compiler une application sur une machine distante sans retour X, il faut utiliser la commande prepalm_tclsh.tcl située dans le même répertoire que le programme prepalm_MP.tcl. Cette commande demande de positionner la variable d'environnement PREPALMMPDIR (cf § 1.2). Les arguments de cette commande sont les mêmes que ceux de la commande prepalm.

1.9 Rappel des points de cette session

Vous avez appris comment lancer l'interface graphique PrePALM, comment créer et manipuler une branche de calcul pour décrire un algorithme. Vous avez aussi appris à créer, compiler et exécuter une application PALM dans un contexte MPI.

Vous avez compris que ce n'est pas l'interface graphique PrePALM qui exécute l'application, elle permet juste de saisir les structures de contrôle, le code des régions fortran et de préparer toute l'infrastructure pour compiler l'application. Une fois ce travail réalisé, des fichiers de service PALM sont générés par l'interface graphique lorsque l'utilisateur fait « make palm files ». Ce sont ces fichiers (C et Fortran) qui son compilés. Les erreurs de syntaxe éventuelles dans le Fortran des branches de calcul sont donc détectées au moment ou vous faite « make » pour compiler les fichiers de service de l'application. Une fois ces fichiers compilés, ils sont assemblés avec la bibliothèque du driver de PALM (fichier libdrv.a de la distribution PALM) pour constituer le programme exécutable palm_main qui est un exécutable MPI, processus maître de toute application PALM (on l'appelle aussi driver). C'est cet exécutable qui tourne sur la machine. Dans notre cas on le lance par la commande ./palm_main car on utilise lammpi. Il serait beaucoup plus correct de faire : mpirun -np 1 ./palm_main pour lancer ce programme.

En décrivant des structures de contrôle et des régions Fortran dans PrePALM, sous une forme qui ressemble à un sous programme Fortran (mais rappelons le qui n'en est pas un), vous décrivez en fait un arbre qui est interprété pas à pas par le driver de PALM. C'est le premier rôle du driver de PALM, interpréter les branches de calcul. Si vous aviez plusieurs branches, l'exécution se ferait en alternance d'une branche à l'autre. PALM joue un rôle de schéduleur pour interpréter en parallèle les différentes branches. A chaque itération du schéduleur, le driver de PALM examine toutes les branches pour regarder ce qu'il y a à exécuter, il choisit une des branches et y fait avancer l'algorithme. Dans les fichiers de service, chaque région Fortran donne lieu à une subroutine Fortran qui partage avec les autres subroutines de la même branche un module Fortran90 qui contient les variables déclarées au niveau de l'interface graphique. C'est par ce mécanisme que les différentes variables sont connues tout au long de la branche. Il est possible d'utiliser des structures de contrôles (boucles et conditions) dans les régions Fortran, mais il est essentiel de faire en sorte que ces structures soient bien « refermées », on ne peut pas par exemple commencer une boucle (DO …) dans une région Fortran et la terminer (END DO) dans une autre région Fortran car la subroutine générée par PrePALM pour cette région ne serait pas correcte. Il y a donc une grosse différence entre une boucle définie explicitement dans l'interface graphique (insert do loop) et une boucle écrite entièrement dans une région fortran.

Cette algorithmique mise à disposition de l'utilisateur va permettre de définir très facilement des algorithmes de couplage. Rappelons que le but n'est pas de faire les calculs dans les branches mais bien de disposer d'un langage de haut niveau pour enchainer en parallèle ou en

séquence des opérations élémentaires définies explicitement par les unités PALM que nous allons voir au chapitre suivant.

2 Session 2 : Lancement des unités

2.1 Introduction

Une unité PALM est une portion de code utilisateur que l'on peut démarrer par l'appel d'une fonction (C, C++) ou d'une subroutine Fortran sans argument (nous verrons plus loin qu'il est également possible de lancer des codes compilés ou des programmes réalisés dans des langages interprétés). Une unité doit être vue comme une tâche élémentaire que l'on peut organiser à l'intérieur d'un algorithme de couplage. La granularité des unités doit être déterminée en fonction des applications et du degré de modularité attendu. Pour du couplage de modèles, un grain très grossier est généralement suffisant, chaque unité peut correspondre à un code de calcul complet. Pour des applications d'assimilation de données où l'on recherche à organiser des opérateurs (modèle direct, adjoint, opérateur d'observation) pour constituer une ou plusieurs méthodes s'appuyant sur un algorithme d'assimilation (3DVAR, 4DVAR, …) un grain plus fin est généralement indispensable.

PALM permet d'assembler des unités développées en différents langages. Pour l'instant nos unités seront de simples sous-programmes indépendants sans communication. Le but est de montrer comment interfacer un code de calcul existant pour en faire une unité PALM.

2.2 Du programme indépendant à l'unité PALM

Pour coupler des codes de calcul, on part en général de codes existants, le point d'entrée de ces codes est l'instruction main (en C ou C++) ou PROGRAM (F90 et F77). La première chose à faire est de remplacer PROGRAM par SUBROUTINE (ou main par un autre nom de fonction). La seule contrainte est de disposer du code source du code à coupler ou au moins du code source du programme principal. Si on n'a pas accès au code source du programme principal, la situation n'est pas désespérée (bien que moins confortable) s'il est possible d'appeler des sous-programmes utilisateurs, fonctionnalité que proposent en général les codes industriels diffusés sans les sources, une session spécifique de ce manuel est consacrée au lancement de ce type de codes.

2.3 Un exemple d'unité PALM

Vous trouverez dans le répertoire session_2 quatre exemples basiques d'unités PALM écrites en C, C++, F77 et F90. Examinons dans un premier temps l'unité écrite en F77 :

```
C$PALM_UNIT -name unit77\
C           -functions {F77 unit77}\
C           -object_files {unit77.o}\
C           -comment {exemple en Fortan77}
      SUBROUTINE UNIT77
      INCLUDE "palmlib.h"
```

```
      WRITE(PL_OUT,*) 'UNIT77 : Bonjour'
      RETURN
      END
```

Les quatre premières lignes sont des commentaires, il n'y a rien d'original dans la suite du sous-programme si ce n'est le chargement du fichier d'entête palmlib.h et l'écriture dans l'unité logique PL_OUT. PL_OUT est tout simplement associé à un fichier de sortie de PALM qui peut être aussi utilisé pour suivre l'exécution des différentes unités (c'est l'insertion de palmlib.h qui donne accès à PL_OUT).

2.4 Les cartes d'identité

Revenons sur les quatre premières lignes, en fait tout est décrit ici pour que la subroutine unit77 soit reconnue comme unité PALM par l'interface graphique PrePALM. Ces lignes sont des commentaires pour pouvoir les insérer dans le code source de l'unité, ce qui est pratique pour la maintenance du code, mais rien n'oblige l'utilisateur à écrire ces lignes à cet endroit, elles pourraient être dans un fichier à part. Une aide complète pour l'écriture des cartes d'identité est accessible dans le menu Help de PrePALM.

Description des différents champs dans notre exemple :

-name unitf77 : donne un nom générique à l'unité, d'une manière générale les noms donnés dans PALM et PrePALM ne doivent jamais contenir d'espace ni de point.

-functions {f77 unit77} : permet de préciser quelle subroutine fortran77 doit être appelée, remarquez le f77 devant unit77, il précise le langage de programmation de unit77. Remarquez aussi les accolades, elles servent à décrire des listes, en effet il est possible de déclarer une unité qui appelle en séquence plusieurs fonctions.

-object_files {unit77.o} : permet de préciser, pour la compilation de l'application, le nom des fichiers .o (objets) dont l'unité a besoin. Si par exemple unit77 appelait un autre sous-programme qui ne soit pas dans le même fichier, il faudrait ajouter le fichier .o contenant cet autre sous-programme. A la compilation de l'application PALM, si le fichier .o n'existe pas ou n'est pas à jour suite à la modification du programme source, le Makefile de PALM cherchera à le générer en utilisant des règles par défaut de compilation. Dans notre exemple il créera donc unit77.o à partir de unit77.f

Remarque importante :
Si votre unité est par exemple un code de calcul appelant de nombreux sous-programmes écrits dans de nombreux fichiers, il n'est pas judicieux de décrire dans le champ object_files tous les .o des fichiers incriminés.
Dans ce cas on procède comme ceci :
On prépare à l'avance la compilation (sans édition de liens) des fichiers source, et, plutôt que de créer l'exécutable, on crée une bibliothèque (.a). C'est cette bibliothèque que l'on donne dans le champ object-files.

Voyez la différence!
- Dans le répertoire session_2, ouvrez les fichiers unit77.f unit90.f90 unitC.c unitCPP.C
- Notez la différence selon les langages de programmation

2.5 Chargement des cartes d'identité des unités

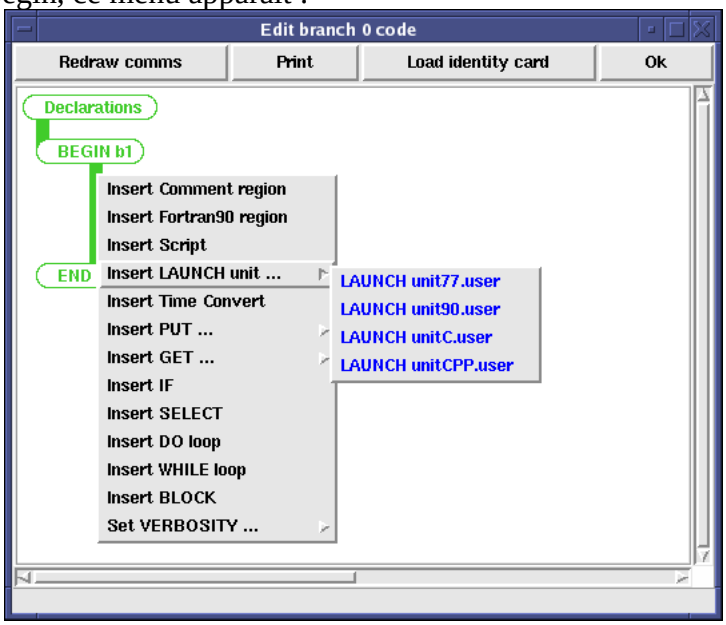
Avant de lancer les unités dans PrePALM, il est nécessaire de charger leurs cartes d'identité.

Chargez !

- Lancez PrePALM dans le répertoire session_2
- Menu File => Load identity card ou icône 
- Chargez les cartes d'identité des 4 unités unit77.f unit90.f90 unitC.c unitCPP.C

2.6 Lancement des unités

Le lancement des unités est fait au niveau du code de branche, en cliquant sur le trait continu après l'instruction begin, ce menu apparaît :



Lancez-vous !

- Créez une branche b1
- Editez le code de branche
- Lancez à la suite les 4 unités
- Sauvegardez votre application
- Créez les fichiers de service
- Vérifiez les ressources en processus dans Settings => Palm execution setting => Number of proc : **1**
- Compilez et exécutez : **mpirun -np 1 ./palm_main**

Le résultat de votre « calcul », soit les écritures dans le fichier PL_OUT, se trouve dans le fichier b1_000.log :

```
**********************************************************************
***************       output file for process of         **************
***************            branch=0 rank=0               **************
```

```
***************         running entity index=2           **************
*****************************************************************

UNIT77 : Bonjour
*****************************************************************
***************       output file for process of         **************
***************            branch=0 rank=0               **************
***************         running entity index=3           **************
*****************************************************************

UNIT90 : Bonjour
*****************************************************************
***************       output file for process of         **************
***************            branch=0 rank=0               **************
***************         running entity index=4           **************
*****************************************************************

UNIT C : Bonjour
*****************************************************************
***************       output file for process of         **************
***************            branch=0 rank=0               **************
***************         running entity index=5           **************
*****************************************************************

UNIT C++ : Bonjour
```

Si vous regardez d'un peu plus près dans le répertoire session_2, vous pouvez constater que maintenant 5 programmes exécutables indépendants ont été créés :

- main_unit77
- main_unit90
- main_unitC
- main_unitCPP
- palm_main

Vous trouverez aussi les programmes principaux source (main_unit*.c) qui ont été créés par PrePALM (fichier de service) Si vous ouvrez le fichier main_unit90.c vous obtenez ceci :

```
#include "palmlibc.h"

int C_MAIN_FOR_FORTRAN(int argc, char **argv, char **envp) {
      int il_err = 0;

      il_err = PALM_Init(argc, argv,"main_unit90");
      f2c_name(unit90)();
      il_err += PALM_Finalize();
      return (0);
}
```

L'appel à la subroutine Fortran unit90 a été encapsulé dans ce programme C, l'appel aux routines d'initialisation et de terminaison de PALM sont faites ici, ceci évite à l'utilisateur d'avoir à ajouter ces appels dans le programme source des unités. Notons que le nom de l'unité (ici "main_unit90") est déterminé dans l'interface graphique PrePALM. Si pour une quelconque raison l'utilisateur décide d'instrumenter son code en conservant son propre programme principal, il est impératif d'appeler la fonction PALM_Init avec le même nom que celui donné dans PrePALM, attention par exemple aux instances multiples d'un même code qui porteront forcément des noms différents.

Le f2c_name() est une macro de pré-compilation (définie dans PALM pour le pré-compilateur) destinée à convertir automatiquement le nom des fonctions pour assurer la compatibilité entre les langages Fortran, C ou C++. En effet même si votre unité est écrite en Fortran, le programme principal généré par PrePALM est écrit en langage C.

Une application PALM est une application MPMD (Multiple Program, Multiple Data) : plusieurs exécutables distincts peuvent discuter entre eux. Le lancement des unités est fait par le driver de PALM (palm_main). Comme les unités sont disposées sur une seule branche, PALM lance les exécutables les uns à la suite des autres.

Important : des commandes à retenir pour faire un peu de ménage avec le Makefile généré par PrePALM :

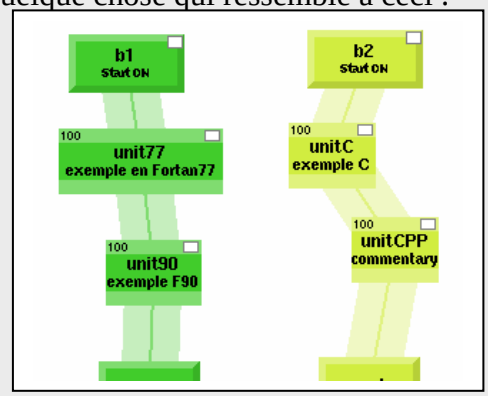
- **make tidy** : détruit uniquement les fichiers de sortie de PALM, ceci est nécessaire pour relancer une exécution car les fichiers de sortie sont ouvert en « append » (écriture en fin de fichier). Si on ne le fait pas on conserve les sorties des exécutions précédentes.
- **make clean** : détruit en plus les exécutables et les fichier .o
- **make allclean** : détruit en plus les fichiers de service créés par PrePALM, ne garde que ce qui est indispensable.

2.7 Calcul parallèle

Nous allons maintenant créer une application ou les unités vont tourner en parallèle. PALM gère deux niveaux de parallélisme, le premier est un parallélisme de tâche que l'on définit simplement dans l'interface graphique en créant plusieurs branches. Le second est un parallélisme interne aux unités, nous reviendrons plus tard sur ce second niveau de parallélisme.

Parallélisez !

- Créez une seconde branche
- Editez le code des deux branches en même temps
- Dans la première branche : click sur unit_C et select to move
- Dans la seconde branche : click sur move LAUNCH line here
- Idem pour unit_CPP, puis fermez les éditeurs de code de branche
- Arrangez un peu le canevas de PrePALM en déplaçant les unités : click gauche pour sélectionner une unité (elle devient rouge), puis click droit pour la déplacer. Vous devriez avoir quelque chose qui ressemble à ceci :



- Changez les ressources en processus dans Settings => Palm execution setting => Number of proc : **2**
- Sauvez, fichiers de service, make clean, make , **mpirun -np 1 ./palm_main**

Bravo ! Vous avez peut-être fait votre premier calcul parallèle sans le savoir, sans rien connaître à MPI, rien qu'en dessinant ! C'est une des caractéristiques de PALM, on peut faire du calcul parallèle sans compétences particulières dans ce domaine. Les résultats sont maintenant dans deux fichiers distincts : le fichier b1_000.log contient les sorties des unités de la branche b1 et le fichier b2_000.log ceux de la branche b2. Ouvrez ces fichiers pour voir les résultats.

Remarque :
Avec la commande mpirun -np 1 ./palm_main, vous n'avez lancé qu'un seul programme (palm_main que l'on nomme aussi driver de PALM) en fait c'est ce programme qui se charge de lancer les autres exécutable (main_unitC, main_unit77, main_unit90, main_unitCPP) en parallèle ou a la suite selon comment ils sont organisés dans les branches.

2.8 L'analyseur de performances

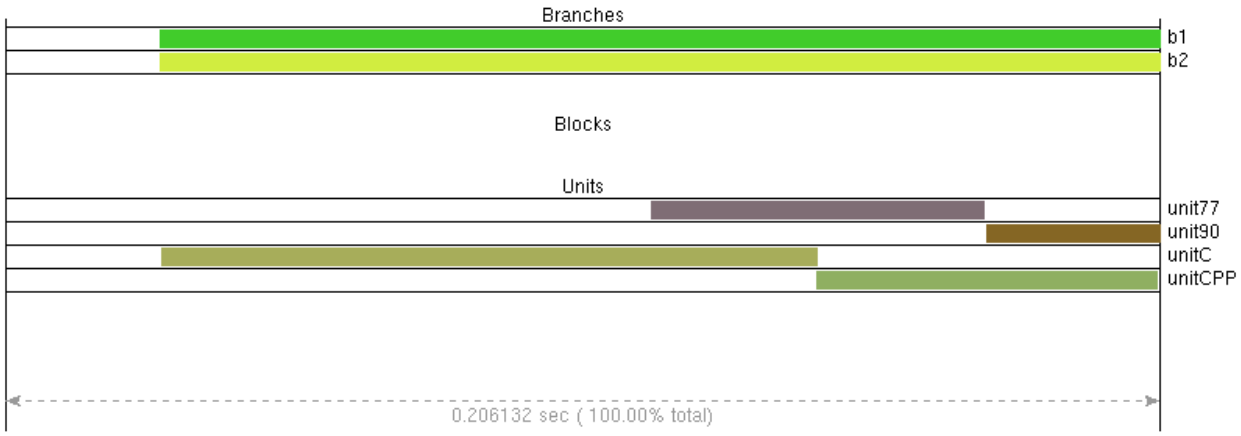
En lui précisant dans PrePALM, PALM est capable de générer des fichiers de trace permettant d'analyser les performances de l'application ou de « rejouer » l'exécution.

Analysez !

- Menu Settings => Palm execution setting, cochez cette case :

  ☐ **Trace execution for animation (file palmperf.log)**

- Fichiers de service ;make clean ;make ; ./palm_main
- Menu Analyse run => Load file : lire palmperf.log
- Un compte rendu de lecture vous donne le temps d'exécution par unité
- Menu Analyse run => Play : vous permet de voir graphiquement le lancement des unités, (on vous laisse trouver vous-même comment l'utiliser)
- Menu Analyse run => performace analyser : donne le tableau ci-après

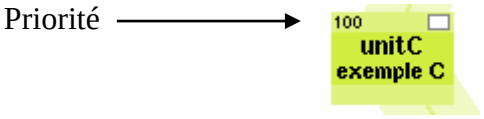


Sur cet exemple, on voit nettement que unit77 a tourné en même temps que unitC et unitCPP.

Gestion des processus (processeurs)

Comme on l'a vu, PALM gère les processus en fonction des unités qu'il doit lancer, en séquence à l'intérieur des branches, en parallèle entre les différentes branches. Dans tous les cas PALM se limitera au nombre de processus spécifié dans PrePALM, la gestion est dynamique à l'intérieur d'une enveloppe statique, ceci tout simplement pour réserver le bon nombre de processeurs dans un job batch et ne pas risquer d'écrouler une machine parallèle.

S'il n'a pas suffisamment de ressources, par exemple si on a deux branches à faire tourner sur un seul processeur, PALM est capable d'alterner l'exécution des différentes unités. Lorsque deux unités sont concurrentes, PALM regarde le niveau de priorité pour déterminer celle qu'il doit lancer en premier. Par défaut ce niveau de priorité est mis à 100 (il est affiché dans le coin supérieur gauche de la boite représentant l'unité), mais il peut être modifié par l'utilisateur : Un click gauche sur la priorité la décrémente, un click droit l'incrémente.



Exercice 2
Lancez l'application PALM à deux branches sur un seul processus, jouez sur les priorités pour faire tourner dans l'ordre :

- -unit77
- -unitC
- -unit90
- -unitCPP

Vérifiez avec le Play et l'analyseur de performance l'ordre de lancement des unités.

2.9 Rappel des points de cette session

Dans cette session vous avez appris à définir et à lancer des unités PALM écrites dans des langages de programmation différents, à créer les cartes d'identités nécessaires pour l'interface graphique PrePALM.
Ensuite vous avez appris comment lancer plusieurs branches concurrentes en parallèle, à utiliser l'analyseur de performance de PrePALM pour avoir un compte rendu graphique de l'exécution de l'application.
Vous savez maintenant comment gérer les ressources et jouer sur le nombre de processus de l'application pour l'optimiser en fonction de votre calculateur.

3 Session 3 : Les blocs

Pour l'instant nous avons vu que chaque unité PALM génère un exécutable indépendant. Pour deux raisons (partage de mémoire et optimisation du temps de lancement) il est intéressant de regrouper plusieurs unités en un seul programme exécutable.
Nous allons construire une nouvelle application avec le lancement d'une unité dans une boucle. Pour pouvoir paramétrer facilement le nombre de lancements de cette unité, nous allons faire appel aux constantes de PrePALM. Les constantes correspondent aux déclarations PARAMETER du langage Fortran, ou #DEFINE du C, ce sont des valeurs connues au moment de la compilation des sous-programmes et qui ne peuvent pas être modifiés au cours de l'exécution.

PrePALM propose un menu pour saisir des constantes (type, nom, valeur ou expression). Ces constantes peuvent être dépendantes les unes des autres sous forme d'une expression arithmétique.

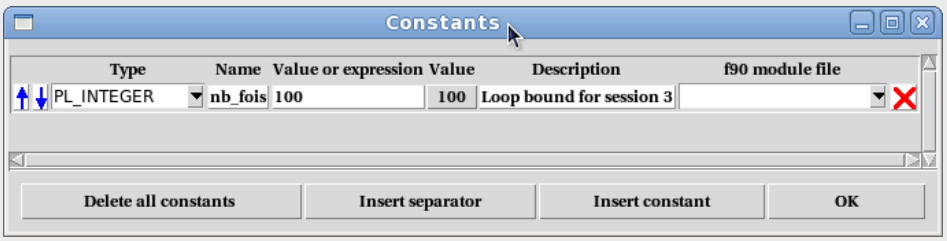
L'utilisation des constantes a un intérêt dans l'interface graphique elle-même, mais aussi dans les unités, pour cela PrePALM peut générer des fichiers utilisateurs à inclure dans les programmes source des unités selon le langage de programmation :

- palm_user_param.f90 F90 use palm_user_param !(module f90)
- palm_user_param.h F77 include 'palm_user_param.h'
- palm_user_paramc.h C,C++ #include ''palm_user_paramc.h''
- palm_user_param.py Python import palm_user_param

Ces fichiers contiennent tous les mêmes constantes, elles peuvent permettre par exemple de dimensionner des tableaux statiques.
En option, l'utilisateur peut indiquer un nom de fichier (module fortran90) s'il désire constituer des sous-ensembles de constantes. Pour plus de lisibilité, il est possible d'introduire des séparateurs dans la liste des constantes.
Dans notre application nous utiliserons une seule constante qui ne servira que dans l'interface graphique pour contrôler la borne supérieure de la boucle de lancement.

Soyez constants !

- Lancez PrePALM depuis le répertoire session_3
- Menu Constants => constants editor :



- Ajoutez une constante, changez son nom et donnez sa valeur (100)

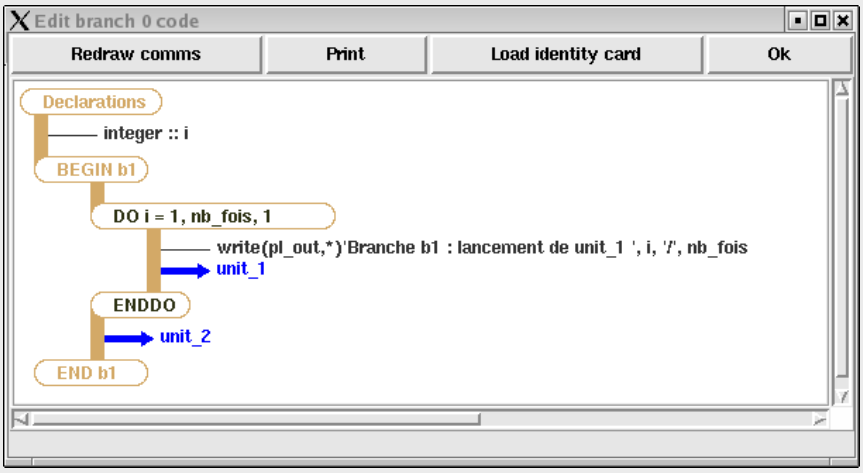
Pour cette application nous allons utiliser deux unités que vous trouverez dans unit_1.f90 et unit_2.f90. Voici le code source de unit_1 :

```
!PALM_UNIT -name unit_1\
!          -functions {F90 unit_1}\
!          -object_files {unit_1.o}\
!          -comment {exemple F90}

SUBROUTINE unit_1

  USE palmlib     !*I The PALM interface

  IMPLICIT NONE

  INTEGER :: iglobal
  COMMON /partage/ iglobal

  iglobal = iglobal + 1

  WRITE(PL_OUT,*) ' '
  WRITE(PL_OUT,*) 'UNIT_1 : Bonjour'
  WRITE(PL_OUT,*) 'UNIT_1 : valeur de iglobal: ', iglobal


END SUBROUTINE unit_1
```

Cette unité imprime la valeur d'une variable (iglobal) qui est incrémentée à chaque appel de la subroutine unit_1. Cette variable est déclarée comme une variable globale grâce à la directive Fortran COMMON. L'unité unit_2.f90 est identique à unit_1 sauf qu'elle n'incrémente pas la variable iglobal.

Ne bloquez pas !

- Avec le code de branche ci-dessous vous êtes capable de construire la première branche de l'application.



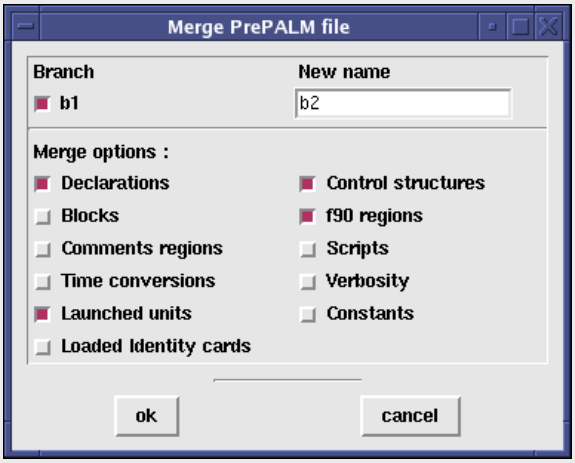
- Sauvegardez votre fichier

Nous allons maintenant construire une seconde branche. Comme elle ressemblera beaucoup à la première, nous allons utiliser la fonction **Merge** de PrePALM qui permet de récupérer une partie

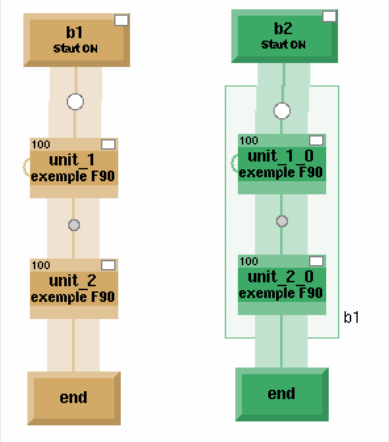
d'une application existante dans l'application courante. Dans notre cas, nous utiliserons le fichier que nous venons de sauvegarder comme application existante.

Copiez-vous !

- Menu File => Merge ppl file => fichier session_3.ppl



- Dans le canevas dédoublez les deux branches (click droit sur la branche pour la déplacer), pour plus de clarté, changez la couleur d'une des branches.
- Ouvrez la seconde branche et insérez un bloc avant la boucle.
- Déplacez la fin du bloc juste avant la fin de la branche
- Modifiez b1 par b2 dans la région fortran
- Votre application doit maintenant ressembler à ceci :



- Sauvegardez et exécutez l'application sur deux processus

Le bloc apparaît dans le canevas de PrePALM comme un rectangle grisé contenant les unités. Remarquez maintenant, qu'en plus du driver de PALM, l'application ne compte plus que 3 exécutables, les unités unit_1_0 et unit_2_0 ont été regroupées en un seul exécutable nommé main_block_1.

Conséquences du bloc :

1) Partage de la mémoire

Si on regarde le fichier b1_000.log, résultat de la branche b1, on remarque que la valeur de la variable globale iglobal vaut 1 à chaque fois que unit_1 a tourné, ce qui est normal car dans la boucle on lance à chaque fois un exécutable et que celui-ci meurt après chaque exécution, pour la même raison, la valeur de iglobal est 0 dans unit_2.

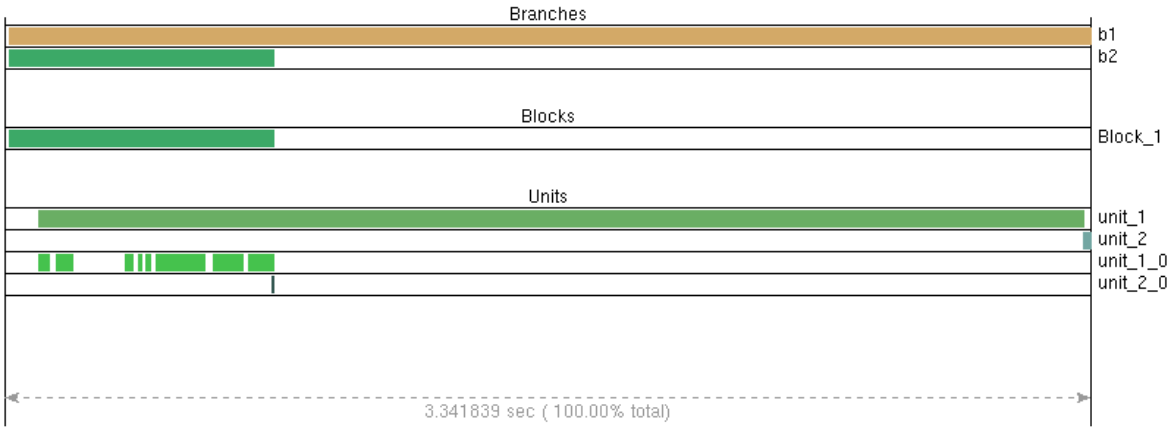
Par contre dans le fichier b2_000.log, résultat de la branche b2, la valeur de iglobal est conservée entre chaque lancement de l'unité unit_1_0. Ce qui est normal car maintenant, le lancement de l'unité correspond à l'appel d'une subroutine dans un programme principal qui effectue la boucle. De même, l'unité unit_2_0 connaît maintenant la valeur de iglobal calculée par unit_1_0.

Avec les blocs, on peut donc passer des informations entre des unités par des variables globales. Ce mécanisme n'est pas celui préconisé par l'approche PALM car il interdit d'avoir des unités parfaitement indépendantes (nous verrons plus loin la bonne manière de procéder), mais il peut être très utile pour séparer « fonctionnellement » des unités ou à des fins d'optimisation de l'application.

Partager la mémoire présente des avantages mais aussi des inconvénients, si vous avez deux unités indépendantes qui déclarent toutes les deux de gros tableaux statiques (ou dynamiques sans des-allocation) la mémoire de l'exécutable sous forme d'un bloc sera la somme de la mémoire des deux unités, ce qui peut être contraignant.

2) Optimisation du temps de calcul

Si vous ouvrez l'analyseur de performance de PrePALM sur cette application vous vous rendrez compte (selon la configuration de votre machine) que la branche b2 a tourné beaucoup plus vite que la branche b1 :



Ce résultat est tout à fait logique car pour la branche b1 on charge en mémoire 100 fois de suite, puis on exécute 100 fois unit_1, alors que pour la branche b2 on n'exécute qu'un seul programme qui contient une boucle interne.

L'utilisation ou non des blocs est donc affaire d'optimisation du temps de calcul et/ou de la mémoire de l'application. Heureusement, l'interface graphique est suffisamment souple pour tester facilement différentes configurations. On voit ici tout l'intérêt d'avoir défini les unités comme des subroutines et non comme des programmes. Ceci laisse toute la souplesse d'encapsuler différemment le code dans l'interface graphique.

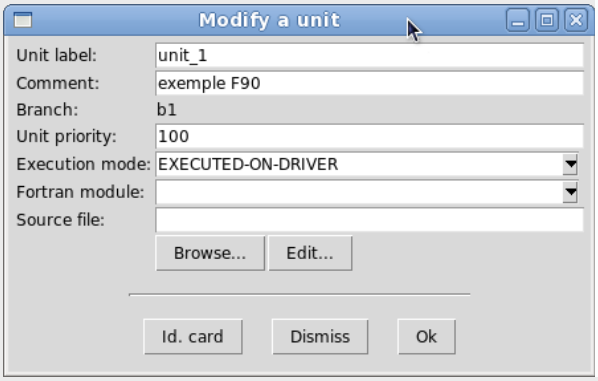
Remarque :
Certains codes de calcul (et ils sont nombreux, suite à notre retour d'expérience) ont bien du mal à tourner en boucle à l'intérieur d'un bloc, tout simplement parce qu'ils ne sont pas conçus pour cet usage : fichiers non fermés, variables non des-allouées, initialisations réalisées une seule fois… Dans ce cas, on évitera simplement d'utiliser les blocs, ce qui n'empêche pas d'exécuter le code de calcul dans une boucle.

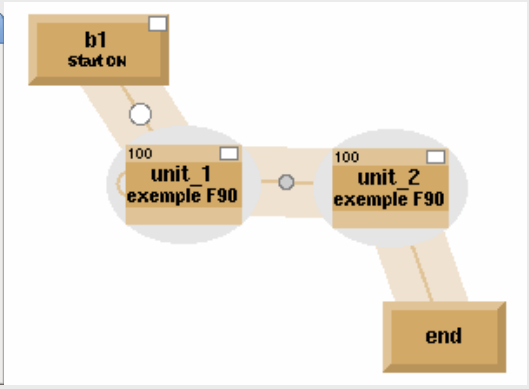
3.1 Lancement des unités dans l'exécutable du driver

Pour permettre encore plus de souplesse et éventuellement éviter un gaspillage de ressource, il est possible d'exécuter les unités non pas comme des processus indépendants mais d'appeler directement la routine dans le programme du driver de PALM (palm_main). Le driver de PALM étant mono-processus seules les unités non parallèles peuvent être exécutées de la sorte.

Assemblez dans le driver !

- Partez du fichier .ppl précédent
- Détruisez la seconde branche
- Editer les **deux** unités et sélectionnez «EXECUTED-ON-DRIVER» pour le mode d'exécution.





- Dans settings -> palm execution settings mettez 0 pour le nombre de processus
- Refaites tous les fichiers de service et testez l'application

Remarquez que les unités exécutées dans le driver de PALM apparaissent avec un cerclage grisé dans le canevas de PrePALM. Le résultat de l'exécution (écritures dans le fichier PL_OUT) est maintenant disponible dans le fichier palmdriver.log. Remarquez que l'application se comporte comme si les unités avaient été assemblées dans un bloc car dans notre cas elles se retrouvent toutes dans le même exécutable (palm_main).

Cette possibilité de faire tourner les unités dans le driver de PALM est très intéressante si vous n'avez pas d'unités parallèles (ou de machine parallèle), vous pouvez par exemple bâtir une application PALM mono-processeur, sans faire de calcul parallèle. Vous gardez toute la souplesse et la modularité de PALM avec la possibilité de décrire plusieurs branches de calcul, d'assembler des unités dans différents langages de programmation, etc. Tant que vos unités ne communiquent pas (nous verrons plus loin comment s'échanger des données entre les unités) il n'y a pas de possibilité de blocage de l'application.

Cette fonctionnalité est également très intéressante dans des applications parallèles, elle évite bien souvent le recourt à un processus supplémentaire, mais elle peut parfois conduire à des blocages de l'application si celle-ci n'est pas synchronisée correctement. En effet, le driver de PALM assure deux fonctions essentielles, lancer les unités selon l'algorithme de couplage décrit dans l'interface graphique, et répondre à des requêtes de ces dernières (principalement pour les communications). Si le driver de PALM est en train de faire tourner une unité, il ne peut plus temporairement continuer à assurer ces deux fonctions, donc lancer une unité dans le driver de PALM n'est pas une opération anodine.

Si vous ne disposez pas d'une machine parallèle, ou que votre application ne se prête pas à du calcul parallèle, il est également possible d'installer une version de PALM monoprocesseur où toutes les unités tournent dans le driver de PALM. Pour cela il suffit de préciser --without-mpi à la configuration de l'installation de PALM.

3.2 Arguments des exécutables lancés par PALM

Les programmes indépendants lancés par PALM sont construits à partir des fonctions (ou subroutines) sans arguments définies par l'utilisateur, leur nom est de la forme main_XXX ou main_block_NN selon qu'ils apparaissent ou non dans des blocks. Par défaut, PALM utilise la fonction Spawn de la norme MPI_2 pour lancer ces programmes.

Si votre unité PALM est issue d'un code de calcul prenant en entrée des arguments, et que, pour diverses raisons vous voulez continuer à utiliser ce mode de fonctionnement, sachez que PALM permet le passage d'arguments au lancement de votre unité ou block. Pour plus de souplesse, la définitions des arguments à passer à votre programme ne se définit pas dans l'interface graphique PrePALM mais par un mécanisme à partir de fichiers.

Si, au moment du lancement de l'exécutable **main_XXX**, PALM trouve dans le répertoire courant un fichier nommé **main_XXX.args**, il lancera ce programme avec les arguments décrits dans ce fichier.

Dans le répertoire session3/arguments, vous trouverez un exemple complet illustrant ce mécanisme avec les différents langages de programmation C, C++, F77 et F90. Le cas du block est également traité, dans ce cas les arguments sont passés à toutes les unités lancées dans le block.

Dans l'exemple le fichier contenant les arguments sont directement créés dans les branches de calcul de PrePALM par des instructions Fortran ou des commandes sh, mais tout autre mécanisme peut être imaginé pour créer ces fichiers.

Dans les fichiers sources C des unités, il convient de définir les arguments classiques en entrée de la fonction principale de l'unité (int argc, char **argv). Pour les programmes Fortran, qui a priori n'acceptent pas d'arguments au lancement, on utilise un mécanisme non standard basé sur l'appel de fonctions qui peuvent retourner ces informations (iargc() pour retourner le nombre d'arguments, getarg(n, arg) pour retourner la chaine de caractère du nième argument).

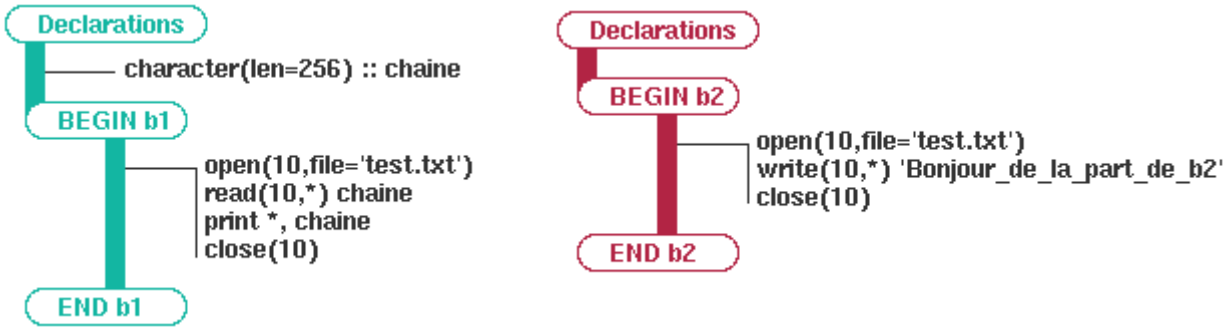
3.3 Rappel des points de cette session

Dans cette session vous avez regroupé plusieurs unités PALM pour construire un seul exécutable appelé block pour optimiser l'application. Vous avez appris à manipuler les constantes de PrePALM, à récupérer dans l'interface graphique une partie d'une application existante (Merge ppl file). Enfin vous avez appris comment utiliser le processus du driver de PALM pour exécuter des unités monoproc et comment passer des arguments aux exécutables indépendants lancés par PALM.

4 Session 4 : Plus loin dans les branches et les unités

4.1 Lancement par une autre branche

Jusqu'à présent, toutes les branches que nous avons définies démarraient au début de l'application car elles avaient comme attribut « start On ». Il est possible de retarder leur exécution en les faisant démarrer par d'autres branches. Dans l'exemple ci dessous, l'application se "plante" car lorsque la branche b1 cherche à lire des données dans le fichier test.txt celui-ci n'a pas encore été créé par la branche b2 dont c'est justement le rôle. Il y a un problème de synchronisation de l'application.



Exercice 3
Lire le fichier ne_marche_pas.ppl dans le répertoire session_4
Vérifiez qu'il ne fonctionne pas.
Passez la branche b1 à start off
Lancez b1 depuis b2 pour que l'application tourne correctement
Sauvez sous un autre nom et vérifiez

4.2 Les steps

Une autre manière de synchroniser une application consiste à utiliser des barrières sur les branches. En calcul parallèle, les barrières sont des synchronisations des processus, on peut voir cela comme un point de rendez-vous, tous les programmes s'attendent au même endroit tant que les autres n'y sont pas arrivés. Ces barrières sont associées aux **steps** de PrePALM. Pour utiliser un step il est nécessaire de le créer dans l'interface graphique. Un step peut ou non avoir l'attribut BARRIER. S'il a cet attribut, l'application se synchronisera (fera attendre les processus) sur tous les appels au même step.

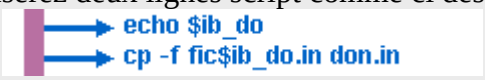
Exercice 4
Ouvrez le fichier ne_marche_pas.ppl dans le répertoire session_4.
Créez un step : sélectionnez la catégorie step, puis bouton insert
Mettez ce step à PL_BARRIER_ON
Dans les deux codes de branches appelez le step (insert step) pour synchroniser la fin de l'écriture du fichier test.txt avec le début de la lecture
Sauvegardez sous un autre nom et testez

4.3 Les scripts

Imaginons une application qui lance un code de calcul en boucle avec plusieurs jeux de données, par exemple pour faire une étude paramétrique. En général les codes de calcul lisent leurs données dans un fichier qui porte un nom particulier, dans notre cas ce fichier s'appelle don.in. Pour faire tourner le code de calcul sur des cas différents, nous sommes amenés à copier, avant le lancement, nos différents jeux de données dans le fichier don.in. Ici nos différents fichiers s'appellent fic1.in, fic2.in, fic3.in … La copie de l'un de ces fichiers peut se faire par une simple commande unix du genre cp –f fic3.in don.in. Dans l'interface graphique, il est possible de lancer des commandes unix ou de lancer des scripts pour faire ce genre d'opérations au moment opportun. Il est même possible, dans ces lignes de commande, de faire référence aux variables du code de branche.

Ecrivez !

- Toujours dans session_4, partez d'un fichier PrePALM vierge (menu File => New file)
- Lisez la carte d'identité de l'unité code dans le fichier code.f90
- Construisez une branche avec une boucle d'indice ib_do variant de 1 à 5
- Lancez l'unité code dans la boucle.
- Avant le lancement insérez deux lignes script comme ci dessous



- testez votre application

Le programme "code" ne fait qu'afficher à l'écran le contenu du fichier don.in, le résultat doit être celui-là :

```
1
 premier_jeu_de_donnees
2
 second
3
 troisieme
4
 quatrieme
5
 cinquieme
```

Exercice 5
Essayez de faire tourner la boucle dans un bloc
Que se passe-t-il ?
Modifiez le code de code.f90 pour que cela fonctionne correctement

4.4 Lancement d'une unité parallèle MPI

Pour ceux qui ne connaissent pas le calcul parallèle, et pour faire simple, ce type de parallélisme (MPI), consiste à diviser un problème en un certain nombre de processus. Chaque processus est une instance du même programme exécutable qui est répliqué au lancement. Par l'appel d'une primitive MPI, chaque processus peut savoir quel numéro d'ordre il est par rapport au nombre total de processus, il peut ainsi se spécialiser pour effectuer une partie du calcul. Une fois lancé, chaque processus vit sa vie et gère ses propres données (variables, tableaux). MPI, qui est un standard, fournit à la fois la manière d'exécuter les processus et le moyen de les faire discuter entre eux à partir de primitives dédiées aux communications entre les processus. Ce type de parallélisme est particulièrement efficace (lorsqu'il est bien codé), et tire partie au maximum des machines à mémoire distribuée où chaque processus est affecté à un processeur.

PALM gère deux niveaux de parallélisme, le premier niveau de parallélisme, que nous avons déjà abordé, est un parallélisme de tâche pris en charge par les branches de calcul, le second niveau est un parallélisme interne aux unités. Tout programme parallèle peut devenir une unité PALM parallèle. Pour illustrer le lancement d'unités parallèles dans PALM nous allons partir d'un exemple de code parallèle que nous allons PALMer.

Cet exemple est un de ceux donnés dans l'installation de LAMMPI, il effectue le calcul de Pi sur plusieurs processus MPI.

Palmez !

- Copiez le fichier fpi.f dans pi_mpi.f, éditez ce fichier
- Ajoutez la carte d'identité suivante :

```
C$PALM_UNIT -name pi_mpi\
C           -functions {F77 pi_mpi}\
C           -object_files {pi_mpi.o}\
C           -parallel mpi\
C           -minproc 2\
C           -maxproc 64\
C           -comment {calcul de pi}
```

- Remplacez program main par subroutine pi_mpi()
- Ajoutez include 'palmlib.h'
- Commentez les lignes suivantes :
  -l'appel à mpi_init
  -l'appel à mpi_finalize
  -la ligne stop
- Remplacez partout dans le programme MPI_COMM_WORLD par PL_COMM_EXEC
- Sauvez votre fichier
- Partez d'un fichier PrePALM vierge et lancez l'unité pi_mpi
- Donnez un nombre de proc compris entre 2 et 64
- Dans Palm execution settings, donnez un nombre de proc en conséquence
- Testez votre application

Tout nouvel utilisateur de PALM peut s'inspirer de ce petit exemple pour adapter n'importe quel code de calcul à PALM. L'écriture de la carte d'identité ne pose pas de problème particulier. Si le programme est éclaté en plusieurs fichiers et a son propre Makefile, on aura cependant tout intérêt à conserver ce Makefile mais au lieu de créer un exécutable on créera plutôt une bibliothèque (fichier .a) que l'on indiquera dans le champ object_files de la carte d'identité.
La recherche du programme principal (main en c ou program en fortran) et son remplacement par un nom de fonction ou de subroutine est très simple à réaliser. Les appels aux primitives mpi_init et mpi_finalize, qu'il faut éliminer, sont généralement fait une seule fois par le programme principal, respectivement au début et à la fin de celui-ci, ces appels sont donc faciles à localiser. Le remplacement du communicateur par défaut MPI_COMM_WORLD par PL_COMM_EXEC peut être plus problématique car on peut le rencontrer dans de nombreux fichiers et un simple oubli peut se révéler catastrophique, il est donc recommandé de se fabriquer une moulinette permettant de le faire automatiquement, ce qui est plus sûr.

4.5 Lancement d'une unité parallèle Open-MP

Pour ceux qui ne connaissent pas, et encore pour essayer de faire simple, Open-MP permet de paralléliser un code de calcul avec une toute autre approche que MPI. Ici, un seul exécutable est lancé. À l'aide de directives destinées au compilateur (et en activant l'option adéquate pour ce dernier), l'utilisateur précise dans le code source les régions où un calcul parallèle est possible : par exemple au début d'une boucle faisant des calculs sur un tableau. À l'exécution, le programme

lance des taches « légères » pour faire travailler en parallèle plusieurs processeurs qui se partagent les calculs en accédant en mémoire aux mêmes données. Ce type de parallélisme est souvent plus facile à mettre en œuvre que MPI, moins intrusif dans le code de calcul, mais il ne permet pas d'augmenter la taille du problème traité au-delà de la mémoire d'un nœud du calculateur ni de gérer efficacement un nombre important de processeurs. Il est cependant particulièrement adapté sur des machines parallèles à mémoire partagée.

Palm permet de lancer également des programmes parallélisés avec Open_MP et de leur affecter dynamiquement le nombre processeurs (threads), il suffit juste de définir le type de parallélisme dans la carte d'identité de l'unité, de compiler l'unité avec les bonnes options (-mp dans notre cas), et d'indiquer le nombre de processus à l'unité dans PrePALM.

Exercice 6
Faites tourner l'unité prodmv_omp.f90 sur quatre processus.

Remarque :
Selon les machines, il se peut que le programme se bloque, dans ce cas il faut agir sur le stack avec la commande :

```
> limit stacksize unlimited
```

Le nombre de processeurs que l'on demande peut aussi être plus grand que le nombre de processeurs de la machine, dans ce cas open_MP signale un warning, ce qui n'empêche pas l'application de fonctionner. Vous devez avoir un résultat de ce type :

```
Warning: omp_set_num_threads (4) greater than available cpus (2)
 Rang :            0 ; Temps :   0.1730000
 Rang :            2 ; Temps :   0.1370000
 Rang :            3 ; Temps :   9.2000000E-02
 Rang :            1 ; Temps :   0.2470000
```

Si par la suite vous avez à faire tourner un code de calcul parallélisé avec Open_MP vous pourrez vous inspirer de cet exemple. La démarche est toujours la même : rechercher le point d'entrée du code de calcul (main en c ou program en fortran), le remplacer par un nom de sous-programme, définir la carte d'identité, créer une bibliothèque plutôt qu'un exécutable.

Pour toute nouvelle unité PALM issue d'un code de calcul, avant de vouloir la faire communiquer avec d'autres unités par PALM (avec des communications PALM que nous allons voir au chapitre suivant), il faut déjà s'assurer que la mécanique qui permet d'exécuter le code de calcul sous forme d'une unité PALM est en place.

4.6 Rappel des points de cette session

Dans cette session vous avez appris qu'une branche de calcul peut être lancé par une autre. Comment synchroniser une application avec une barrière associé à un évènement (step). Ensuite vous avez vu comment adapter un code de calcul basé su MPI ou Open-MP pour en faire une unité PALM.

5 Session 5 : Les communications

5.1 Introduction

Nous avons vu jusqu'à maintenant toutes les possibilités de PALM au niveau de la gestion des processus. Pour faire du vrai couplage, il est temps maintenant de faire discuter les différentes unités entre elles !

Que ce soit pour un code complet ou pour une routine très simple, la première question à se poser est de savoir quoi s'échanger : quelles sont les données de couplage utiles pour construire une application couplée ? Seul l'utilisateur, ou le groupe d'utilisateurs, peuvent répondre à cette question. Pour faire du couplage entre un modèle d'océan et un modèle d'atmosphère on sera amené à envoyer des champs de température à l'interface des deux modèles. Pour coupler un modèle d'écoulement de fluide avec un outil de maillage on sera plutôt amené à s'échanger des contraintes sur une structure et des maillages. Dans le code informatique, ces grandeurs ou ces champs physiques sont tous contenus dans des variables, qui ont un type (entier, réel, structure de données, …) et une taille (tableaux 1d, 2d,…) Pour rester un outil générique, PALM ne donne aucune contrainte quant à la nature, au type et au format des données à échanger.

Dans PALM, on appelle **objet** les données à s'échanger, et **espace** la description informatique de ces objets. Plusieurs objets peuvent avoir le même espace. Dans l'interface graphique comme dans les unités, les objets et les espaces vont être caractérisés par un nom. Pour avoir un maximum de souplesse pour bâtir des applications PALM, l'échange d'information va se faire en deux étapes qui permettent une indépendance totale entre les unités.

Une unité (le code source), ne dira jamais à qui il va envoyer un objet, elle sera cependant obligée de faire savoir, à un moment donné du programme, que cette donnée est calculée et prête à être envoyée. Pour cela il faudra insérer dans le code source un appel à une primitive PALM, nommée PALM_Put.

Une unité ne dira pas non plus de qui elle doit recevoir ses données, mais simplement qu'elle a besoin d'informations et dans quelle variable informatique les données doivent être récupérées. Pour cela il faudra insérer dans le code source de l'unité un appel à la primitive PALM_Get.

La « vraie » mise en relation des entrées et des sorties se fait tout simplement dans l'interface graphique en reliant deux plots représentant graphiquement l'appel aux primitives PALM_Put et PALM_Get effectués dans le code. Cet appel doit être répertorié au préalable dans la carte d'identité de l'unité.

Lorsque l'on couple des codes de calcul, on fait souvent appel à des processus itératifs, les objets que l'on veut s'échanger reflètent par exemple l'évolution temporelle d'une grandeur physique. Dans PALM, il est prévu de pouvoir différencier deux instances temporelles d'un même objet pour leur réserver un traitement différent le cas échéant. Lors de l'appel aux primitives Put/Get (dans le code source) on précisera donc si besoin un champ « time » permettant de connaître l'instant auquel l'objet est associé. Pour PALM, ce champ time n'est autre qu'un entier, l'utilisateur est libre de relier cet entier à une date physique s'il le désire (cf chapitre 20.2 pour la conversion de date en entier) ou d'ignorer cet attribut en utilisant une constante prédéfinie PL_NO_TIME si l'objet n'a pas d'instances temporelles.

5.2 Préparation des unités, appel aux primitives PALM

Le plus simple pour comprendre est de partir d'un exemple, dans le répertoire session_5 ouvrons le fichier producteur.f90. Cette unité produit une matrice carrée de taille IP_SIZE*IP_SIZE et un vecteur de taille IP_SIZE. Examinons en détail sa carte d'identité :

```
 1    !PALM_UNIT -name producteur\
 2    !          -functions {F90 producteur}\
 3    !          -object_files {producteur.o}\
 4    !          -comment {producteur}
 5    !
 6    !PALM_SPACE -name mat2d\
 7    !           -shape (IP_SIZE, IP_SIZE)\
 8    !           -element_size PL_DOUBLE_PRECISION\
 9    !           -comment {tableau 2d double precision}
10    !
11    !PALM_SPACE -name vect1d\
12    !           -shape (IP_SIZE)\
13    !           -element_size PL_DOUBLE_PRECISION\
14    !           -comment {tableau 1d double precision}
15    !
16    !PALM_OBJECT -name ref_time\
17    !            -space one_integer\
18    !            -intent IN\
19    !            -comment {Temps auquel le vecteur est produit}
20    !
21    !PALM_OBJECT -name matrice\
22    !            -space mat2d\
23    !            -intent OUT\
24    !            -comment {matrice 2d}
25    !
26    !PALM_OBJECT -name vecteur\
27    !            -space vect1d\
28    !            -time ON\
29    !            -intent OUT\
30    !            -comment {vecteur 1d}
```

1 : En plus du mot clé PALM_UNIT que vous connaissez déjà, vient s'ajouter des mots clés PALM_SPACE (6,11) et PALM_OBJECT (16,21,26).

6 : Le premier PALM_SPACE, permet de définir un tableau à deux dimensions, champ shape (7). Ces deux dimensions sont décrites avec un paramètre (IP_SIZE). IP_SIZE est une constante qui devra être définie dans l'interface graphique PrePALM. On définit ensuite la taille de chaque élément du tableau ; comme cette taille peut être dépendante des options de compilation de la bibliothèque palm, elle est donnée avec des mots spécifiques pour PALM. Cette taille prendra directement la bonne valeur selon la bibliothèque PALM utilisée, par exemple pour la promotion automatique on non des réels simple précision en double précision dépendante des options de compilation (-r4 ou –r8 en général).

11 : Le second PALM_SPACE permet de définir un tableau à une seule dimension. Remarquez les parenthèses dans la définition du champ -shape, il ne faut jamais les oublier.

16 : Le premier PALM_OBJECT permet de définir un objet en entrée de l'unité (intent IN). Il pointe sur un espace qui n'a pas été défini (one_integer). En fait, un certain nombre d'espaces sont prédéfinis, comme one_integer, one_real, one_double, one_complex, one_string (chaîne de 256 caractères) ou encore one_logical, il faut se rappeler que ces objets prédéfinis existent pour ne pas alourdir inutilement les cartes d'identité.

Les autres objets (21,26) sont décrits en sortie (intent OUT). Ils correspondent respectivement à un vecteur et à une matrice qui vont être produits par l'unité. Remarquez que pour le vecteur, il est précisé que son champ time (time ON) sera utilisé ; l'appel à la primitive PALM_Put se fera donc

avec un temps différent de PL_NO_TIME et PALM sera capable de gérer indépendamment les différentes instances de cet objet.
Examinons maintenant le code source de cette unité. Dans notre exemple tous les appels aux primitives PALM sont dans le sous programme de l'unité. Bien entendu, pour de vrais codes de calcul, rien n'empêche d'appeler ces primitives PALM dans n'importe quel sous-programme. Notons que l'ordre dans lequel les primitives Put/Get sont appelées est sans importance pour le bon fonctionnement du couplage, il a cependant une incidence sur les performances de l'application.

```
33    SUBROUTINE producteur
34
35      USE palmlib             ! interface PALM
36      USE palm_user_param     ! constantes de PrePALM
37
38      IMPLICIT NONE
39
40      CHARACTER(LEN=PL_LNAME) :: cl_object, cl_space
41
42      DOUBLE PRECISION :: dla_vect(IP_SIZE), dla_mat(IP_SIZE,IP_SIZE)
43      integer :: il_vect_time, i, il_err
44
45    ! initialisation de dla_mat : matrice diagonale 1, 2, 3 ...
46      dla_mat = 0.d0
47      DO i = 1 , IP_SIZE
48        dla_mat(i,i) = i
49      ENDDO
50
51    ! envoi de la matrice
52      cl_space  = 'mat2d'
53      cl_object = 'matrice'
54      CALL PALM_Put(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, dla_mat,
il_err)
55
56
57    ! appel de PALM_get pour connaitre le temps auquel on doit produire le
vecteur
58
59      cl_space  = 'one_integer'
60      cl_object = 'ref_time'
61      CALL PALM_Get(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, il_vect_time,
il_err)
62
63      IF (il_err.ne.0) THEN
64          WRITE(PL_OUT, *) 'Producteur: le temps n''a pas ete recu, c''est
grave ...'
65          CALL PALM_Abort(il_err)
66      ENDIF
67
68    ! initialisation du vecteur : (1,2,3,...)*il_vect_time (pourquoi pas?)
69      DO i = 1 , IP_SIZE
70        dla_vect(i) = i * il_vect_time
71      ENDDO
72
73    ! envoi du vecteur
74      cl_space  = 'vect1d'
75      cl_object = 'vecteur'
76      CALL PALM_Put(cl_space, cl_object, il_vect_time, PL_NO_TAG, dla_vect,
il_err)
77
78
79    END SUBROUTINE producteur
```

36 : La première chose à remarquer est l'utilisation du module palm_user_param, ce module est celui crée par PrePALM à partir des constantes, il permet aux unités d'avoir accès aux paramètres définis dans l'interface graphique. Dans notre cas on peut donc utiliser la constante IP_SIZE (la même que pour la définition de l'espace) pour dimensionner les tableaux (42). Ceci pourra nous permettre de changer facilement la taille du vecteur et de la matrice sans modifier le code source des unités, uniquement en recompilant l'application.
40 : Les chaînes de caractères cl_object et cl_space vont contenir les noms des espaces et des objets, elles sont déclarées de taille PL_LNAME (longueur d'une chaîne de ce type pour PALM). Cette constante, comme beaucoup d'autres, est déclarée dans le module palmlib (bibliothèque PALM) dont on a fait un use (35).
54 : On appelle la primitive PALM_Put pour envoyer la matrice. On serait tenté d'écrire directement `CALL PALM_Put('mat2d', 'matrice' , PL_NO_TIME, PL_NO_TAG, dla_mat, il_err)` avec le nom de l'espace et de l'objet directement en argument, mais avec certains compilateurs, le Fortran a la mauvaise habitude de mal gérer les chaînes de caractère passées en argument ; on passe donc par les variables intermédiaires cl_space et cl_object qui ont été définies à une longueur adéquate pour PALM (PL_LNAME).
61 : On fait un appel à PALM_Get pour connaître le temps auquel on doit envoyer l'objet vecteur, ceci nous permettra d'illustrer les objets utilisant l'attribut time.
63 à 66 : Toutes les primitives PALM retournent un code d'erreur, une erreur différente de 0 indique un problème et donc que la variable retournée par le PALM_Get n'a pas été mise à jour correctement. Notre unité est faite pour produire un objet vecteur au temps qu'elle reçoit au préalable, si elle n'a pas ce temps, elle ne peut pas travailler, donc on arrête l'application par un appel à PALM_Abort. Notons au passage que c'est le seul moyen d'arrêter proprement une application PALM, les appels à STOP, call EXIT, CALL MPI_ABORT sont à bannir car ils n'informent pas le driver de PALM.
76 : Le vecteur est envoyé au temps il_vect_time.

Exercice 7

Ouvrez le fichier vecteur_print.f90 du répertoire session_5. La carte d'identité n'est pas complète, avant de la remplir, répondez aux questions en regardant le code Fortran.
Questions :

- Quelle est l'instruction du programme qui nous permet de dimensionner le tableau dla_vect à IP_SIZE ?
- Combien y'a-t-il d'objets IN ?
- D'objets OUT ?
- D'espaces utilisés ?
- D'espaces à déclarer dans la carte d'identité ?
- Que se passe t-il si le temps n'est pas reçu ?

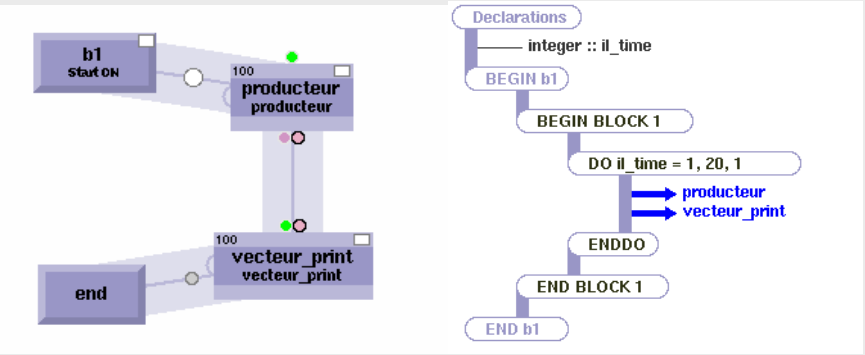
Complétez la carte d'identité et sauvez le fichier !

5.3 Les communications dans PrePALM

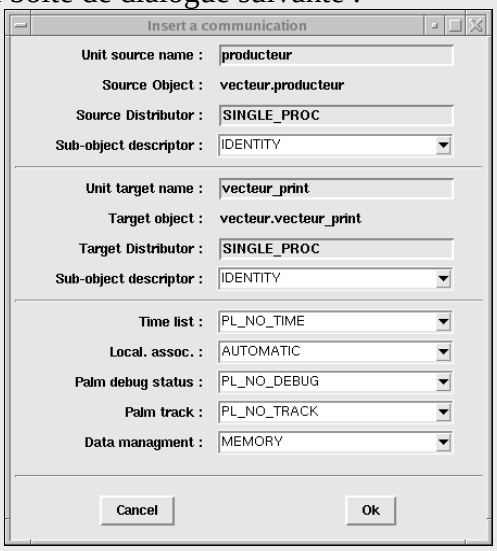
Nous allons maintenant faire fonctionner ensemble nos deux unités.

Communiquez !

- Partez d'un fichier PrePALM vierge dans le répertoire session_5
- Ajoutez une constante IP_SIZE qui vaut 1000 (taille du vecteur et de la matrice)
- Chargez les cartes d'identité de producteur et vecteur_print
- Insérez une branche b1, qui a l'attribut IP_START_ON
- Editez le code de branche, lancez producteur puis vecteur_print
- Ajoutez une boucle de 1 à 20 d'indice **il_time** autour du lancement des deux unités, et un bloc autour de la boucle, fermez la branche



- Sans cliquer, déplacez le curseur de la souris sur les plots (petits ronds de couleur des unités), examinez la fenêtre qui s'affiche et aussi le message d'aide en bas de PrePALM.
- Cliquez sur le plot correspondant à la production du vecteur de producteur, il devient rouge, puis sur le plot de vecteur_print correspondant au PALM_Get.
- Vous devriez obtenir la boite de dialogue suivante :



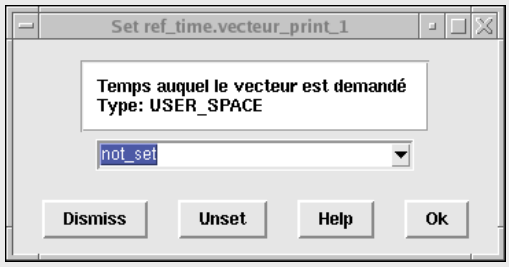
- dans le champ Time list, à la place de PL_NO_TIME, mettez 1:20

Vous venez de décrire votre première communication entre deux unités PALM. En initialisant le champ Time list à 1:20, vous autorisez l'envoi et la réception du vecteur (s'il est produit) à ces temps. Il reste maintenant à commander l'unité producteur pour qu'elle produise effectivement le vecteur à ces 20 temps et vecteur_print pour qu'elle demande aussi le vecteur pour tous ces temps.

Nous n'avons pas d'unité qui produit une liste de temps, il serait dommage d'être obligé d'en construire une à cette seule fin. Commander l'unité de produire ses vecteurs à un temps donné fait partie de l'**algorithme du couplage**, c'est donc à l'interface graphique de fournir le moyen de faire cela. Vous avez lancé les deux unités dans une boucle « temporelle » de 1 à 20, c'est bien sûr cette variable que nous allons utiliser pour dire aux deux unités de travailler à ces temps. Il y a deux solutions pour faire cela, soit faire un PALM_Put dans la branche, soit donner directement une valeur « en dur » à l'unité consommatrice.

Commandez !

- Ouvrez le code de branche
- Dans la boucle, avant producteur insérez un appel à PALM_Put d'une variable entière (insert Put … => one_integer)
- Saisissez il_time comme nom de variable
- Fermez le code de branche, un plot apparaît sur le trait de la branche dans le canevas
- Créez une communication entre cet objet et l'entrée de producteur (clic sur les deux plots)
- Vous pourriez faire de même pour vecteur_print, mais nous allons procéder autrement. Cliquez avec le bouton **droit** de la souris sur l'entrée ref_time de vecteur_print
- Le menu suivant s'ouvre :



- À la place de not_set sélectionnez la variable de branche il_time dans le menu déroulant, puis validez
- Votre application est prête vous pouvez la tester

5.4 Les listes de temps de la communication

Dans notre exemple, nous avons entré 1:20 pour la liste de temps. La syntaxe du champ **Time liste** est la suivante :

deb1[**:fin1**[**:stp1**]] [| **deb2** [**:fin2**[**:stp2**]]] [; …]

Les expressions entre [] sont optionnelles.
Le caractère pipe | permet de décrire des temps différents côté source et coté cible.
Le point virgule ; est utilisé pour séparer deux expressions.
Le caractère deux point : sert pour les boucles.

Exemples :

Expression	Source	Cible
18 ; 33 : 34	18 33 34	18 33 34
20 : 30 : 5	20 25 30	20 25 30
4|404	4	404
18|118 ;1|1 :3	18 1 1 1	118 1 2 3
1 :3 :2 | 1	1 3	1 1
1 :3 | 4 :6	1 2 3	4 5 6
1 :3 | 4 :8	Incorrect car deux boucles de chaque coté du | avec un nombre d'éléments différent	

Il est possible, pour chaque champ élémentaire, d'utiliser des constantes de PrePALM et des expressions arithmétiques. Les **variables** de branches sont **interdites,** car la définition doit rester statique, mais les constantes de PrePALM sont autorisées. Si les temps sources et cibles sont différents (utilisation du |), on peut référencer le temps source en fonction du temps cible noté **i**, du numéro d'ordre du temps cible noté **o,** du nombre total noté **nb**, et réciproquement.

Exemples :

Expression	Source	Cible
4:6 | **i**+100	4 5 6	104 105 106
4:6 | **o**+100	4 5 6	101 102 103
i*i | 1 :3	1 4 9	1 2 3
nb-**o** +1| 100:104:2	3 2 1	100 102 104

Combinées entre elles, ces notations permettent de décrire tout type d'association entre les temps sources et les temps cibles. C'est une possibilité, en pratique il est rare de traiter d'associer des temps différents entre la source et la cible.

Descriptions des champs tag

Si l'un des deux plots a son champ tag activé (-tag ON), la fenêtre de saisie demande de valider le champ **tag list**. Il s'agit ici de décrire tous les tags auxquels la communication doit être validée. La syntaxe du champ tag list est strictement identique à celle du champ time list. Nous n'utiliserons pas les tags pendant cette formation pour ne pas compliquer les choses, vous apprécierez vous même leur utilité lorsque vous développerez vos applications.

Combinaison des champs time et tag

Les temps et les tags décrits se combinent entre eux par un produit cartésien. Par exemple si on entre 10 :12 pour le champ time et 7 | 107 ; 4 | 44 pour le champ tag, les communications autorisées seront :

Source		Cible	
Time	Tag	Time	Tag
10	7	10	107
11	7	11	107
12	7	12	107
10	4	10	44
11	4	11	44
12	4	12	44

5.5 Les données en dur

Pour donner le temps auquel doit travailler vecteur_print, nous avons utilisé la variable de branche de la boucle. Dans ce champ, PrePALM accepte n'importe quel type d'expression Fortran90 valide. Cette possibilité de donner directement une expression pour un plot en entrée n'est pas réservée aux variables scalaires (0d), n'importe quel PALM_Get peut être initialisé de la sorte. Cette fonction peut être très intéressante pour faire des tests unitaires sur les unités. La seule contrainte est que l'expression pour initialiser la variable doit tenir en une seule instruction Fortran90.

Exercice 8 : test unitaire de vecteur_print
Partez d'un PrePALM vierge
Définissez la taille du vecteur à 50 dans les constantes de PrePALM
Chargez l'unité vecteur_print
Lancez l'unité en boucle avec ib_do variant de 10 à 100 par pas de 10
Définissez en dur le plot ref_time à la variable de boucle
Déclarez une variable entière i dans la branche
Initialisez en dur (clic droit sur le plot de l'objet) le vecteur à l'expression Fortran90 suivante :
`(/(i,i=1,IP_SIZE)/)*ib_do`
Remarque : ne pas utiliser de caractère espace dans l'expression, PrePALM ne les accepterait pas.

5.6 L'espace NULL et l'héritage

Dans cette session, nos deux unités producteur et vecteur_print peuvent s'échanger un tableau 1d car les objets sont compatibles, ie ils ont le même type et la même taille d'espace. Dans producteur et vecteur_print le tableau 1d de doubles précision est déclaré de taille IP_SIZE, constante utilisateur définie dans l'interface graphique, c'est cette même déclaration qui assure la compatibilité des objets à s'échanger. Notons que le contrôle de cohérence sur la taille des objets se fait au moment où l'utilisateur tente d'ajouter une communication. Ce contrôle se fait sur le type et la taille de l'espace et non sur les noms d'espace qui peuvent être différents dans l'unité source et l'unité cible. Pour des unités qui ont un certain degrés de parenté, par exemple issue d'un même code de calcul ou développées par la même personne, le fait d'assurer la compatibilité des espaces en leur associant une taille définie par une constante de PrePALM est satisfaisante même si elle impose d'utiliser le même nom de constante dans les deux unités. Cependant, notons que dans ce cas, il est clair qu'on n'assure en rien l'indépendance totale des unités puisqu'il faut que les deux unités se "partagent" la même constante.

L'utilisation de deux noms de constantes différentes par exemple IP_SIZE_P et IP_SIZE_V que l'on définirait, pour une application donnée, comme égales dans le menu d'édition des constantes de PrePALM, réglerait ce problème, mais seulement en partie car ceci n'autoriserait pas par exemple le fait de pouvoir manipuler, dans la même application, deux vecteurs de taille différentes dans deux instances de la même unité.

Pour rendre une unité comme vecteur_print beaucoup plus générique, la solution consisterait plutôt à utiliser l'espace générique NULL comme attribut de l'objet. Un espace NULL précise à PrePALM que les caractéristiques de ce dernier seront transmise lorsque l'utilisateur créera la communication. Les objets d'espace NULL sont de couleur jaune, puis ils changent de couleur lorsque la communication est établie. Ceci a naturellement une conséquence dans la programmation de l'unité vecteur_print, dans ce cas le tableau doit être alloué dynamiquement pour pouvoir prendre la taille désirée. Des primitives PALM permettent de récupérer la taille de l'objet dont on a hérité, ceci se fait en deux étapes :

- demande à PALM du nom de l'espace hérité : **PALM_Object_get_spacename**
- demande à PALM de la taille de l'espace : **PALM_Space_get_shape**

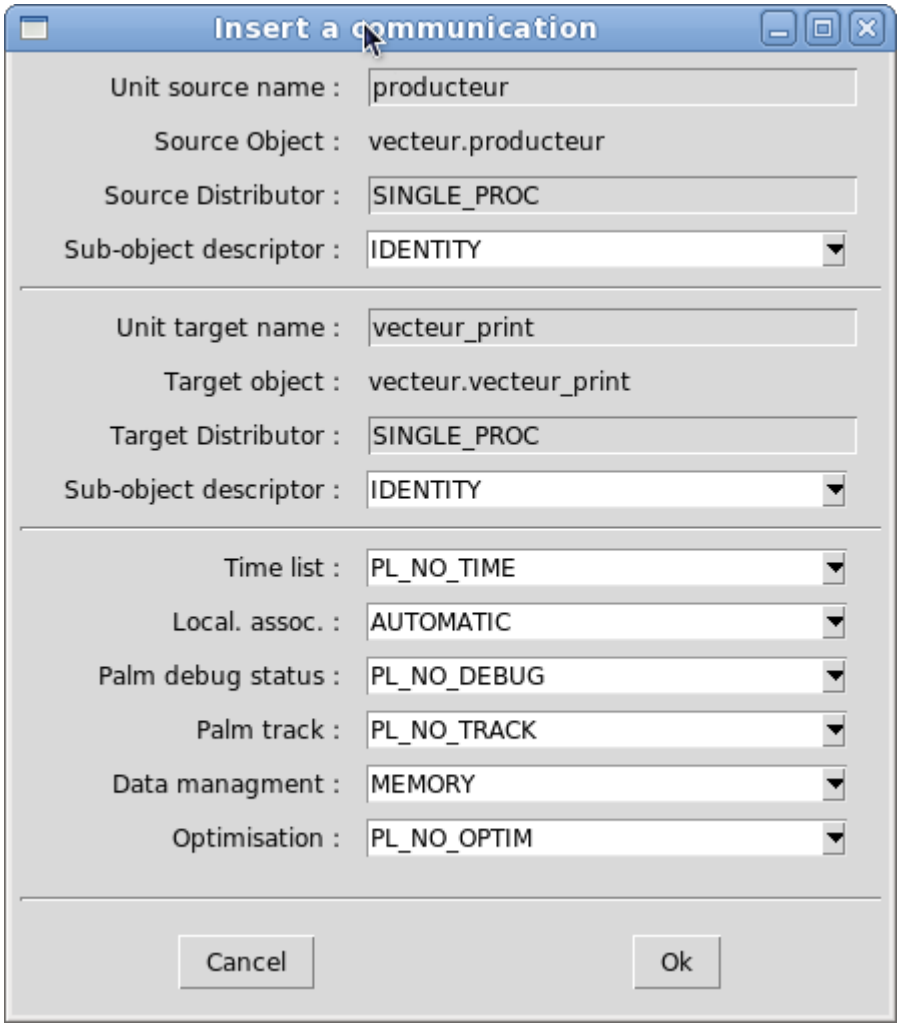
Une fois la taille du tableau connue, on peut allouer le tableau avant de faire le PALM_Get/Put. Notons que pour l'appel au PALM_Get ou le PALM_Put, l'espace à donner est l'espace 'NULL' comme défini dans la carte d'identité et non le nom de l'espace hérité.

Enfin notez que si vous détruisez une communication qui a permis l'héritage d'un espace pour un espace NULL, le plot conserve la couleur de l'espace hérité, ce qui vous interdit d'hériter à nouveau d'un autre espace. Pour revenir à un espace NULL (couleur jaune) il convient d'éditer les attributs de l'objet, pour cela il faut cliquer sur l'objet pour le sélectionner, puis double clic dans la liste de gauche de PrePALM afin d'ouvrir la fenêtre qui permet de remettre l'espace de l'objet à NULL.

Dans le répertoire de la session 9, vous trouverez une unité vecteur_print.f90 qui fait tout cela. Plus tard, dans cette session 9, nous apprendrons aussi à manipuler des objets de taille dynamique, dont la taille n'est connue qu'au moment de lancer l'application PALM, mais d'ici là, pour simplifier, nous garderons des unités qui définissent statiquement leurs espaces.

5.7 Les attributs des communications

Examinons maintenant tous les attributs possibles d'une communication. La boite de dialogue permettant d'insérer ou d'éditer une communication comporte certaines options que nous n'utiliserons pas forcément au cours de cette formation, mais qu'il faut connaitre à des fins de débogage ou d'optimisation des communications.



Dans la première zone de cette boite de dialogue on ne fait que rappeler les caractéristiques de l'objet source, soit :

- le nom de l'unité,
- le nom de l'objet, ici suffixé par le nom de l'unité pour le différencier si besoin des objets qui appartiendraient à des instances différentes de la même unité,
- le nom du distributeur associé à l'objet, ce nom servira pour les communications parallèles, dans cette session l'objet est de type SINGLE_PROC car l'unité n'est pas parallèle et le tableau correspondant à l'objet se trouve en totalité sur le processus de l'unité. Nous reviendrons sur l'utilité de ce champ dans la session consacrée aux communications parallèles.

- un nom de descripteur de sous-objet, lorsque l'objet est un tableau, il est possible de ne récupérer qu'un sous-ensemble des éléments de ce tableau, dans ce cas il faut définir un descripteur de sous objet qui détermine les parties à sélectionner dans le tableau. Une session de ce manuel illustre l'utilisation des sous-objets.

La seconde zone concerne les mêmes attributs pour l'unité cible.

Dans la troisième zone, on trouve :

- le champ "Time list" que nous avons décrit plus haut, cette zone n'apparait que si l'un des deux objets (source ou cible) possède l'attribut time ON dans la carte d'identité de l'unité.
- le champ "Tag list" si l'un des deux objets possède l'attribut tag ON
- le champs "local. assoc." qui sera décrit dans la session relative aux communications parallèles.
- le champ "debug status" qui peut prendre les valeurs PL_NO_DEBUG, PL_DEBUG_ON_SEND, PL_DEBUG_ON_RECV ou PL_DEBUG_ON_BOTH selon que l'utilisateur désire appeler une fonction dite de débogage (dont le prototype est décrit dans le fichier palm_debug.f90 ou palm_debug.c). Cette fonction de débogage permet par exemple, sans modifier l'unité source ou cible, de contrôler ou d'afficher les valeurs du tableau en transit. Pour utiliser le débogage des objets, il faut modifier le fichier palm_debug.f90, vous trouverez des explications détaillées dans ce fichier.
- le champ track, qui, associé au niveau de verbosité sur les communications, permet de suivre le déroulement des différentes étapes de la communication : envoie de la notification de Put/Get au driver de PALM, routage par le driver de PALM, etc, ces informations sont écrites dans les fichier de log de palm (palmdriver.log et branche_XXX.log).
- le champ "Data management", par défaut ce champ est positionné à "MEMORY". L'utilisateur peut éventuellement choisir "DISK". C'est une option avancée de PALM dont il est souhaitable de se passer car elle peut ralentir le couplage. Cette option permet de régler des problèmes éventuels de mémoire car dans ce cas, les objets "en transit" (Get posté après le Put) sont provisoirement stockés sur disque (plutôt qu'en mémoire) avant d'être envoyés. Il existe d'autres mécanismes pour régler ces problèmes (rares) de mémoire, et généralement, sauf cas très particuliers, l'utilisateur n'a pas à choisir l'attribut "DISK" pour le champ "Data management".
- le champ "Optimisation" permet lui d'optimiser certaines communications. L'activation de PL_OPTIM ne peut fonctionner que dans certains cas bien particuliers car elle impose certaines contraintes au niveau de l'agencement des Put/Get dans les unités PALM et peut vite conduire à des blocages de l'application, en effet l'utilisation de PL_OPTIM rend la primitive PALM_Put bloquante. Cette option est très utile pour des codes massivement parallèles. En effet dans le cas classique (PL_NO_OPTIM) tous les processus s'adressent dans un premier temps au driver de PALM pour connaitre le routage des données (envoi direct à l'unité cible ou stockage temporaire dans la mémoire du driver de PALM) ce qui peut, avec un grand nombre de processus, conduire à un goulet d'étranglement. Dans le cas PL_OPTIM le routage des communication est fait statiquement, les unités ne s'adressent donc jamais au driver de PALM.

PL_NO_OPTIM	PL_OPTIM
PALM_Put jamais bloquant, PALM rend la main à l'unité en interceptant les données à envoyer si nécessaire.	PALM_Put bloquant, l'unité source reste en attente du PALM_Get de l'unité cible pour réaliser la communication.

Bufferisation des données si le PALM_Put arrive avant le PALM_Get.	
Routage dynamique, envoie de message au driver de PALM	Routage statique, pas d'envoi de message au driver de PALM
Gain en souplesse et en généricité.	Gain en performance pour des unités parallèles avec un grand nombre de processus

5.8 Rappel des points de cette session

Dans cette session vous avez mis en pratique le concept de "one-sided communication" qui est la base du schéma de communication de PALM. Vous avez appris ce qu'est un objet et un espace, comment les décrire dans les cartes d'identité. Vous avez retenu que les espaces contenant une donnée de type simple (one_integer, one_real, etc) sont prédéfinis dans PALM et n'ont pas besoin d'être déclarés dans les cartes d'identité.
Vous savez définir les communications effectives de l'application de couplage, éventuellement en leur associant un champ TIME ou un champ TAG si le code produit plusieurs instances du même objet. Vous avez appris à décrire des listes de temps pour autoriser ou non les communications à se faire.
Vous savez que le PALM_Get peut être directement associé à une valeur définie dans l'interface graphique comme les constantes ou les variables de branche.
Vous avez appris comment terminer une application PALM proprement par la primitive PALM_Abort en testant les codes d'erreur retournés par les primitives PALM.
La notion d'espace NULL et d'héritage d'espace a été introduite.

Remarque :
A ce stade de la formation, vous avez vu l’essentiel du coupleur PALM, vous seriez déjà capable de créer une application couplée avec des unités indépendantes s’échangeant des informations. Toutes les autres sessions vont maintenant vous montrer des fonctionnalités plus avancées dont vous n’aurez pas nécessairement besoin mais qui peuvent vous faciliter le travail pour développer une application complexe.

6 Session 6 : Les unités prédéfinies

6.1 Introduction

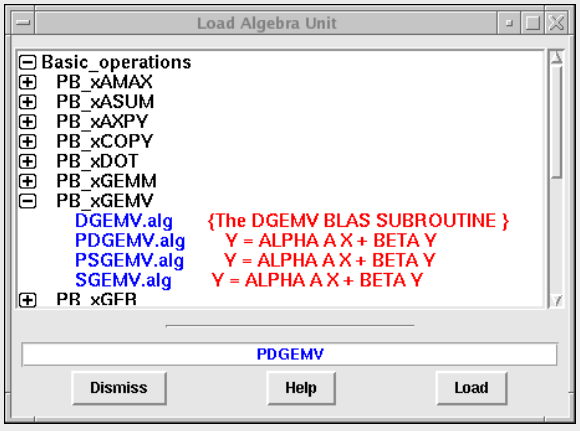
Pour construire une application de couplage, l'utilisateur peut être amené à faire des opérations d'algèbre linéaire sur les objets PALM avant de les échanger entre deux unités. Par exemple un produit matrice vecteur. Ces opérations « génériques », qui peuvent aller des plus basiques aux plus complexes sont disponibles directement dans l'interface graphique PrePALM. Lorsqu'elles existent, il est même fortement conseillé d'utiliser ces fonctions plutôt que de les développer vous-même car elles font appel aux bibliothèques mathématiques testées et optimisées sur le calculateur.

Les unités prédéfinies (ou unités d'algèbre) ont le même mode de fonctionnement que les unités utilisateur. La seule différence concerne la gestion de leurs attributs time et tag au niveau de la gestion des objets communiqués à ces unités. En effet, ces unités doivent être capables de recevoir leurs objets aux temps manipulés par l'utilisateur dans son application. Pour chaque objet reçu et/ou envoyé, il faudra donc préciser à quel time et à quel tag le PALM_Get et/ou PALM_Put doit être fait dans l'unité d'algèbre.

Pour illustrer l'utilisation des boîtes d'algèbre nous allons faire le produit de la matrice et du vecteur produits par l'unité producteur (identique à celle de la session 5)

Utilisez des boites pré-définies !

- Ouvrez un nouveau PrePALM dans le répertoire session_6
- Dans une première branche START_ON lancez producteur puis vecteur_print
- Faites travailler ces deux unités au temps 10 en donnant une valeur en dur aux bons plots
- Chargez la boite d'algèbre DGEMV par le menu File => Load Algebra unit

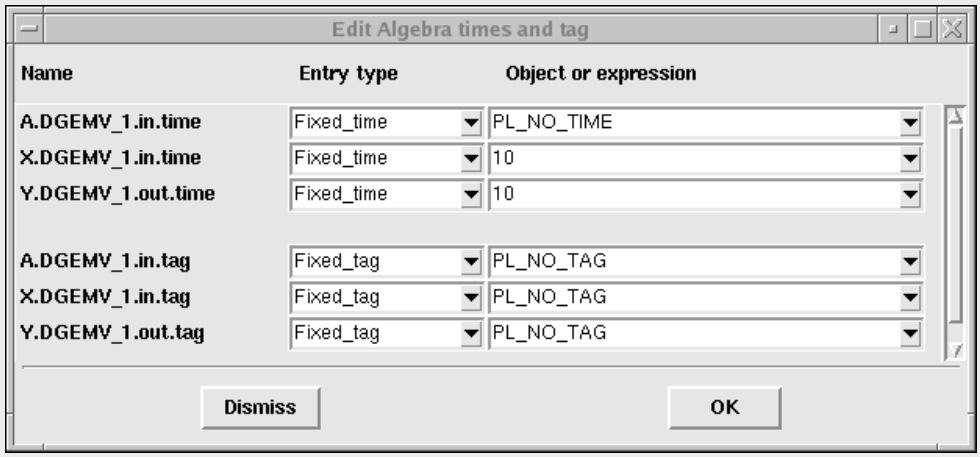


- Ouvrez Basic_operations => PB_xGEMV
- Sélectionnez DGEMV puis Load

Après le chargement de l'unité, PrePALM ouvre une fenêtre d'aide concernant cette routine. Vérifiez qu'elle correspond à ce que vous voulez faire et fermez la fenêtre.

Utilisez des boites pré-définies !

- Créez une seconde branche START_ON
- Lancez l'unité d'algèbre dans cette branche, fermez le code de branche
- Remarquez les trois plots jaunes en entrée et le plot jaune en sortie, ils indiquent que l'espace associé à ces objets n'est pas encore défini (espace NULL)
- Créez une communication PL_NO_TIME entre la matrice de producteur et l'objet A de DGEMV. Le trait en pointillés indique que le temps est PL_NO_TIME
- Envoyez le vecteur de producteur à l'objet X de DGEMV, entrez 10 pour la liste de temps
- Envoyez la sortie Y de DGEMV à vecteur_print au temps 10
- Mettez une valeur en dur 0. (clic droit) pour l'entrée Y de DGEMV
- Cliquez maintenant sur le rectangle de gauche de l'unité DGEMV (time & tag receiver)



- Cette boite de dialogue permet de gérer les temps des objets de l'unité d'algèbre. La matrice est produite avec PL_NO_TIME dans producteur, elle doit donc être reçue avec PL_NO_TIME dans DGEMV. Par contre l'objet X est produit au temps 10, il doit donc être demandé à ce même temps dans DGEMV.
- Entrez 10 pour X.DGEMV_1.in.time et Y.DGEMV_1.out.time
- Testez maintenant votre application

Remarque

Pour notre exemple nous avons imposé à une valeur Fixed_time le temps des deux objets X et Y.

Ce temps peut également être reçu : champ Entry types, sélection de Received_time. Dans ce cas un plot apparaîtra dans le rectangle time & tag receiver et il faudra alors définir une communication pour ce plot.

Le temps peut aussi être calculé à partir d'autres temps reçus : champ Entry type, sélection de Calculated_time. Dans ce cas il faut entrer une expression qui peut faire intervenir d'autres temps (reçus ou calculés), pour utiliser les autres temps, on donne le nom complet (colonne de gauche) précédé du signe $. Pour faire tourner cette application en boucle, on pourrait par exemple recevoir

le temps de l'indice de boucle pour X.DGEMV_1.in.time et calculer le temps pour Y.DGEMV_1.out.time en donnant comme expression : $X.DGEMV_1.in.time

Exercice 9
Partez de l'application précédente
Modifiez la taille du vecteur et de la matrice dans les constantes de PrePALM, passez de 1000 à 10 (pour plus de lisibilité des résultats).
Faites tourner producteur et vecteur_print dans un bloc, dans une boucle de 1 à 10 en produisant le vecteur à ces différents temps.

Le but est de multiplier le vecteur par la matrice uniquement pour les temps pairs (2,4,6,8 et 10). Pour les autres temps le vecteur produit par producteur doit être imprimé directement par vecteur_print (sans transiter par DGEMV)

Aide :
Vous pouvez conserver la seconde branche à START_ON et lancer DGEMV dans une boucle.
L'aiguillage entre les valeurs paires et impaires du temps du vecteur pour les deux unités se fera en sélectionnant les objets dans les champs Time list des communications.
Utilisez les temps reçus et calculés ainsi que les Put de variables de branche pour gérer les temps des objets communiqués à DGEM.
Une autre solution consiste à positionner la branche b2 à start OFF et à la lancer dans la branche b1 (sous condition) autant de fois que nécessaire.

6.2 Rappel des points de cette session

PALM fournit une boite à outils pour des opérations d'algèbre linéaire sur les vecteurs et les matrices. La seule différence entre ces unités prédéfinies et les unités utilisateur est le mécanisme de gestion des attributs time et tag des objets reçus ou envoyés par ces unités.

7 Session 7 : Les objets de type dérivé

7.1 Introduction

Les objets que nous avons échangés jusqu'à présent étaient tous d'un type simple, entier ou réel double précision. Il est possible de gérer dans PALM des objets de type dérivé correspondant à des structures de données définies par l'utilisateur. Cette possibilité est intéressante par exemple pour envoyer en un seul message plusieurs tableaux de nature différente ou simplement des tableaux de structures de données définies par l'utilisateur dans une unité. Selon que les types dérivés sont continus ou non en mémoire l'appel aux primitives PALM est plus ou moins pratique. Nous allons examiner les deux cas qui peuvent se présenter.

7.2 Objets continus en mémoire

Pour les objets continus en mémoire, qui ont donc été déclarés comme tels dans le programme source de l'unité, la procédure est identique aux objets classiques. La seule différence réside dans la description de la taille de leur espace. PALM doit en effet connaître (ou être en mesure de déduire) la taille du tableau à manipuler. Deux solutions sont proposées pour décrire la taille des espaces de type dérivée.

La première consiste à décrire le champ –element_size sous forme d'une expression arithmétique faisant intervenir les types de base de la structure sous forme de mots clés PALM identifiant les types de base (PL_INTEGER_SIZE, PL_REAL_SIZE, ...). Supposons que notre type dérivé soit le suivant :

```
  TYPE personne
    SEQUENCE
    CHARACTER*20 :: nom
    CHARACTER*20 :: prenom
    INTEGER :: age
    REAL :: taille
  END TYPE personne
```

Remarquez l'attribut SEQUENCE dans la définition Fortran90 de ce type dérivé, il force le compilateur à faire en sorte que les quatre champs soient continus en mémoire. Au niveau de la carte d'identité, dans une rubrique espace, la taille des éléments peut être décrite de la façon suivante :

```
!PALM_SPACE -name groupe_space\
!           -shape (3)\
!           -element_size 40*PL_CHARACTER_SIZE+PL_INTEGER_SIZE+PL_REAL_SIZE
```

La seconde solution consiste à faire référence à une liste d'espace dont la taille a déjà été définie. Pour le même exemple, il faudra déjà définir un espace de taille 20 caractères (chaine20) puis définir une liste d'items contenant un nom d'item et l'espace associé à cet item. Pour le même type dérivé on peut décrire son espace de la manière suivante :

```
!PALM_SPACE -name chaine20\
!           -shape (1)\
!           -element_size 20*PL_CHARACTER_SIZE\
!           -comment {20 caracteres}
!
!PALM_SPACE -name groupe_space\
!           -shape (3)\
!           -element_size PL_AUTO_SIZE\
!            -items {{nom chaine20} {prenom chaine20} {age one_integer} {taille
one_real}}\
!           -comment {type derive}
```

La seconde solution semble plus compliquée mais elle est compatible avec les objets non continus en mémoire. On préférera donc utiliser cette seconde manière pour décrire les objets de type dérivé.

Typez !

- Ouvrez un nouveau PrePALM dans le répertoire session_7
- Regardez le code source des unités personnes.f90 et pers_print.f90
- Lancez ces deux unités à la suite dans une seule branche
- Connectez les deux plots
- Testez l'application

L'exécution devrait vous donner ceci :

```
Alain     Dupond    a 27 ans et mesure 1.85 m
Sophie    Mercier   a 22 ans et mesure 1.62 m
Anne      Smith     a 43 ans et mesure 1.71 m
```

Exercice 10
Partez de l'application précédente
Construisez une troisième unité nommée vieillir.f90
Cette unité fera un PALM_Get d'un entier correspondant à un nombre d'année n, et fera vieillir de n an(s) toutes les personnes d'un objet groupe (à déclarer en entrée et en sortie)
Si aucune communication n'est décrite pour n dans l'application, n prendra une valeur par défaut de 1 an, pour cela initialisez la variable avant le PALM_Get et testez le code d'erreur.
Testez votre unité en l'intercalant entre personne et pers_print

7.3 Objets non continus en mémoire

Il est également possible de décrire des objets dont les éléments ne sont pas forcément continus en mémoire. Cette possibilité est très utile pour manipuler des structures de données (en langage C) manipulant des pointeurs (donc des tableaux allouables dynamiquement où l'alignement mémoire ne peut pas être garanti) ou plus simplement pour envoyer en un seul message plusieurs tableaux de nature différente.

Pour illustrer cette fonctionnalité, nous allons envoyer deux tableaux de nature différente en un seul PALM_Put. L'unité qui produit les données est écrite en C, celle qui les récupère est écrite en Fortran90, ceci pour illustrer les différences entre les langages concernant les appels aux primitives PALM.

Ouvrez le code source de producteur.c :

```
1    /*PALM_UNIT -name producteur\
2              -functions {C producteur}\
3              -object_files {producteur.o}\
4              -comment {pack de 2 tableaux}
5    */
6
7    /*PALM_SPACE -name entiers\
8                -shape (NB_ENTIERS)\
9              -element_size PL_INTEGER
10    */
11
12    /*PALM_SPACE -name reels\
13                -shape (NB_REELS)\
14          -element_size PL_REAL
15    */
16
17    /*PALM_SPACE -name typeder_s\
18                -shape (1)\
19                -element_size PL_AUTO_SIZE\
20                -items { {les_entiers entiers } {les_reels reels} }
21    */
22
23
24    /*PALM_OBJECT -name typeder_o\
25                 -space typeder_s\
26                 -intent OUT\
27                 -comment {exemple}
28
29    */
```

17 : avec le mot clé item nos deux tableaux vont être assemblés dans un objet composé de deux espaces différents (7 et 12). La taille des deux tableaux est paramétrée dans PrePALM par l'utilisation des deux constantes NB_ENTIERS et NB_REELS.

```
30    #include <stdio.h>
31    #include <stdlib.h>
32
33    #include "palmlibc.h"
34    #include "palm_user_paramc.h"
35
36
37    int producteur() {
38
39      float mes_reels[NB_REELS];
40      char cla_obj[PL_LNAME], cla_space[PL_LNAME];
41      void* buffer;
42      int il_time,il_tag;
43      int i,il_err;
44      int mes_entiers[NB_ENTIERS];
```

```
45      int ila_pos;
46
47
48      for (i=0; i<NB_ENTIERS; i++) {
49        mes_entiers[i] = i ;
50      }
51      for (i=0; i<NB_REELS; i++)  {
52        mes_reels[i] = i*1000. ;
53      }
54
55      /* allocation du buffer pour pack */
56      buffer = malloc(PALM_Space_get_size("typeder_s"));
57
58
59      ila_pos=0;
60
61
62      PALM_Pack(buffer,"typeder_s","les_entiers",&ila_pos,mes_entiers);
63      PALM_Pack(buffer,"typeder_s","les_reels",&ila_pos,mes_reels);
64
65      sprintf(cla_obj,"typeder_o");
66      sprintf(cla_space,"typeder_s");
67      il_time = PL_NO_TIME;
68      il_tag = PL_NO_TAG;
69
70      il_err = PALM_Put(cla_space, cla_obj, &il_time, &il_tag, buffer);
71
72      free(buffer);
73    }
```

34 : en langage C, cet include permet d'utiliser les constantes définies dans PrePALM pour dimensionner les tableaux.
56 : pour envoyer les deux tableaux avec un seul PALM_Put, on est obligé de les copier dans une variable intermédiaire. En C on définit un pointeur void que l'on alloue à la taille de l'objet déclaré dans la carte d'identité. La primitive PALM_Space_get_size permet de retourner cette taille.
59 : la variable ila_pos sert à définir la position de chaque élément de la structure, ici nous n'avons qu'un seul élément (shape (1) pour typeder_s). Dans notre exemple ila_pos vaut 0 car en C les tableaux commencent à l'indice 0.
62 et 63 : les deux vecteurs sont recopiés dans la variable buffer avec les primitives PALM_Pack
70 : l'objet assemblé est envoyé avec la primitive PALM_Put

Examinons maintenant de plus près l'unité consommateur.f90 :

```
1     !PALM_UNIT -name consommateur\
2     !          -functions {F90 consommateur}\
3     !          -object_files {consommateur.o}\
4     !          -comment {unpack de 2 tableaux}
5     !
6     !PALM_SPACE -name entiers\
7     !           -shape (NB_ENTIERS)\
8     !           -element_size PL_INTEGER
9     !
10    !PALM_SPACE -name reels\
11    !           -shape (NB_REELS)\
12    !            -element_size PL_REAL
13    !
14    !PALM_SPACE -name typeder_s\
15    !           -shape (1)\
```

```
16    !           -element_size PL_AUTO_SIZE\
17    !           -items { {les_entiers entiers } {les_reels reels} }
18    !
19    !
20    !PALM_OBJECT -name typeder_o\
21    !            -space typeder_s\
22    !            -intent IN\
23    !            -comment {exemple}
24
25
26    SUBROUTINE consommateur
27
28       USE palmlib
29       USE palm_user_param
30       IMPLICIT NONE
31
32       CHARACTER(LEN=PL_LNAME) :: cl_object, cl_space,cl_item
33       INTEGER :: il_err, il_size
34
35       REAL :: mes_reels(NB_REELS)
36       INTEGER :: mes_entiers(NB_ENTIERS)
37
38       INTEGER , ALLOCATABLE :: buffer(:)
39       INTEGER :: il_pos
40
41
42
43    !  allocation du buffer pour reception de l'objet
44      cl_space = 'typeder_s'
45      CALL PALM_Space_get_size(cl_space,il_size, il_err)
46    ! remarquez que la taille est en octet, comme on utilise un tableau
47    ! entiers (4 octets) par entier, on divise cette taille par 4
48      il_size= il_size/4
49
50      ALLOCATE(buffer(il_size))
51
52      cl_object = 'typeder_o'
53          CALL  PALM_Get(cl_space,  cl_object,  PL_NO_TIME,  PL_NO_TAG,  buffer,
il_err)
54
55      il_pos = 1 ! on recupere le premier element (il n'y en a qu'un)
56
57      cl_item = 'les_entiers'
58      CALL PALM_Unpack(buffer,cl_space,cl_item,il_pos, mes_entiers, il_err)
59      print *, 'entiers--->', mes_entiers
60
61      cl_item = 'les_reels'
62      CALL PALM_Unpack(buffer,cl_space,cl_item,il_pos,mes_reels, il_err)
63      print *, 'reels--->', mes_reels
64
65      DEALLOCATE(buffer)
66
67    END SUBROUTINE consommateur
```

45 : l'appel à la primitive PALM_Space_get_size est différent en C et en Fortran.
48 : lisez le commentaire
55 : ici la position de l'élément est 1 car les tableaux Fortran commencent leur premier indice à 1.

Exercice 11
Testez ces deux unités

Remarque
Les types dérivés sont bien pratiques et apportent souvent de la rigueur et de la souplesse en programmation. Il ne faut pas non plus tomber dans un excès et vouloir à tout prix encapsuler toutes les données dans des types dérivés qui peuvent être lourd à manipuler. Plus vos unités PALM manipuleront des types dérivées, moins elles seront portables et interfaçables avec d'autres unités. Pensez par exemple aux unités d'algèbre linéaires, pour être portables, elles ne manipulent que des types simples. Si vous envoyez toutes vos données sous forme de types dérivés vous serez dans l'impossibilité d'utiliser des boites d'algèbre prédéfinies, sauf en intercalant des unités PALM d'interface.

7.4 Rappel des points de cette session

Cette session est entièrement consacrée à la manipulation de types dérivés et de structures. Si les objets sont continus en mémoire, ils ne demandent qu'une description correcte des espaces dans la carte d'identité de l'unité. Sinon, vous avez appris à utiliser les primitives PALM_Pack et PALM_Unpack pour les envoyer et ou les recevoir avec les primitives PALM_Put/Get.

8 Session 8 : Le BUFFER et l'interpolation temporelle

8.1 Introduction

Lorsque l'on couple deux codes de calcul, il est rare que les deux programmes aient le même pas de temps. Pour contourner ce problème on est en général amené à interpoler dans le temps les champs physiques, donc à conserver en mémoire plusieurs instances temporelles du même objet. L'interpolation temporelle est directement offerte dans le coupleur PALM : une unité peut par exemple produire ses objets toutes les 10 secondes alors qu'une seconde unité les demande toutes les 3 secondes. Associé à la fonctionnalité d'interpolation temporelle nous allons voir comment gérer des données dans une mémoire permanente de PALM appelée le BUFFER. Pour ne pas saturer ce BUFFER d'informations qui n'auraient plus lieu d'y rester, un mécanisme assez souple permet de gérer finement les objets stockés dans le BUFFER, c'est le langage steplang.

8.2 Préparation des unités

Nous allons repartir de l'unité producteur que nous avons améliorée pour qu'elle ressemble de plus près au fonctionnement d'un code de calcul. Dans une boucle interne elle produira le vecteur à différents pas de temps, ce qui est souvent le cas dans les codes de calcul. Pour avoir plus de souplesse, cette unité demandera les indices de début, de fin, et de fréquence de sa boucle interne.

Envoyez vos objets dans le BUFFER!

- Ouvrez un nouveau PrePALM dans le répertoire session_8
- Définissez quatre constantes : IP_SIZE = 100000 (taille du vecteur), debut_prod = 0, fin_prod = 1000 et step_prod = 10
- Construisez une branche b1 qui lance producteur. En entrée de producteur, pour contrôler les instants produits, donnez les valeurs en dur en sélectionnant les constantes entrées précédemment
- Envoyez le vecteur dans le BUFFER de PALM, pour cela double-cliquez sur le plot du vecteur. La boite de dialogue vous propose un nom d'objet pour le vecteur dans le BUFFER (vecteur). Pour le champ time list entrez **debut_prod:fin_prod:step_prod** puis validez
- Dans le canevas vous devez voir apparaître une communication qui va se brancher sur un petit carré matérialisant le BUFFER de PALM

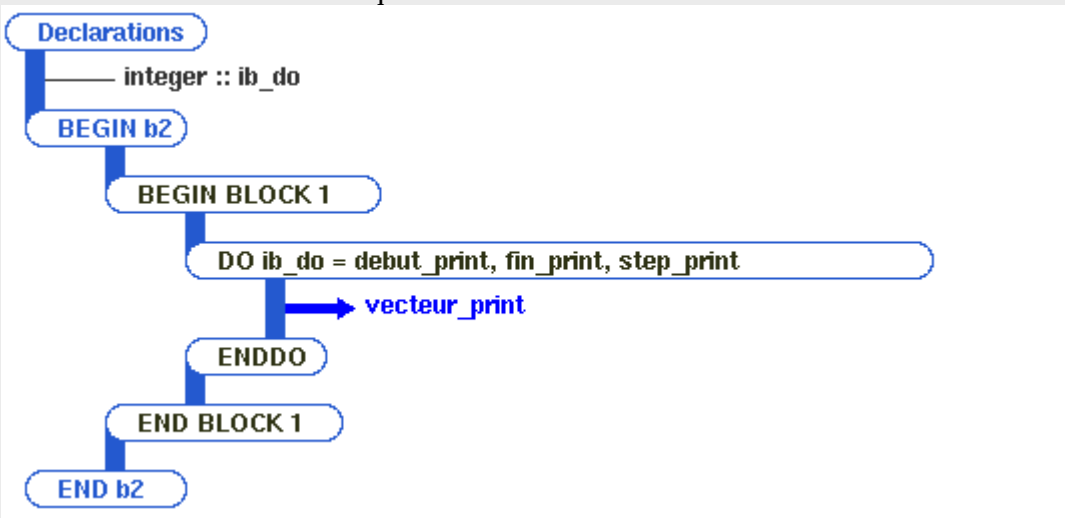
Vous venez d'autoriser toutes les instances temporelles du vecteur à être stockées dans la mémoire permanente de PALM. Physiquement, ces données sont gérées par le processus du driver de PALM (palm_main). Mais attention, tant que vous ne préciserez pas qu'il faut les détruire, ces objets résideront en mémoire.

Nous allons maintenant faire de l'interpolation temporelle sur le vecteur. Dans cet exemple nous nous contenterons de choisir une interpolation linéaire entre les deux instants les plus proches de

celui demandé. On pourrait aussi demander à PALM de prendre le plus proche voisin ou définir nous-mêmes une interpolation calculée à partir des deux voisins les plus proches, dans ce cas il faudrait intervenir dans un fichier source fortran (palm_time_int.f90 du menu Make PALM files) pour coder les calculs.

Interpolez !

- Définissez trois nouvelles constantes debut_print = 1, fin_print = 1000 et step_print = 7
- Créez une seconde branche qui fait ceci :



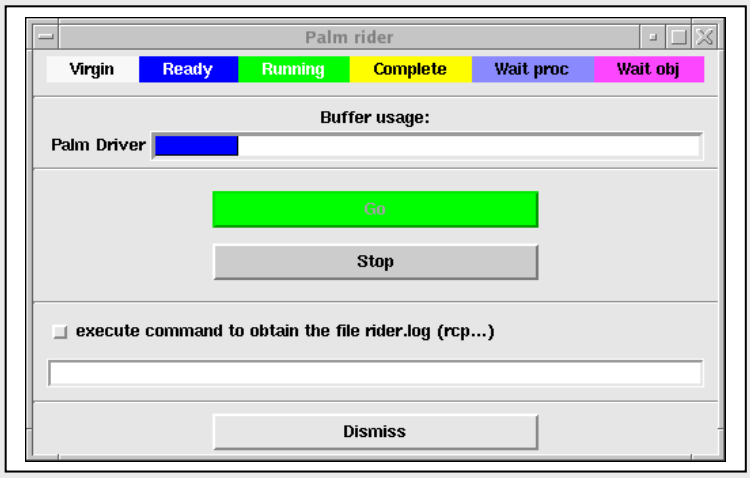
- Donnez (en dur) l'indice de boucle à l'unité vecteur_print pour le plot ref_time
- Double cliquez sur le plot du vecteur
- Entrez debut_print:fin_print:step_print pour le champ time list
- Choisissez **PL_GET_LINEAR** pour le champ interpolation
- Dans le menu settings => palm execution settings cochez toutes les cases sauf la dernière
- Dans le menu settings => palm memory settings mettez 100 pour le champ **Memory for the driver mailbuff …**, cela correspond à la taille maximum du BUFFER de PALM
- Testez l'application

8.3 Suivi de l'exécution en temps réel

Il est normal que l'exécution prenne un certain temps à tourner, en fait dans la boucle de production du vecteur de producteur un appel à sleep(1) ralentit artificiellement l'exécution de cette unité. Nous allons en profiter pour suivre le déroulement de l'exécution en temps réel.

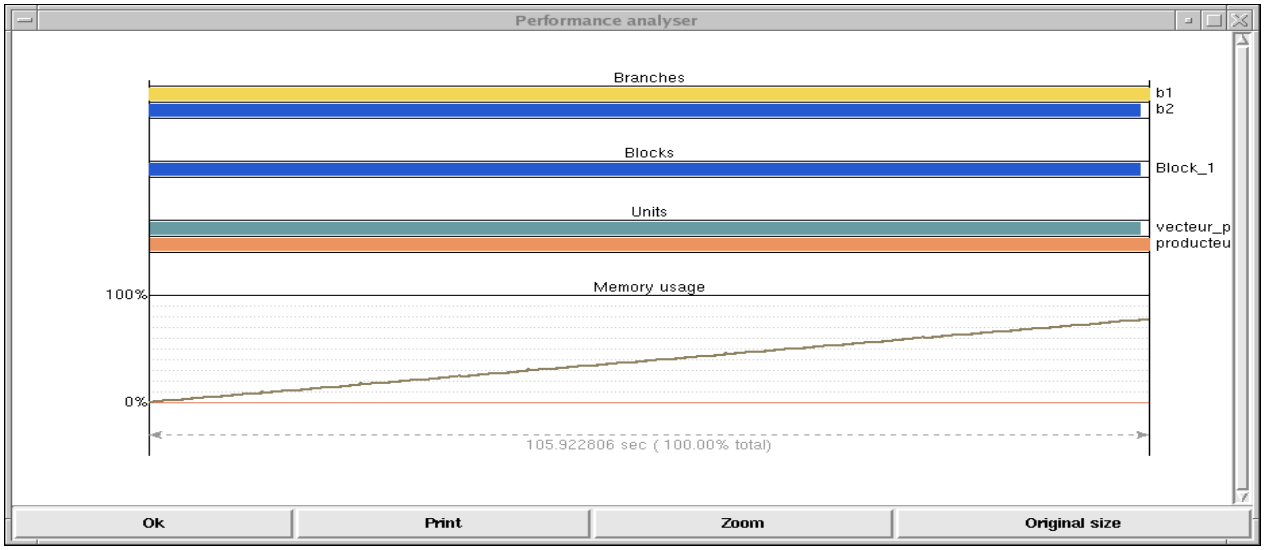
Supervisez l'application !

- Menu Analyse run => connect to run



- Clic sur go !

Dans le canevas de PrePALM, les unités prennent une couleur verte lorsqu'elles sont en train d'être exécutée et jaune lorsqu'elles ont terminé. À droite des unités, vous pouvez remarquer un nombre en rouge qui indique combien de fois chaque unité s'est exécutée. Sur la gauche de l'unité producteur (qui ne tourne qu'une seule fois car elle a une boucle sur le temps interne), un « camembert » permet de suivre la progression de l'unité, si vous ouvrez le fichier producteur.f90 vous trouverez un appel à la primitive PALM_Unit_set_progress qui permet cela. La jauge Buffer usage de la boite de dialogue « Palm rider » permet de suivre l'occupation de la mémoire du driver de PALM, vous remarquerez que cette taille ne fait qu'augmenter tout au long du run, il y a donc risque de saturation de la mémoire. Ceci est normal puisque nous avons indiqué que les objets produits à tous les pas de temps sont stockés dans le BUFFER de PALM. Nous allons voir comment contourner ce problème pour notre application. Mais avant cela, vérifiez que dans le fichier b2_000.log vous avez les bonnes valeurs interpolées du vecteur, c'est facile à vérifier car notre vecteur dépend linéairement du temps. Une autre façon de voir l'occupation du BUFFER consiste à ouvrir l'analyseur de performance après avoir chargé le fichier palmperf.log :

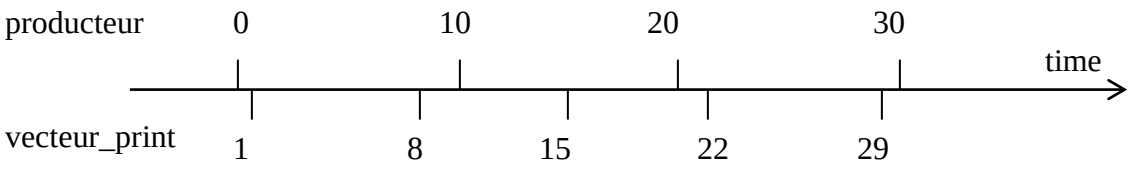


8.4 Les steps, les évènements et les actions

Dans la session 4, nous avons déjà vu comment définir des steps de synchronisation (PL_BARRIER_ON). Les steps sont des évènements que l'on provoque dans les branches. Associés à ces steps, il est possible de définir des actions sur les objets du BUFFER. Les communications gérées par PALM sont également des événements sur lesquels on peut déclencher des actions sur les objets dans le BUFFER. L'association entre les événements et les actions se fait par un langage de programmation propre à PALM nommé "steplang". Dans le menu Help de PrePALM => Help on steplang grammar vous trouverez la syntaxe du langage steplang. Lisez attentivement cette aide.

Vous n'avez pas compris grand chose ? Ça n'est pas grave, un exemple concret va vous montrer que ce n'est pas si difficile que ça d'utiliser steplang.

Revenons à notre problème d'interpolation. L'unité producteur produit le vecteur tous les 10 pas de temps à partir du temps 0. L'unité vecteur_print a besoin des valeurs de ce vecteur tous les 7 pas de temps à partir du temps 1. Pour que l'interpolation fonctionne à un pas de temps déterminé il faut naturellement que le vecteur ait été produit aux deux pas de temps de producteur encadrant ce pas de temps.



Par exemple lorsque vecteur_print a besoin du pas de temps 15, il faut que producteur ait produit 10 et 20. Par contre, à ce moment de l'application, on n'aura plus jamais besoin du temps 0. Idem pour les temps 22 et 29 qui n'auront plus besoin du temps 10. (voir figure)

Pour optimiser la mémoire, on aimerait faire cette action :

Pour tous les temps à partir du temps 15, à chaque fois que l'on fait une communication entre le BUFFER et vecteur_print, détruire l'objet du BUFFER avant l'encadrement de ce temps.

Ceci se traduit facilement en langage steplang par :

```
for $time in [15:$fin_prod:$step_print] {
  on {
     com("BUFFER", 0, "vecteur", $time, PL_NO_TAG,
         "vecteur_print", 0, "vecteur", $time, PL_NO_TAG);
  } do {
     $time1 = ($time / 10 - 1 ) * 10 ;
     delete("vecteur", $time1, PL_NO_TAG);
  }
}
```

Belle cuisine avec des opérations sur des entiers et les priorités opératoires !

La souplesse du langage Steplang et un peu d'imagination permet donc de traduire aisément des actions relativement complexes pour gérer les objets dans le buffer.

Faites le ménage !

- Menu Step-actions => Edit Step-actions
- Entrez les instructions steplang décrites ci-dessus en vous aidant de prepalm :
  - Repérez la communication entre le buffer et vecteur_print, cliquez sur le carré matérialisant la communication depuis le buffer
  - Dans l'éditeur de steplang cliquez sur le bouton « Paste last comm/step selected »
- Vérifiez la syntaxe de votre script par la commande Check step-actions syntax
- Relancez votre application et vérifiez que les objets devenus inutiles sont bien éliminés du BUFFER.

Nous avons vu comment optimiser la mémoire pour cette application. Dans notre cas tout se passe correctement car l'unité producteur produit ses objets beaucoup plus lentement qu'ils ne sont interpolés et récupérés. En fait producteur est ralenti artificiellement d'une seconde par l'appel à la fonction sleep(1) entre chaque objet produit. Si vous commentiez cet appel, vous vous rendriez compte que le BUFFER s'étend en mémoire et que celle ci ne diminue qu'au fur et à mesure que vecteur_print récupère les objets dans le BUFFER. Ceci se produit tout simplement parce qu'un **PALM_Put** n'est **jamais bloquant**. Pour optimiser l'application en mémoire, il conviendrait donc de synchroniser les deux unités. Les synchronisations peuvent se faire tout simplement par des appels à la primitive PALM_Get qui, elle, est bloquante (si le Get est effectivement connecté dans PrePALM).

Pour cela, dans producteur, nous allons ajouter un appel à PALM_Get d'une valeur entière de synchronisation, peut importe ce qu'on récupère, ce Get ne sert qu'à synchroniser. Il doit être fait dans la boucle interne de production du vecteur, après celle-ci.

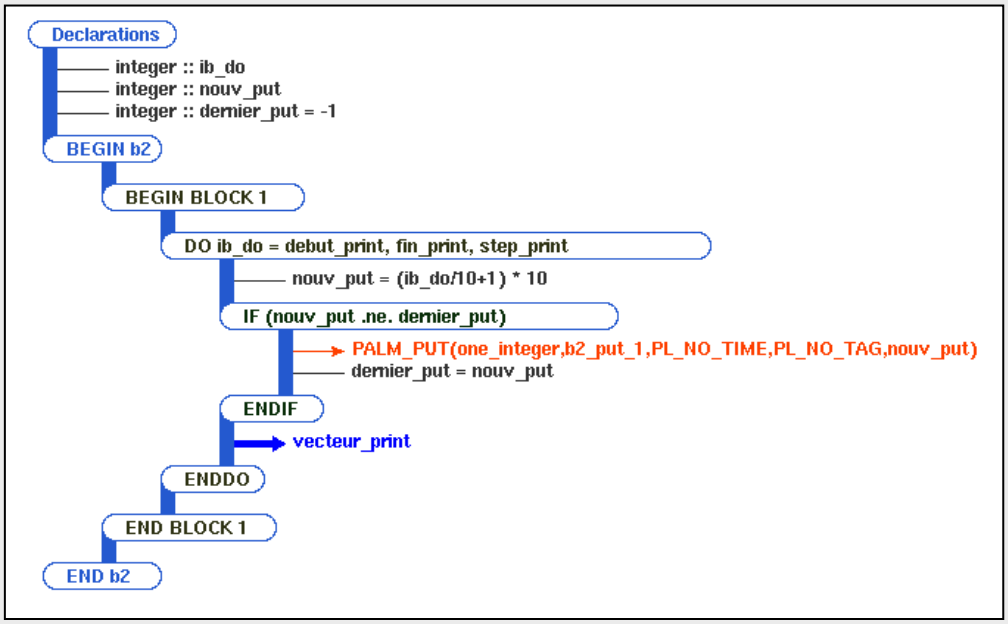
Ralentissez producteur !

- Editez le fichier producteur.f90
- Commentez l'appel à sleep(1)
- Ajoutez un objet 'synchro' : one_integer IN, sans time, sans tag au niveau de la carte d'identité.
- Dans la boucle de production du vecteur, après le PALM_Put, faites un PALM_Get de l'objet nouvellement crée.
- Relisez la carte d'identité de producteur dans PrePALM
- Le plot que vous avez ajouté doit apparaître dans le canevas de PrePALM

Il nous faut maintenant décider qui va envoyer un objet à chaque itération de producteur pour le ralentir. Le mieux est de faire ce travail dans la branche de vecteur_print où nous avons une boucle externe sur le temps. Le seul problème est que les temps que nous gérons dans cette branche ne sont pas les mêmes que ceux de producteur, quelques petits calculs seront donc nécessaires pour générer le bon nombre de Put de synchronisation dans la branche. Heureusement PrePALM sait faire ce genre de calculs avec les régions Fortran.

Synchronisez !

- Modifiez la branche de vecteur_print ainsi :



- Connectez le PALM_Put de branche à l'unité producteur
- Testez l'application

Maintenant, l'unité producteur est ralentie "juste ce qu'il faut" pour produire ses objets au fur et à mesure des besoins de vecteur_print. Ces synchronisations sont très importantes lorsque l'on fait du calcul parallèle et qu'on cherche à optimiser les applications. On peut donc être amené à ajouter des appels aux primitives PALM pour ce seul besoin.

8.5 Les esclaves mémoire

Lorsqu'il n'est pas possible de libérer suffisamment de place dans le BUFFER de PALM, ce dernier offre une autre alternative. On adjoint au driver de PALM d'autres processus dédiés à la gestion de la mémoire du BUFFER. Ces processus, s'ils sont exécutés sur d'autres processeurs, permettent d'étendre le BUFFER et donc de dépasser les limites du driver seul. Remarquez que cette option n'a d'intérêt que sur des machines à mémoire distribuée car sur des machines à mémoire partagée l'ensemble de la mémoire est accessible par chaque processeur. Notons que la mémoire demandée pour le driver de PALM (palm_main) et les esclaves mémoire n'est pas allouée au départ, mais au fur et à mesure des besoins. Demander plus de mémoire que disponible sur le processeurs sur lequel tourne le driver peut conduire, seulement si elle est effectivement consommée, à une pagination de l'application ou pire à un plantage (dans ce cas un message explicite est généré dans palm_driver.log).

Dans le menu Palm memory settings, vous pouvez préciser le nombre d'esclaves mémoire désirés et préciser si ces derniers doivent être lancés dès le début de l'application (min memory slaves = max memory slaves) ou au fur et à mesure des besoins (min memory slaves = 0).

L'application que nous allons utiliser pour illustrer l'utilisation des esclaves mémoire est toujours notre problème d'interpolation, mais maintenant les objets sont récupérés dans le BUFFER en ordre inverse de leur production. On est donc obligé de stocker toute la trajectoire en mémoire. Les personnes qui font de l'assimilation de données trouveront sûrement un intérêt à faire cela...

Faites vous aider !

- Ouvrez le fichier slave_mem.ppl avec PrePALM
- Lancez l'application avec le suivi en temps réel et observez le comportement de l'application

Exercice 12
Répondez aux questions :
Combien y'a t'il d'esclaves mémoire ?
Combien de processus, y compris le driver de PALM ?
Quelle est la taille totale de la mailbuf de PALM ?
Que fait-on sur le Step 1 ?
Y'a t'il un intérêt à faire tourner les deux unités en parallèle ?
Que peut-on faire pour gagner un processus ?

Faites tourner l'application avec un processus de moins.

8.6 Rappel des points de cette session

Dans cette session vous avez appris comment utiliser la fonctionnalité d'interpolation temporelle de PALM pour recevoir des objets à des temps intermédiaires non produits. Vous avez appris à utiliser le Buffer de PALM en y déposant explicitement des objets qui ne sont pas détruits à la réception (contrairement à la mailbuff de PALM).
Pour gérer ces objets dans le Buffer on utilise le langage steplang qui permet de préciser des actions sur des événements comme les steps de PALM ou les communications.
Vous avez mis en œuvre l'outil de suivi en temps réel de l'application offert par PrePALM. Enfin vous avez appris les principes de base pour optimiser la mémoire totale de l'application et la synchronisation des codes à coupler.

9 Session 9 : L'héritage d'espace et les objets dynamiques

Dans toutes les unités que nous avons utilisées jusqu'à présent, la taille mémoire des objets était connue au moment de la compilation de l'application. On a vu que cette taille pouvait être paramétrée via les constantes de PrePALM, mais elle reste cependant statique, elle ne peut pas évoluer au cours du run. Les codes de calcul que l'on se propose de coupler avec PALM peuvent fonctionner avec des objets dont la taille peut n'être connue qu'au moment où le programme s'exécute, typiquement si les tableaux qui contiennent les données sont alloués dynamiquement. Nous allons maintenant aborder la manière de traiter ce type d'objets.

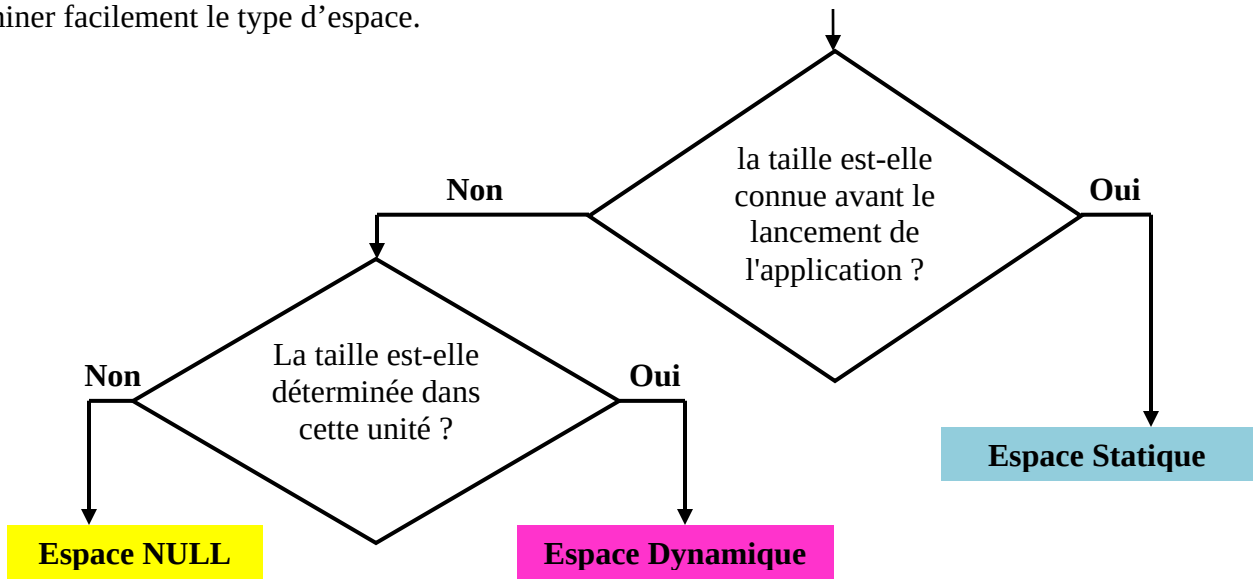
Faisons déjà une première constatation, la taille d'un objet dynamique doit tout de même être connue avant l'échange, et c'est forcement l'unité qui produit l'objet et elle seule qui connaît sa taille courante. Donc cette unité peut et doit, avant l'envoi, faire connaître à PALM la taille effective de son objet pour que le coupleur puisse faire transiter la bonne quantité d'informations, la primitive PALM_Space_set_shape va servir à cela.

L'unité qui reçoit l'objet peut se contenter d'utiliser le même espace pour l'échange, comme les unités sont supposées être indépendantes (les espaces sont propres à chaque unité), on utilise dans ce cas l'espace NULL pour l'objet. C'est encore dans l'interface graphique PrePALM qu'on va préciser que l'espace de l'unité réceptrice hérite des propriétés de l'objet source. Ceci se fait tout simplement en définissant une nouvelle communication, les espaces NULL héritent des caractéristiques des espaces communiqués.

Dans PrePALM les objets dynamiques ont une couleur rose fluo qui permet de les différencier des objets statiques. Les objets dont l'espace est indéfini (NULL) ont une couleur jaune.

Notez bien que les espaces NULL peuvent aussi bien hériter des propriétés d'un espace dynamique que d'un espace statique. Les boites d'algèbre, que nous avons présentées dans la session 6, utilisent les espaces NULL pour être génériques, ainsi peut-on trouver dans la même application plusieurs instances de la même unité prédéfinie qui travaillent sur des objets de taille différente, ce que n'autorise pas l'usage des constantes.

Important, à retenir : La réponse aux deux questions du diagramme ci-dessous permet de déterminer facilement le type d'espace.



Exercice 13
Dans cet exercice, l'unité producteur va maintenant produire une matrice et un vecteur au temps PL_NO_TIME. La taille de la matrice et du vecteur est demandée par un palm_get dans l'unité producteur.
Une seconde unité, produit_mv, permettra de faire le produit de la matrice par le vecteur et de restituer le résultat à l'unité vecteur_print. Les unités produit_mv et vecteur_print devrons donc être capables d'accepter des matrices et des vecteurs de dimension quelconque.

Répondez aux questions :
Combien d'objets pour producteur, quels espaces (statique, dynamique, NULL) ?
Combien d'objets pour produit_mv, quels espaces (statique, dynamique, NULL) ?
Combien d'objets pour vecteur_print, quels espaces (statique, dynamique, NULL) ?

Ouvrez l'unité producteur.f90 du répertoire session_9.
Par l'appel à quelle primitive, PALM connaît-t-il la taille de la matrice et du vecteur ?
A quoi correspondent le second et le troisième argument de cette primitive ?

Ouvrez l'unité produit_mv
Remarquez que les espaces des objets sont déclarés à NULL dans la carte d'identité.
Comment connaît-on l'espace hérité ?
Comment connaît-on la taille des espaces ?

Améliorez l'unité produit_mv pour que celle-ci arrête l'application PALM si la matrice ou le vecteur ne sont pas reçus.

Assemblez ces trois unités dans une seule branche, faire une boucle autour des trois unités (par exemple de 10 à 100 par pas de 10) qui vous permettra de traiter différentes tailles de vecteur.

Faites tourner l'application et vérifiez les résultats.

Unité producteur.f90 :

```
!PALM_UNIT -name producteur\
!          -functions {F90 producteur}\
!          -object_files {producteur.o}\
!          -comment {producteur}
!
!PALM_SPACE -name mat2d\
!           -shape (:,:)\
!           -element_size PL_DOUBLE_PRECISION\
!           -comment {tableau 2d double precision}
!
!PALM_SPACE -name vect1d\
!           -shape (:)\
!           -element_size PL_DOUBLE_PRECISION\
!           -comment {tableau 1d double precision}
!
!PALM_OBJECT -name dynsize\
```

```
!            -space one_integer\
!            -intent IN\
!            -comment {vector and matrix size}
!
!
!PALM_OBJECT -name matrice\
!            -space mat2d\
!            -intent OUT\
!            -comment {matrice 2d}
!
!PALM_OBJECT -name vecteur\
!            -space vect1d\
!            -intent OUT\
!            -comment {vecteur 1d}


    SUBROUTINE producteur

      USE palmlib             ! interface PALM

      IMPLICIT NONE

      CHARACTER(LEN=PL_LNAME) :: cl_object, cl_space

      DOUBLE PRECISION, ALLOCATABLE :: dla_vect(:), dla_mat(:,:)
      integer :: dyn_size, i, il_err, il_rank, ila_shape(2)


    ! taille du vecteur (et de la matrice)
      dyn_size = 0
      cl_space  = 'one_integer'
      cl_object = 'dynsize'
      CALL PALM_Get(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, dyn_size,
il_err)
      IF (il_err .ne. 0) THEN
         write(PL_OUT,*) 'Taille du vecteur non recue dans producteur'
         call PALM_Abort(il_err)
      ENDIF

    ! allocation dynamique des tableaux
      ALLOCATE(dla_vect(dyn_size))
      ALLOCATE(dla_mat(dyn_size,dyn_size))

    ! initialisation de dla_mat : matrice diagonale 1, 2, 3 ...
      dla_mat = 0.d0
      DO i = 1 , dyn_size
        dla_mat(i,i) = i
      ENDDO

    ! envoi de la matrice
      cl_space  = 'mat2d'
      il_rank = 2
      ila_shape(1) = dyn_size
      ila_shape(2) = dyn_size

      call PALM_Space_set_shape(cl_space, il_rank, ila_shape, il_err)

      cl_object = 'matrice'
      CALL PALM_Put(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, dla_mat, il_err)



    ! initialisation du vecteur
      DO i = 1 , dyn_size
```

```
        dla_vect(i) = i
      ENDDO

    ! envoi du vecteur
      cl_space  = 'vect1d'
      il_rank = 1
      ila_shape(1) = dyn_size
      call PALM_Space_set_shape(cl_space, il_rank, ila_shape, il_err)

      cl_object = 'vecteur'
      CALL PALM_Put(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, dla_vect,
il_err)

      DEALLOCATE(dla_vect, dla_mat)


    END SUBROUTINE producteur
```

Unité vecteur_print.f90 :

```
    !PALM_UNIT -name vecteur_print\
    !          -functions {F90 vecteur_print}\
    !          -object_files {vecteur_print.o}\
    !          -comment {vecteur_print}
    !
    !
    !PALM_OBJECT -name vecteur\
    !            -space NULL\
    !            -intent IN\
    !            -comment {vecteur 1d}


    SUBROUTINE vecteur_print

      USE palmlib             ! interface PALM
      USE palm_user_param     ! constantes de Prepalm

      IMPLICIT NONE

      CHARACTER(LEN=PL_LNAME) :: cl_object, cl_space

      DOUBLE PRECISION, ALLOCATABLE :: dla_vect(:)
      integer ::  i, il_err, il_shape

    !l'espace a ete declare NULL, pour savoir quel est l'espace herite
    ! il faut appeler la primitive Object_get_spacename
      cl_object = 'vecteur'
      call PALM_Object_get_spacename(cl_object, cl_space, il_err)


    ! on peut maintenant connaitre la taille de l'espace
    !( on sait que le rang est 1)
      CALL PALM_Space_get_shape(cl_space, 1, il_shape, il_err)
      ALLOCATE(dla_vect(il_shape))



    ! reception du vecteur
      cl_space  = 'NULL'
      cl_object = 'vecteur'
      CALL PALM_Get(cl_space, cl_object, PL_NO_TIME, PL_NO_TAG, dla_vect,
il_err)

    ! ecriture
```

```
    WRITE(PL_OUT,*) ' '
    WRITE(PL_OUT,*) 'Vecteur_print recu le vecteur :'
    WRITE(PL_OUT,*) (dla_vect(i), i=1,il_shape )

    DEALLOCATE(dla_vect)

  END SUBROUTINE vecteur_print
```

Exercice 13 bis
Créez une seconde branche START_OFF, déplacez l'unité vecteur_print sur cette branche et lancer la seconde branche dans la première, au début de la boucle, juste avant le lancement de producteur.

Si vous lancez l'application elle doit se terminer par un PALM_Abort avec le message suivant dans palmdriver.log :

```
Error in if_space_give_shape : the shape of the dynamic space
vect1d.producteur has probably not yet been set.
```

Synchronisez l'application pour faire en sorte que l'espace soit définit avant son utilisation dans vecteur_print. Vous pouvez synchroniser soit avec un "Step barrier ON" soit par une communication.

9.1 Rappel des points de cette session

Dans cette session vous avez appris comment décrire des espaces dynamiques, dont vous ne connaissez pas la taille avant de démarrer l'application. Coté envoi vous devez préciser cette taille à PALM, par la pritmitive PALM_Space_set_shape, coté réception, il faut déclarer à NULL l'espace de l'objet, demander quel espace a été associé (PALM_Object_get_spacename) et quelle est la taille de cet espace (PALM_Space_get_shape) avant l'appel de la primitive PALM_Get.

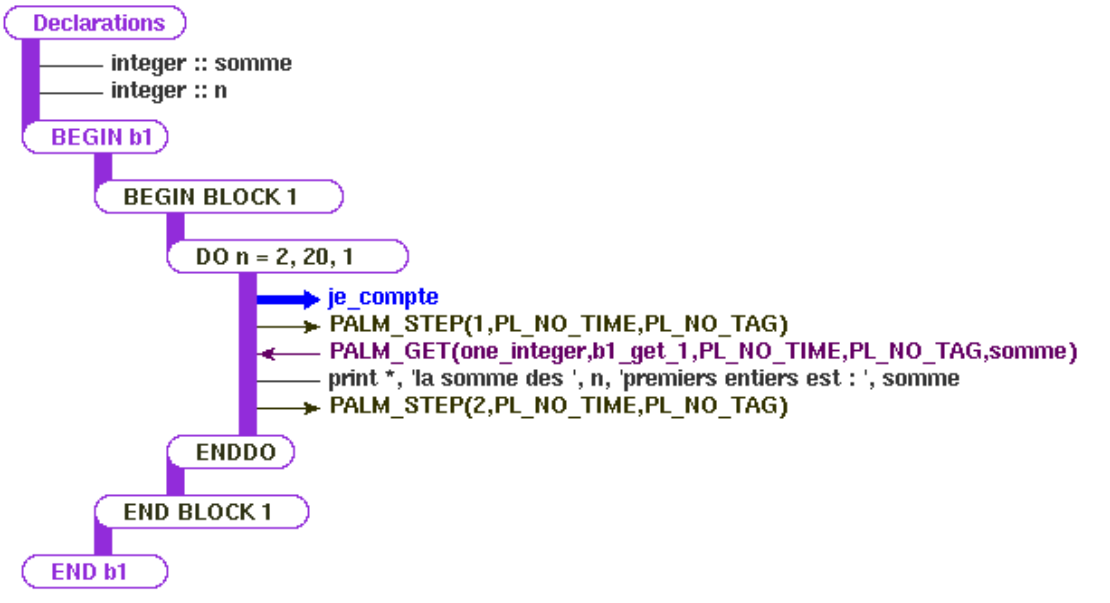
10 Session 10 : Composition d'objets dans le BUFFER

Nous avons vu l'intérêt du BUFFER de PALM pour conserver des objets dans une mémoire permanente et pouvoir ainsi les interpoler ou les récupérer à tout moment de l'application. Un autre rôle du BUFFER est la composition d'objets, par exemple pour calculer des moyennes ou des sommes d'objets.

L'unité je_compte.f90, que vous trouverez dans le répertoire session_10, produit en boucle les entiers de 1 à n, n étant une entrée de l'unité. Si on donne 4 en entrée de je_compte elle produit 1 2 3 et 4. Nous allons tout simplement faire la somme des entiers produits par cette unité et faire afficher le résultat par la branche. Pour avoir les 20 sommes des entiers de 2 à 20, nous relancerons à chaque fois l'unité je compte en recommençant la somme à 0, ceci nous permettra d'illustrer la remise à zéro d'un objet dans le BUFFER avec STEPLANG.

Composez !

- Créez la branche ci dessous :



- Dans le canevas de PrePALM définissez les communications
  - l'entrée n de je_compte prend la valeur de l'indice de boucle (click droit)
  - la sortie des entiers va dans le BUFFER avec un nom d'objet « somme » , cochez le champ add de Palm algebra et entrez 1 et 1 pour les coefficients
  - le PALM_get de branche récupère cet objet du BUFFER
- un objet en cours de composition dans le BUFFER est supposé être non prêt, avec l'aide de steplang (Help=> Help on steplang grammar) écrivez le script qui permet de rendre l'objet « ready » sur le step 1 et de le remettre à zéro sur le step 2.
- Testez

N’oubliez pas cette fonctionnalité de PALM, elle permet souvent d’éviter d’avoir recours à une unité d’algèbre prédéfinie ou à une unité utilisateur. Elle peut être utile pas exemple pour changer l’unité physique d’un champ échangé entre deux unités PALM qui n’auraient pas le même système de mesure.

10.1 Rappel des points de cette session

Vous avez appris dans cette session à utiliser une autre fonctionnalité de PALM qui est la composition d'objets dans le BUFFER, pour des objets de même espace, vous pouvez faire une combinaison linaire entre ce que vous ajoutez et ce qui s'y trouve déjà.
Pour éliminer les problèmes de synchronisation dus au parallélisme, un objet en cours de composition a le drapeau "non-prêt", pour pouvoir le récupérer il est nécessaire de faire appel au langage steplang pour le rendre "prêt".

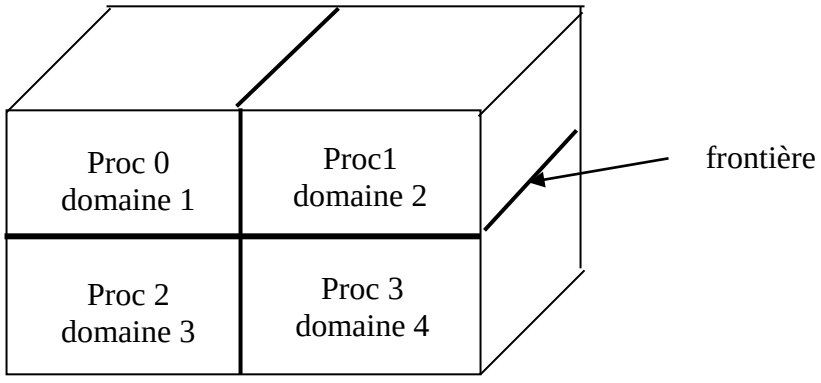
11 Session 11 : Les communications parallèles

11.1 Généralités

Le logiciel PALM est conçu pour gérer des unités parallèles. Qui dit programme parallèle, dit généralement distribution des données (c'est-à-dire comment sont réparties les données sur les différents processus occupés par le programme parallèle). Nous allons maintenant voir comment PALM peut gérer ces données distribuées pour faciliter l'échange entre les unités.

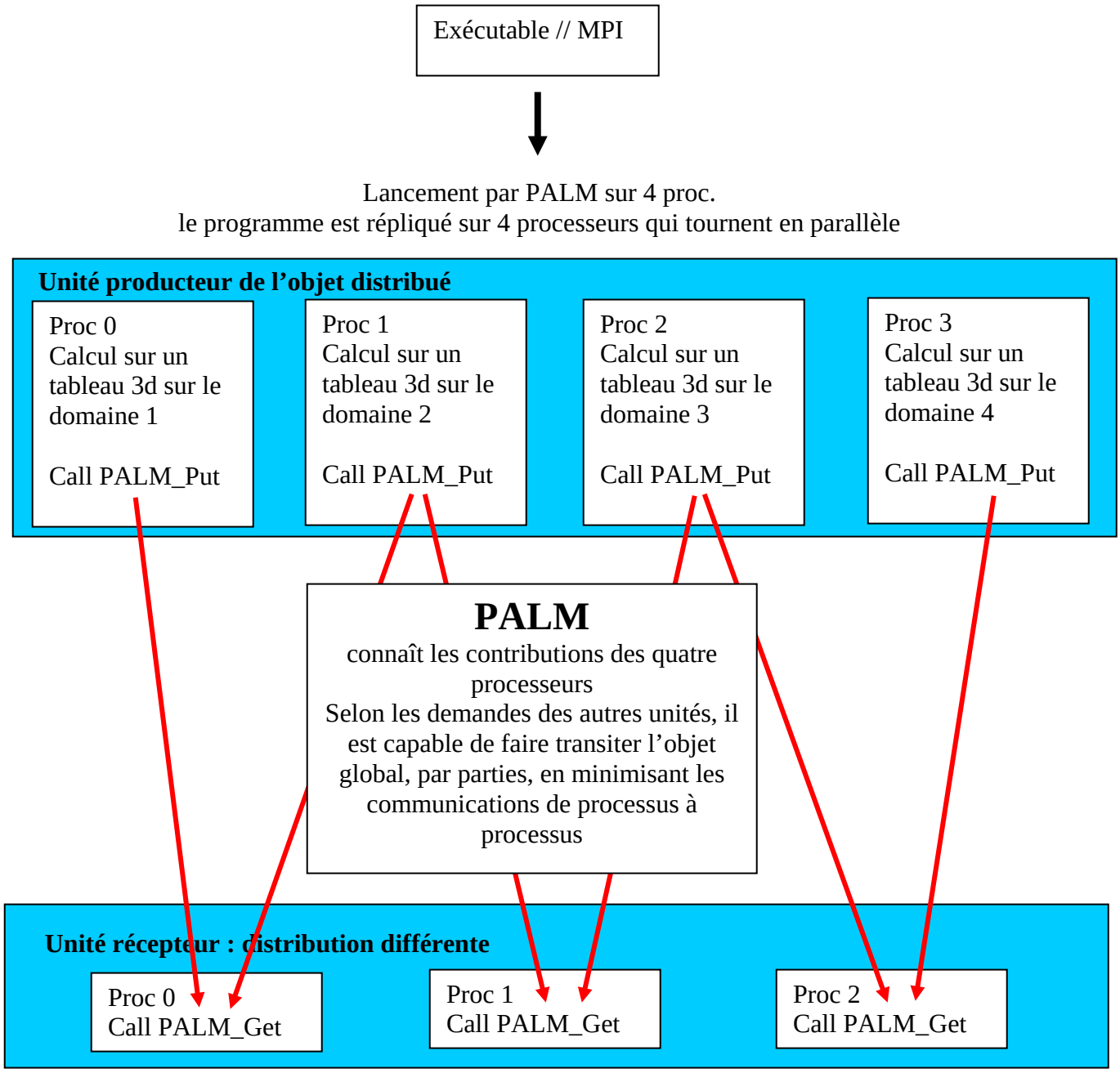
Tout d'abord quelques notions de calcul parallèle sont indispensables pour cette session. Nous parlons ici de parallélisme au sens décomposition de domaines, protocole SPMD (Single Program Multiple Data). Pour traiter un tout, le même programme est répliqué sur plusieurs processeurs (ou processus) qui se partagent le problème à traiter, et selon une stratégie qui est décidée par le programmeur. Ce type de programmation n'est possible qu'avec l'aide d'une bibliothèque de calcul parallèle comme MPI. À l'exécution, chaque processus se spécialise pour traiter une partie du problème. Notez que la décomposition du domaine n'est pas faite par PALM, mais que ce dernier va s'appuyer dessus pour gérer l'objet dans sa totalité.

Pour être concret, imaginons un programme de circulation océanique où l'espace 3d est discrétisé en différences finies sous forme de mailles élémentaires. Ce type de programme manipule en mémoire des tableaux 3d (correspondants à une discrétisation de l'espace physique) qui couvrent l'ensemble de la Terre. Pour gagner en rapidité de calcul, ou simplement pour traiter une taille de problème qui ne tient pas sur la mémoire d'un seul processeur, le code est parallélisé. Chaque processus traite une partie seulement du tableau virtuel 3d sur lequel il fait ses calculs. En général, avec ce type de parallélisme, on est amené à s'échanger à chaque pas de temps des informations aux frontières de chaque domaine pour pouvoir continuer le processus d'itération. Tout ceci n'empêche pas de pouvoir considérer le tableau global 3d comme un objet PALM, même si physiquement il est distribué sur la mémoire de plusieurs processeurs.



Exemple de découpe d'un domaine 3D sur 4 processeurs

Chaque processeur ne connaît qu'une partie du tableau. Les tableaux locaux à chaque processeur ne sont pas forcément de même taille en mémoire. Pour ne pas avoir à regrouper le tableau sur un seul processeur avant son envoi, PALM offre la possibilité que chaque processeur fasse un PALM_Put de seulement la partie du domaine qu'il connaît.



Pour que PALM puisse gérer de tels échanges, on voit qu'il est nécessaire de décrire de quelle manière les objets sont distribués coté source et coté cible, condition nécessaire et suffisante, c'est le rôle des distributeurs.

11.2 Les distributeurs

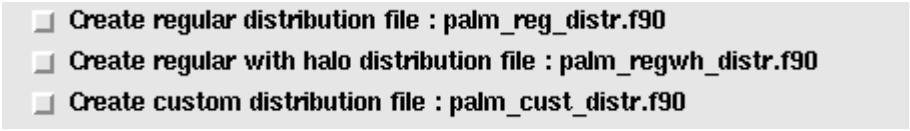
Les distributeurs ne sont rien d'autre que la manière de faire connaître à PALM comment les objets sont distribués, donc comment le code à été parallélisé, l'information pertinente est : « quelle est la partie de l'objet global gérée par tel ou tel processus ». Pour que PALM puisse reconnaître ces

informations il faut les lui donner avec une syntaxe déterminée. Pour apporter toute la souplesse nécessaire, PALM offre la possibilité de décrire les distributeurs de deux façons différentes plus ou moins adaptées selon le cas de figure.

Les distributeurs de type **regular** permettent de décrire un motif de base qui se répète à l'intérieur du tableau global. Ces distributions, directement inspirées des découpages utilisés par les bibliothèques de calcul scientifique parallèles comme SCALAPACK, sont très concises mais en général peu adaptées aux codes de calcul.

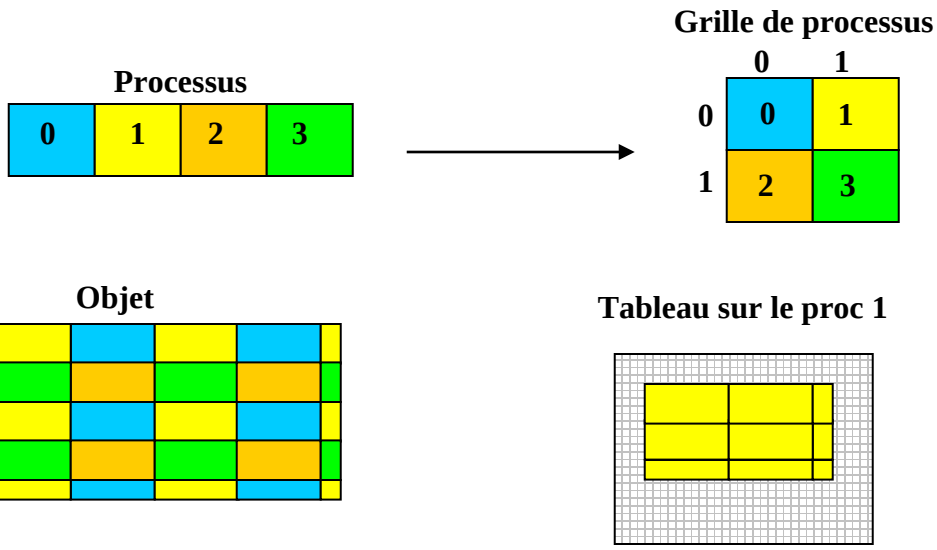
Les distributeurs de type **custom** sont moins concis mais permettent de décrire tout type de distribution.

C'est dans la carte d'identité que les objets distribués doivent préciser quelle distribution ils utilisent. Bien qu'il soit possible de décrire le distributeur directement dans la carte d'identité de l'unité sous forme d'une liste d'entiers ou de constantes, il est vivement conseillé de le décrire dans une fonction de distribution. PrePALM vous créera un squelette de fonction de distribution (subroutine FORTRAN 90) en cochant une des cases dans la boite de dialogue apparaissant avec le menu « Make PALM files »



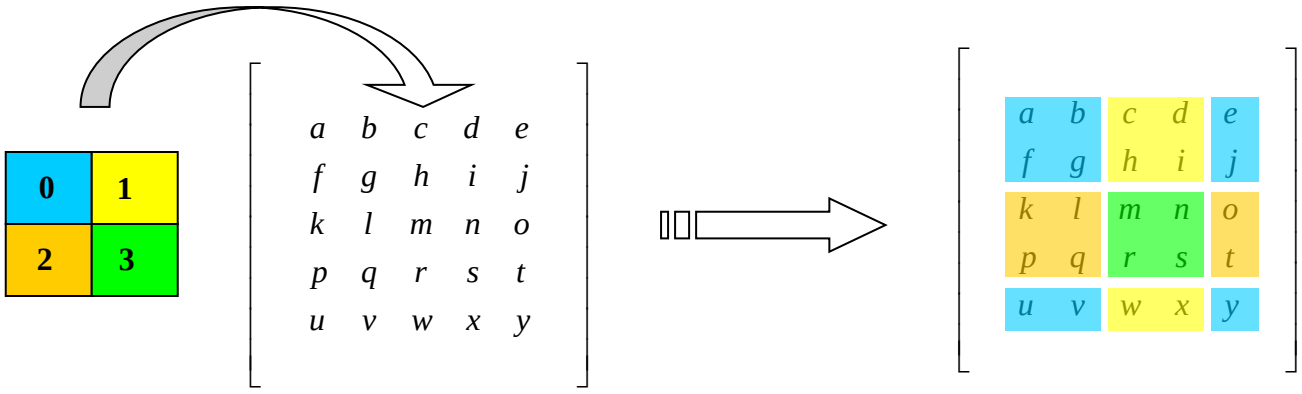
11.3 Distributeur de type "cyclique par blocs"

Cette distribution est directement inspirée des distributions utilisées dans la bibliothèque de programmes SCALAPACK. Dans PALM, elles sont appelées "regular distributions".
Les processus d'une distribution sont organisés selon une grille multidimensionnelle, **ayant le même nombre de dimensions que l'objet global**. L'objet global est découpé en blocs de manière régulière, n_i éléments par bloc dans chaque dimension *i*.
Si la taille de l'objet global dans une dimension n'est pas un multiple exact de la taille des blocs dans cette dimension, alors le dernier bloc est plus petit que les autres.
Exemple d'un objet 2D distribué sur une grille de 2x2 processus :



Pour définir l'ordre dans lequel les blocs sont affectés aux processus, on numérote cycliquement les blocs, dans chaque dimension, dans une fourchette correspondant à la taille de la grille de processus dans cette dimension. On est libre de choisir le numéro à partir duquel on commence à compter. Dans l'exemple ci-dessus, on a choisi (0,1) couleur jaune.
Les blocs sont stockés dans l'objet local de façon contiguë. L'objet local peut être plus grand que l'ensemble des blocs.

Plus concrètement, prenons une matrice 2D de taille 5x5 que l'on souhaite distribuer sur la grille des 4 processus précédente pour laquelle on commence à compter à (0,0) :



Nous obtenons donc la distribution suivante :

Tableau sur le processus 0 : $\begin{bmatrix} a & b & e \\ f & g & j \\ u & v & y \end{bmatrix}$

Tableau sur le processus 1 : $\begin{bmatrix} c & d \\ h & i \\ w & x \end{bmatrix}$

Tableau sur le processus 2 : $\begin{bmatrix} k & l & o \\ p & q & t \end{bmatrix}$ Tableau sur le processus 3 : $\begin{bmatrix} m & n \\ r & s \end{bmatrix}$

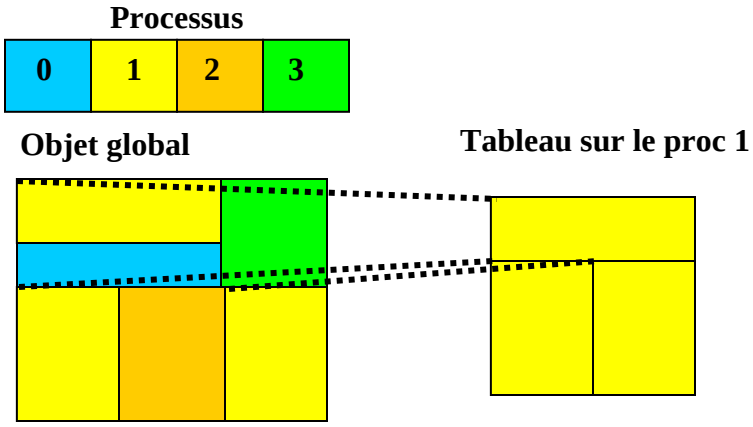
Pour PALM, les informations nécessaires pour décrire une telle distribution sont les suivantes:

1. La forme de la grille de processus (c.a.d. sa taille dans toutes les dimensions) : (2, 2) dans l'exemple ci-dessus
2. La taille d'un bloc élémentaire: (2,2) dans notre exemple
3. Les coordonnées dans la grille du processus qui va contenir le premier bloc de l'objet global: (0,0) dans l'exemple ci-dessus
4. Pour chaque processus:
    1. la forme de l'objet local
    2. les coordonnées dans le tableau local du premier élément du premier bloc

Cette manière de décrire les objets distribués est très concise mais ne permet pas de représenter tout type de distribution.

11.4 Distributeur de type 'CUSTOM'

Cette méthode de distribution est la plus souple: pour chaque processus, on liste les blocs qui y sont stockés et leur place dans l'objet local. Cette méthode permet de définir n'importe quelle distribution. En particulier, elle est bien adaptée à la décomposition de domaine utilisant des grilles non structurées parce que les blocs peuvent être placés arbitrairement dans l'objet local.
Exemple :



Les informations nécessaires pour décrire une telle distribution sont les suivantes:

Pour chaque processus

1. le nombre de block
2. la forme de l'objet local
3. Pour chaque bloc qui y est stocké
    1. la forme du bloc
    2. les coordonnées du premier élément du bloc dans l'objet global

3. les coordonnées du premier élément du bloc dans l'objet local

11.5 Exemples d'objets distribués

Dans le répertoire session_11, ouvrez avec un éditeur de texte le fichier toy_ocean.f90. Répondez aux questions suivantes en regardant la carte d'identité de l'unité :

- L'unité est-elle parallèle et de quel type ?
- Sur combien de processus l'unité orca_toymodel peut-elle fonctionner ?
- Combien d'objets sont définis (IN et OUT) ?
- Combien d'objets sont distribués ?
- Quel est le distributeur associé à l'objet field ?
- Quelle est le rang de l'objet et la taille globale de l'objet distribué ?

En regardant plus précisément le distributeur :

- Quel est le type du distributeur ?
- Sur combien de processus le distributeur porte-t-il ?
- Dans quel fichier se trouve la fonction de distribution ?

A partir du code Fortran :

- Quelle est la variable qui va contenir l'objet local distribué ?
- Pourquoi cette variable est-elle allouée dynamiquement ?
- Quelle est la taille de cette variable ?
- Par quelle subroutine cette taille est-elle déterminée et en fonction de quels paramètres ?

Ouvrez maintenant la fonction de distribution qui se trouve dans le fichier ocean_distrib.f90. Remarquez que cette fonction, dont le squelette a été généré par PrePALM, comporte deux modes d'appel selon l'argument id_action. Cette fonction, n'est pas appelée par vous dans les unités, mais uniquement par PALM. Le premier mode de fonctionnement permet de retourner la taille du tableau qui va contenir le distributeur (ceci est utile pour PALM pour allouer dynamiquement le tableau de travail qui contiendra le distributeur), le second mode permet de retourner un vecteur d'entier contenant le distributeur.

Dans notre exemple, remarquez que la fonction de distribution utilise la même subroutine que celle utilisée par l'unité toy_ocean (my_domain). Cette fonction détermine la découpe du domaine sur chaque processeur. Il faut bien noter que si vous avez à écrire une distribution, celle-ci ne s'invente pas : elle dépend entièrement de la manière utilisée pour paralléliser le code. Donc écrire une fonction de distribution revient souvent à faire du copier/coller de la partie qui parallélise le code (qui détermine les domaines) et à organiser ces données dans un ordre lisible par PALM.

Distribuez !

- Dans le répertoire session_11, lancez PrePALM
- Ajouter les trois constantes suivantes :

ip_nlon_ocean	182
ip_nlat_ocean	149
ip_nbproc_ocean	8

- Chargez les deux unités toy_ocean.f90 et plot_tcl.f90
- Lancez ces deux unités dans deux branches différentes
- Donnez le bon nombre de processus à l'application
- Faites fonctionner la boucle interne du modèle de 0 à 100 par pas de 20 (6 temps) en donnant des valeurs en dur pour les entrées min_time, max_time et freq_time
- Ajoutez une boucle (0 à 100 par pas de 20) autour de l'unité plot_tcl
- Envoyez les objets lon lat et msk produites par toy_océan dans le BUFFER car elles ne sont produites qu'une seule fois par toy_ocean, mais l'unité plot_tcl en a besoin à chaque fois qu'elle est lancée.
- Envoyez le champ produit à l'unité plot_tcl, n'oubliez pas de remplir le champ time, le trait épais matérialise une communication parallèle
- Pour plot_tcl :
  - donnez l'indice de boucle pour le champ time
  - ip_nlon_ocean pour nlon
  - ip_nlat_ocean pour nlat
  - pour lon, lat et mask, récupérez ces objets dans le BUFFER
  - entrez 3000 pour le champ refresh
- Dans la branche de plot, après le lancement de l'unité, insérez le script suivant : wish plot.tcl
- Testez

Exercice 15

Faites travailler le modèle sur un seul pas de temps, et modifiez le code source de toy_ocean pour que les champs produits dépendent du processus (on peut par exemple forcer les champs à la valeur du numéro de processus) et ainsi faire ressortir la décomposition du domaine. Vous pouvez mettre –1 pour l'objet refresh de l'unité plot_tcl, ce qui permettra de garder le dessin à l'écran, dans ce cas ajouter aussi le caractère & (lancement asynchrone) à la fin de la commande wish plot.tcl ce qui permettra à l'application PALM de terminer avant de refermer le dessin.

Testez différentes valeurs pour le nombre de processus de l'unité toy.

Dans notre exemple, une seule des deux unités est parallèle, l'objet échangé n'est distribué que du côté de toy_ocean. Comme nous l'avons vu, il est possible d'échanger des objets distribués côté source et côté cible avec des distributions identiques ou différentes, tout est possible avec PALM ! Le fait qu'une unité soit parallèle ne change rien à l'algorithme de couplage, il est donc très aisé,

pour gagner du temps dans une application PALM, de ne paralléliser que les unités qui prennent le plus de temps.

Bien que la fonction de distribution soit suffisamment générique dans notre cas pour fonctionner avec un nombre variable de processeur, les objets distribués ne peuvent pas, dans la version PALM_MP 3.0.2, être dynamiques, cette possibilité sera développée par la suite.

11.6 Les localisations et les associations de processus

L'exemple que nous venons de faire tourner est assez simple, mais on peut facilement tomber sur des cas plus complexes. Par exemple un code parallèle peut très bien décomposer son domaine sur moins de processeurs que le nombre sur lequel il tourne. De même, certains objets peuvent ne pas être distribués, mais simplement être répliqués sur tous les processus. Toutes ces caractéristiques doivent être communiquées à PALM. Ceci se fait encore via les cartes d'identités (en effet, les localisations sont attachées aux objets et sont donc définies dans la carte d'identité) et dans l'interface graphique lors de la création des communications.

Concentrons nous tout d'abord sur les informations à préciser au niveau des cartes d'identités. Si vous reprenez celle de l'unité toy_ocean de l'exemple, vous vous rendrez-compte que certains objets ont un champ "–localisation" dans leur description :

```
!PALM_OBJECT -name freq_time\
!            -space one_integer\
!            -localisation REPLICATED_ON_ALL_PROCS\
!            -intent IN\
!            -comment {frequence}

!PALM_OBJECT -name field\
!            -space one_matrix\
!            -intent OUT\
!            -distributor ocean_distrib\
!            -localisation DISTRIBUTED_ON_ALL_PROCS\
!            -time ON\
!            -comment {Champs calcules}
```

Les localisations permettent de préciser deux choses pour les objets des unités parallèles :

1) faire la différence entre des objets distribués ou répliqués
2) préciser sur quels processeurs les distributeurs s'appliquent, et dans quel ordre.

Il existe deux manières de définir les localisations :

1) Utiliser les localisations prédéfinies (comme pour les deux objets ci-dessus) :

-DISTRIBUTED_ON_ALL_PROCS : l'objet est distribué sur tous les processus de l'unité

-REPLICATED_ON_ALL_PROCS : l'objet est répliqué sur tous les processus de l'unité

-SINGLE_ON_FIRST_PROC (c'est la localisation par défaut, utilisée lorsque le champs "-localisation" n'est pas spécifié): l'objet n'est ni distribué, ni répliqué, il n'a qu'une seule instance localisée sur le processus 0 de l'unité (c'est le cas par exemple pour tous les objets des unités non-parallèles).

L'utilisation d'une de ces trois localisations doit être spécifiée lors de la définition de l'objet concerné, dans le champs "–localisation".

2) Si la localisation d'un objet est différente des localisations prédéfinies, il faut la décrire explicitement en utilisant le mot clé `PALM_LOCALISATION`. Par exemple, supposons qu'un objet soit distribué sur les processus 0, 4, 3 et 2 alors que l'unité tourne sur 5 processus, il est nécessaire de passer par une localisation du type :

```
!PALM_LOCALISATION  -name nom_de_la_localisation\
!                   -type distributed \
!                   -description {0;4;3;2}
```

Le champs "-name" permet de nommer cette localisation. Dans la définition de l'objet concerné par cette localisation, il faut spécifier ce nom dans le champs "–localisation". Le champs "-type" peut être soit "distributed", pour un objet distribué, soit "replicated" pour un objet répliqué. Enfin, le champs "-description" permet d'identifier les processus concernés. La syntaxe de la description est la même que pour les listes de temps.

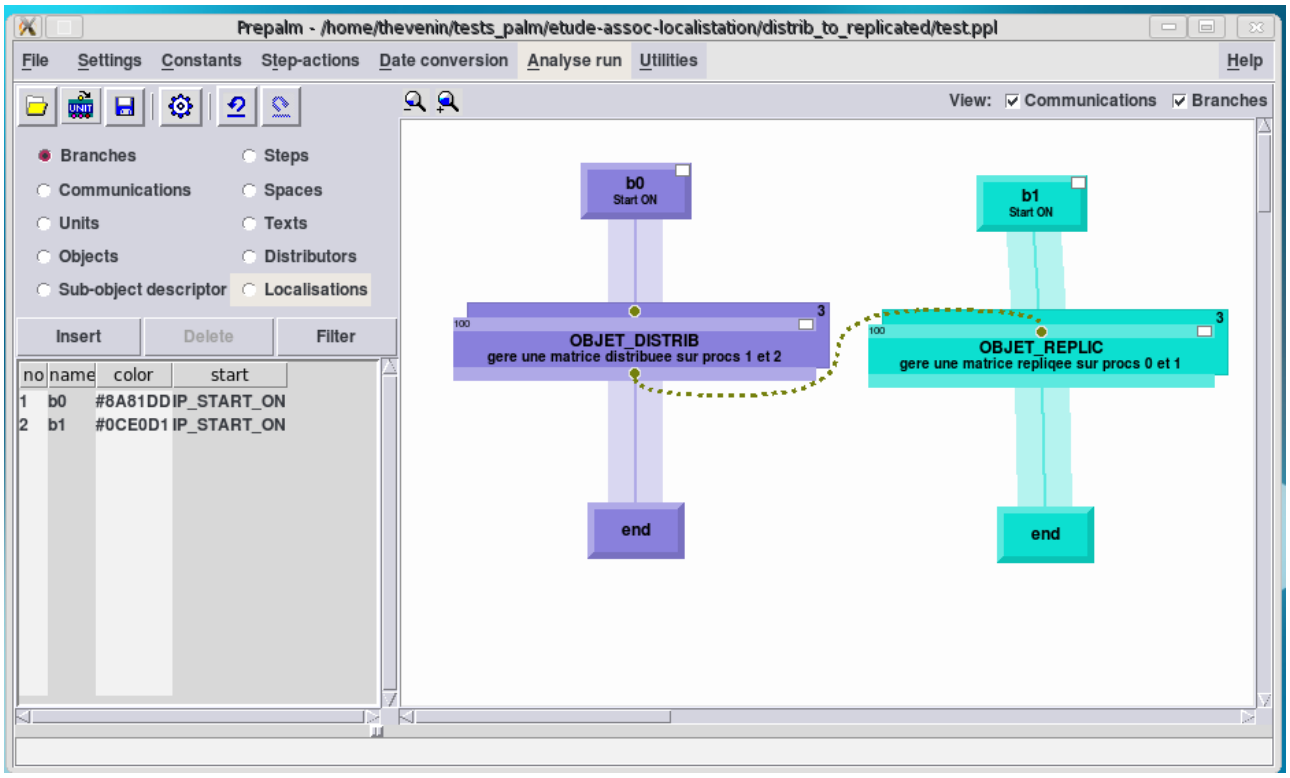
Une fois les localisations définies dans la carte d'identité, lors de la création des communications, il faudra remplir le champs "local. assoc." dans la fenêtre concernant les propriétés des communications. Il faut alors distinguer le cas d'un objet distribué d'un objet répliqué.

Un objet est dit distribué lorsqu'il est réparti sur plusieurs processus d'une unité, qui traitent chacun une partie locale de cet objet. Le distributeur de l'objet décrit la façon dont est découpé l'objet (nombre de processus sur lesquels est distribué l'objet, taille et coordonnées des blocs définis dans l'objet global, tailles des tableaux locaux contenant ces blocs, …). Dans ce cas, la localisation décrit la liste des processus de l'unité sur lesquels est distribué l'objet, c'est à dire les numéros des processus de l'unité qui vont gérer les parties locales de l'objet décrites dans le distributeur. Dans la fenêtre des propriétés des communications, au niveau des associations, il faudra simplement spécifier **le rang (numéro) du 1^{ier} processus qui fait parti de la distribution**.

Un objet est dit répliqué si plusieurs processus de l'unité auquel il appartient gèrent de façon indépendante une instance de cet objet. Dans ce cas, la localisation décrit la liste des processus de l'unité qui gèrent une instance de cet objet. Dans la fenêtre des propriétés des communications, il faudra indiquer **les rangs des processus faisant partie de la localisation** associée à cet objet. La syntaxe est alors la suivante : **deb1**[**:fin1**[**:stp1**]] [| **deb2** [**:fin2**[**:stp2**]]] [; …]

Exemple :

Considérons deux unités parallèles qui échangent un objet. Les deux unités tournent sur trois processus. Pour la première unité (OBJET_DISTRIB), l'objet est distribué sur les processus 1 et 2 tandis que sur la deuxième (OBJET_REPLIC), l'objet est répliqué sur les processus 0 et 1 :



Concernant cet objet, nous avons dans la carte d'identité de l'unité OBJET_DISTRIB :

```
/*PALM_LOCALISATION -name loc\
                    -type distributed\
                    -description {1;2}
*/

/*PALM_OBJECT -name field\
              -space matrice\
              -distributor distr\
              -localisation loc\
              -intent INOUT\
              -comment {matrice distribuee}
*/
```

Au niveau du "PALM_OBJECT", nous avons le mot clé "-distributor" qui fait référence à la distribution de cet objet (définie dans PALM_DISTRIBUTOR) et nous avons aussi le mot clé "-localisation" qui se réfère à la localisation "loc" qui est de type distribuée, sur les processus 1 et 2.

Pour l'unité OBJET_REPLIC :

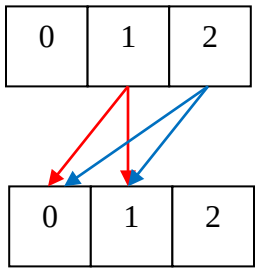
```
/*PALM_LOCALISATION -name loc\
                    -type replicated\
                    -description {0;1}
*/
```

```
/*PALM_OBJECT -name field\
              -space matrice\
              -localisation loc\
              -intent IN\
              -comment {matrice distribuee}
*/
```

Au niveau du "PALM_OBJECT", nous avons bien le mot clé "-localisation" qui fait référence à la localisation "loc" qui est de type répliquée, sur les processus 0 et 1.

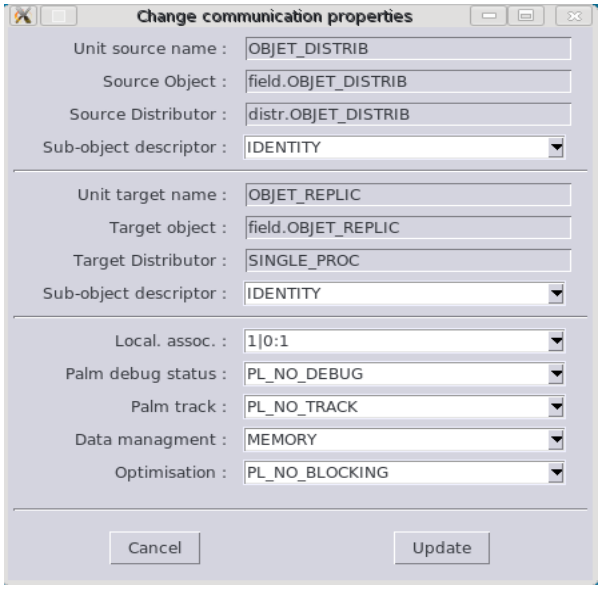
Concernant la communication, nous avons donc l'unité OBJET_DISTRIB qui envoie un objet distribué sur les processus 1 et 2 vers l'unité OBJET_REPLIC dont l'objet est répliqué sur les processus 0 et 1 (Et tout ça avec un unique duo PALM_Put / PALM_Get !!) :

Unité OBJET_DISTRIB :



Unité OBJET_REPLIC :

Au niveau de la définition de la communication dans PrePALM, il convient de décrire précisément la manière dont les instances de l'objet sont transmises entre l'unité source et cible. Nous avons donc l'association suivante (champs local. assoc.) :



<u>Pour les objets répliqués</u>, dans la plupart des cas, l'association peut être déduite des localisations, il suffit donc de sélectionner l'association AUTOMATIC proposée par défaut par PrePALM. Dans ce cas, l'association est traitée comme décrit dans le tableau ci dessous :

Source	Target	Association

SINGLE_ON_FIRST_PROC	SINGLE_ON_FIRST_PROC	L'objet, non distribué est envoyé du proc 0 au proc 0 Equivalent à assoc : 0
SINGLE_ON_FIRST_PROC	DISTRIBUTED_ON_ALL_PROCS	Chaque partie de l'objet, non distribué coté source, est envoyé aux différents processus coté cible Assoc : 0
SINGLE_ON_FIRST_PROC	REPLICATED_ON_ALL_PROCS	L'objet, non distribué coté source, est envoyé entièrement à tout les processus coté cible. Assoc : 0 | 0 : nbproc_tgt-1
DISTRIBUTED_ON_ALL_PROCS	SINGLE_ON_FIRST_PROC	L'objet, distribué sur tous les processus coté source est envoyé au processus 0 coté cible Assoc : 0
DISTRIBUTED_ON_ALL_PROCS	DISTRIBUTED_ON_ALL_PROCS	L'objet est distribué des deux cotés sur tous les processus. Assoc : 0
DISTRIBUTED_ON_ALL_PROCS	REPLICATED_ON_ALL_PROCS	L'objet, distribué coté source, est reconstitué puis envoyé à tous les processus coté cible. Assoc : 0 | 0 : nbproc_tgt-1
REPLICATED_ON_ALL_PROCS	SINGLE_ON_FIRST_PROC	Non traité
REPLICATED_ON_ALL_PROCS	DISTRIBUTED_ON_ALL_PROCS	Non traité
REPLICATED_ON_ALL_PROCS	REPLICATED_ON_ALL_PROCS	Chaque processus, coté source, envoie son objet au processus de même rang coté cible Assoc : 0 :nbproc_src-1

Rassurez-vous, même si PALM envisage ce cas de figure pour pouvoir traiter tous type de communication parallèle, il est très rare d'avoir à décrire des localisations différentes de celles prédéfinies.

11.7 Rappel des points de cette session

Dans cette session vous avez découvert comment PALM traite les communications entre des unités parallèles si les données sont distribuées sur plusieurs processus.
Vous avez vu comment les objets sont généralement répartis en mémoire sur les différents processus. Ceci nous a permis d'introduire la notion de distributeur; i.e. la syntaxe pour décrire des

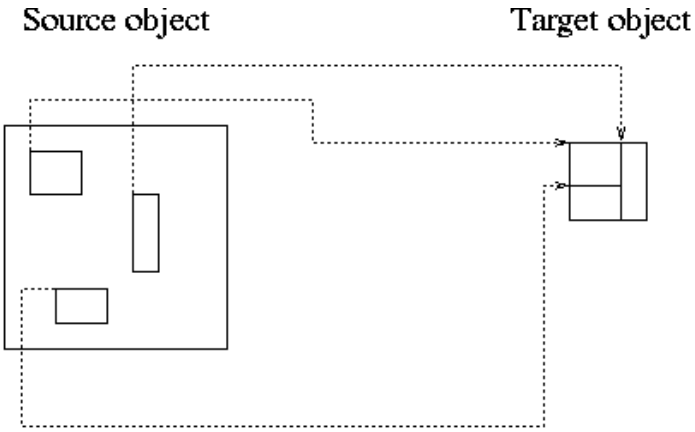
distributions. Pour un maximum de souplesse, PALM permet de manipuler différents types de distributions : régulières ou custom.
Finalement vous avez appris à utiliser la notion de localisation pour décrire sur quel sous-ensemble de processus un objet peut être distribué ou répliqué. Au niveau de la communication tout ceci se combine avec la notion d'association de processus pour garantir les bons échanges d'informations.

12 Session 12 : Les sous-objets

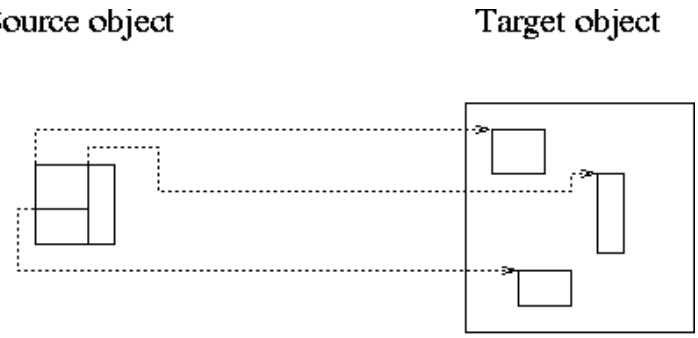
Les sous-objets ont été introduits dans PALM_MP pour assurer un niveau d'indépendance encore plus important entre les unités. Il s'agit, par exemple de ne récupérer qu'une partie d'un objet dans une unité cible sans avoir à modifier le code de l'unité source. On utilise à cet effet des descripteurs de sous objets. Un sous-objet est vu comme un ensemble de sous-blocs de l'objet auquel il appartient.

Cette fonctionnalité permet de :

- récupérer dans une unité cible une partie seulement d'un objet produit par une unité source,



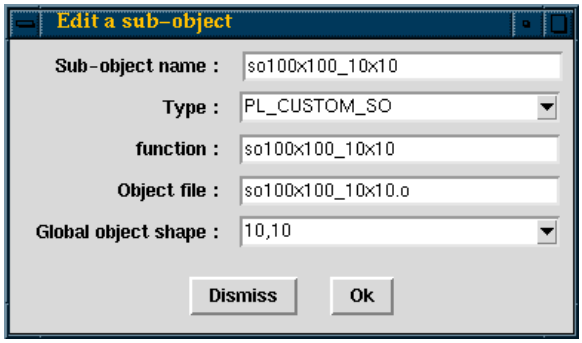
- mettre à jour dans une unité cible seulement une partie d'un objet lors d'un `PALM_Get`,



Les sous-objets doivent être définis dans PrePALM et non dans les unités. En effet, ils sont complètement dépendants de l'application dans laquelle ils sont utilisés, c'est pourquoi ils ne sont pas définis dans la carte d'identité d'une unité.

Pour définir un sous-objet, l'utilisateur doit cocher la case **Sub-object descriptor** dans le Sélecteur puis cliquer sur le bouton **Insert**. Une fenêtre apparaît. L'utilisateur doit saisir le nom du sous-

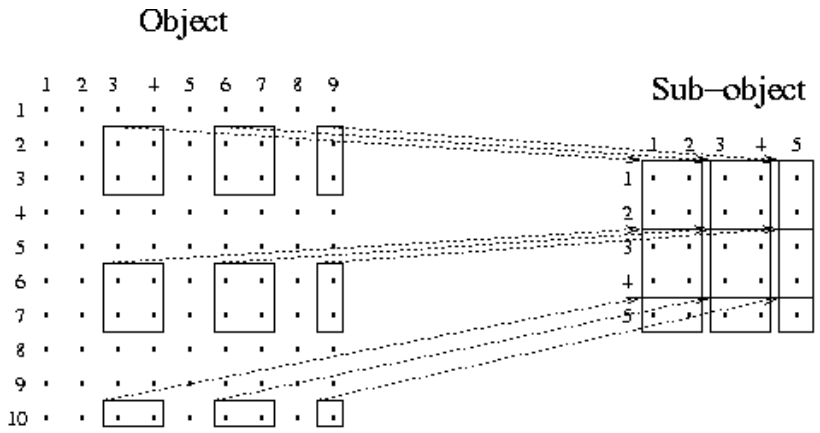
objet, son type, le nom de la fonction qui le décrit, le nom du fichier compilé qui contient la fonction, et le shape de l'objet pour lequel il est défini (que l'on appellera par la suite objet global).



Les différents types de sous-objets correspondent aux différents modèles qui permettent de les décrire. Un sous-objet peut-être de type PL_CUSTOM_SO ou PL_REGULAR_SO. Nous retrouvons ici la même terminologie que pour les descripteurs de distributions. Le modèle REGULAR sera utilisé pour décrire facilement des sous-objets réguliers, et le modèle CUSTOM sera utilisé dans les autres cas.

Un sous-objet régulier est un sous-objet dont les blocs ont la même taille (sauf éventuellement pour les derniers dans chaque dimension), et sont régulièrement espacés dans l'objet global.

Pour décrire un sous-objet, l'utilisateur doit écrire une fonction adaptée au type du sous-objet qu'il a créé. Un template des différentes fonctions est fourni par PrePALM en cochant **Create regular sub-object file** ou **Create custom sub-object file** dans la fenêtre **Make PALM file**.



Example of a regular sub–object

Les fonctions de description des sous-objets sont construites dans le même esprit que celles des distributions. Il s'agit pour l'utilisateur de remplir un vecteur d'entiers avec les données qui permettent de décrire le sous-objet.

Pour décrire un sous-objet régulier, l'utilisateur doit fournir :

- le rang du sous-objet,
- son shape (taille dans chaque dimension),
- le shape du bloc élémentaire,
- le nombre de blocs dans chaque direction,
- la taille de l'espace entre les blocs dans chaque dimension,
- les coordonnées du coin supérieur gauche du bloc supérieur gauche du sous-objet dans l'objet global.

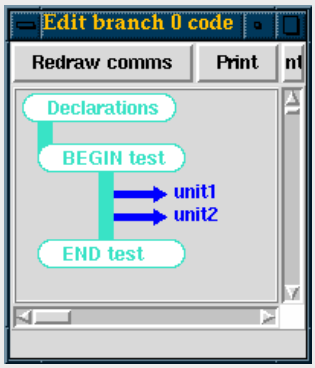
Pour décrire un sous-objet custom, l'utilisateur doit fournir :

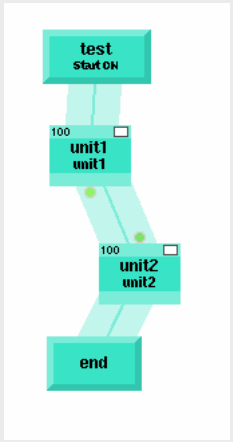
- le rang du sous-objet,
- son shape,
- le nombre de blocs du sous-objet,
- la description de chaque bloc (shape, coordonnées du coin supérieur gauche dans l'objet global et dans le sous-objet).

Lorsqu'un sous-objet est créé dans PrePALM, l'utilisateur peut s'en servir lors de la définition d'une communication en indiquant le nom des sous-objets source et cible de la communication (le sous-objet par défaut étant IDENTITY, ce qui signifie que le sous-objet est égal à l'objet).
Dans le répertoire session_12, vous trouverez deux unités. L'unité unit1.f90 produit un tableau 2d de 100x100 réels et fait un PALM_PUT de ces données. L'unité unit2.f90 fait un PALM_Get d'un tableau 10x10 de réels et imprime les éléments. Nous allons voir comment faire pour que unit2 récupère le centre du tableau produit par unit1.

Prenez un bout d'objet !

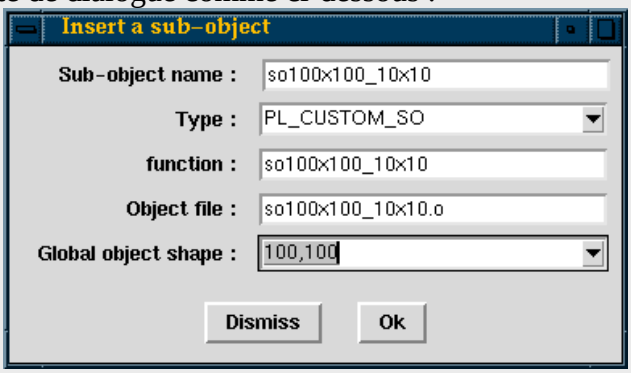
- Dans le répertoire session_12, lancez PrePALM
- Créez l'application suivante :





- A partir du menu File => Make Palm files, cochez la case create custom sub-object file et créez le fichier palm_cust_so.f90
- Renommez ce fichier en so100x100_10x10.f90
- Editez ce fichier
  Vous n'avez que trois lignes à modifier dans ce fichier :
  - le nom du subroutine, mettez **`so100x100_10x10`**
  - le nombre de blocks, ici c'est **1**
  - le vecteur d'entier contenant le descripteur :
    `ida_descr = (/10,10,  1, 10,10,45,45,1,1/)`

- Dans PrePALM sélectionnez la catégorie Sub-object descriptor en haut à gauche de la fenêtre principale, puis sur le bouton Insert juste en dessous
- Remplissez la boite de dialogue comme ci-dessous :



- vous n'avez plus qu'à créer la communication en utilisant le descripteur de sous objet coté source et à tester l'application

12.1 Rappel des points de cette session

Cette courte session vous a montré comment envoyer ou recevoir des portions d'objets sans refaire l'instrumentation du code. Les sous objets sont basés sur une description univoque des portions d'objets que l'on veut envoyer ou recevoir en précisant leur place dans les tableaux multidimensionnels. Cette description se fait dans des fonctions fortran prédéfinies qui dépendent de l'application.

13 Session 13 : Lire et écrire dans des fichiers, interpoler des champs géophysiques

Comme nous l'avons vu, les primitives PALM_Get/Put permettent respectivement aux unités PALM de demander des informations ou de les rendre disponibles, l'envoi ou la réception de données n'est effectif que lorsque des communications sont décrites (correctement !) dans PrePALM. En général ces données sont fournies ou expédiées vers d'autres unités de la même application, tout se passe en mémoire et les données manipulées par les primitives PALM_Get/Put sont perdues après l'exécution de l'application. Tant qu'à manipuler des données (dont les caractéristiques informatiques ont par ailleurs été décrites dans les cartes d'identités) il semble naturel de pouvoir les stocker de manière pérenne dans des fichiers. A l'inverse, on peut très bien imaginer que les unités qui demandent des données puissent les lire directement dans des fichiers sans passer par l'écriture d'une unité dédiée à cela. C'est le rôle des fichiers de PrePALM, l'idée est de pouvoir relier directement les plots correspondants aux Put/Get des unités à des fichiers. L'appel à un PALM_Get dans une unité déclenchera une lecture, l'appel à un PALM_Put une écriture.

Le format de fichier retenu est le standard NETCDF bien connu de la communauté du climat, qui offre plusieurs avantages :

- Le format est auto-descriptif, le fichier contient un entête qui décrit ce qui est contenu dans le fichier.
- Les accès sont directs, les enregistrements peuvent être lus/écrits dans n'importe quel ordre.
- Les données sont stockées de manière optimale en termes de place car elles sont écrites dans un format binaire, mais contrairement au binaire Fortran, ce format est portable d'une machine à l'autre. Le binaire NETCDF conserve la précision de la machine.
- La bibliothèque NETCDF est en général installée sur tous les calculateurs, dans le cas contraire, son installation est facile à réaliser.
- De nombreux logiciels de prés ou post traitement utilisent ce format.

Comme illustration des fichiers, reprenons le modèle jouet de la session 11, à chaque pas de temps, le modèle toy_ocean produit des champs 2d sur une grille de discrétisation spatiale relativement complexe (grille orca du fameux modèle du LOCEAN, c'est une grille structurée mais non régulière). Nous allons stocker ces champs dans un fichier pour ensuite les interpoler spatialement sur une grille différente (grille du modèle ARPEGE de Météo-France). Ce cas de figure correspondrait par exemple à un forçage de données de températures de surface de l'océan dans un modèle d'atmosphère.

La première chose à faire est de décrire le format du fichier qui va contenir ces champs de surface à chaque pas de temps. Pour PrePALM les fichiers NETCDF sont décrits comme les unités au travers des cartes d'identité.

Stockez !

- Ouvrez le fichier champs_orca.id, carte d'identité du fichier que nous allons manipuler
- Remarquez le mot clés PALM_FILE à la place de PALM_UNIT, remarquez également la rubrique –shape_label dans la définition des espaces, c'est une donnée supplémentaire à fournir par rapport aux unités dans la définition des espaces. Sinon tout le reste est identique, on retrouve les mêmes informations que dans les cartes d'identité des unités PALM.
- Avec PrePALM ouvrez maintenant l'application creation_fichier.ppl. Pour comprendre l'application répondez aux questions :
  - sur combien de processus tourne le modèle jouet ?
  - combien d'instances temporelles de l'objet field vont être produites ?
  - pourquoi la communication entre l'objet field et le fichier apparait sous forme d'un trait épais continu ?
  - pourquoi les autres communications apparaissent sous forme de traits fins en pointillés ?
  - quel est le nom du fichier netcdf qui va être créé ?

.

- Faites tourner cette application pour créer le fichier de sortie du modèle

L'interpolation spatiale des champs contenus dans ce fichier va maintenant être faite avec une boite prédéfinie que vous pouvez trouver directement dans PrePALM (menu file->load algegbra unit -> interpolation -> geophysic ->dscrip.alg). Cette unité a été développée à partir des routines d'interpolation du coupleur OASIS3 du CERFACS. Ce coupleur, qui est très utilisé dans la communauté du climat présente l'avantage (à partir de la version OASIS4) de permettre de déporter l'interpolation dans les exécutables parallèles des modèles à coupler, l'interpolation peut donc se faire en parallèle, ce qui n'est pas possible pour le moment avec PALM. En outre, le déploiement d'OASIS4 sur certaines machines est plus facile à mettre en œuvre car il ne nécessite pas MPI2, contrairement à PALM. Pour plus d'informations sur le fonctionnement de l'interpolation il convient de se reporter à la documentation du coupleur OASIS.

Nous avons déjà la carte d'identité du fichier que nous voulons interpoler, cependant elle a été décrite pour un fichier en écriture, les objets sont « -intent IN ». Nous voulons maintenant accéder à ce fichier en lecture, nous avons deux solutions pour redéfinir la carte d'identité de ce fichier. Soit nous copions le fichier champs_orca.in et nous remplaçons les « -intent IN » par des « -intent OUT », soit nous utilisons un utilitaire de PrePALM qui permet de créer les cartes d'identité de fichiers à partir de fichiers NETCDF existants. Vous n'avez pas à le faire pour les TP car pour vous faciliter la tâche ce travail a déjà été fait.

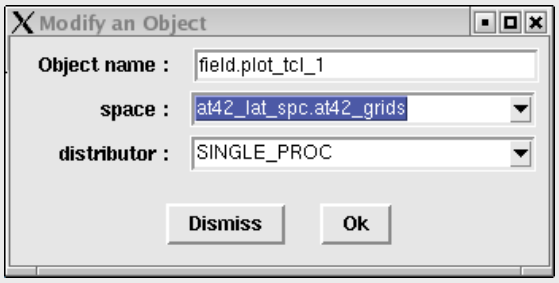
L'unité d'interpolation fonctionne comme cela :

- elle demande le type de grille, les grilles, les connectivités des deux grilles, l'une étant considérée comme la source et l'autre comme la cible,
- elle demande le champ à interpoler pour la grille source,

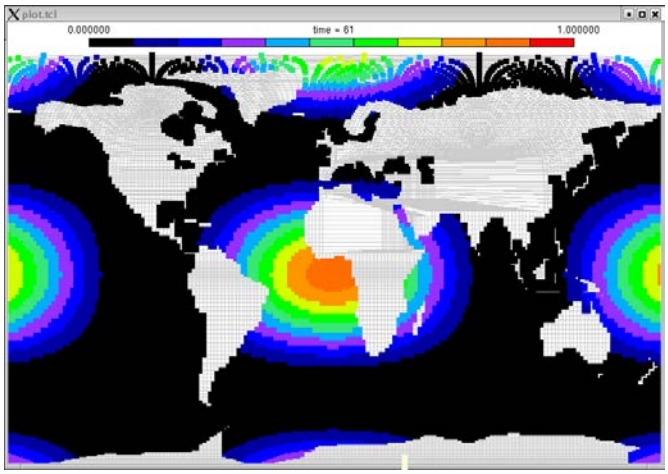
- les différentes méthodes d'interpolation sont également des entrées de la boite prédéfinie. En général on donne ces valeurs « en dur »
- elle fournit le champ interpolé sur la grille cible,
- pour des raisons d'optimisation, la première fois que l'unité tourne, certaines donnés longues à calculer sont stockées dans un fichier de travail dépendant des options données par l'utilisateur, les autres fois que l'unité est appelée, elle se contente de relire les données calculées dans ce fichier.

Interpolez !

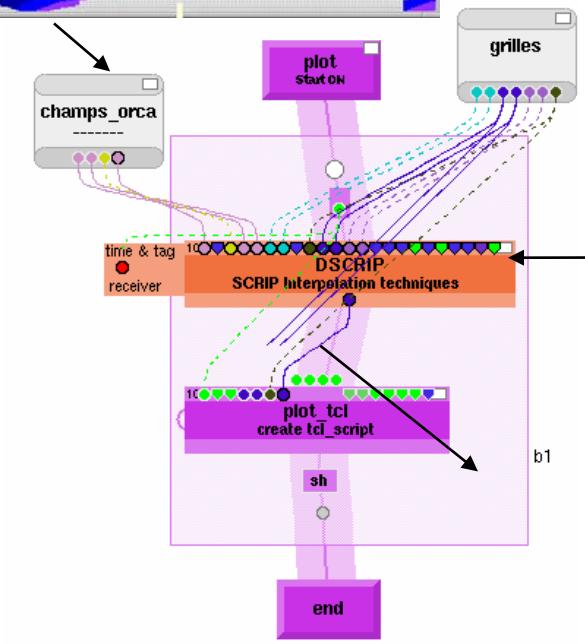
- Avec PrePALM ouvrez le fichier session_13.ppl qui contient une branche appelant l'unité d'interpolation et l'utilitaire de visualisation des champs. Le fichier contenant les champs sur la grille orca est déjà branché dans PrePALM. Remarquez qu'on ne va interpoler que certaines instances temporelles dans ce fichier (boucle do).
- Dans le menu Utilities sélectionnez « Generate id_card of files », choisissez le fichier at42_grids.nc qui contient les grilles sur lesquelles on veut interpoler les champs (en plus des grilles at_42, ce fichier contient les connectivités pour la grille orca).
- La carte d'identité qui vient d'être générée par PrePALM s'appelle at42_grids.id, chargez la, et insérez une instance de ce fichier dans le canevas (click du milieu de la souris dans le canevas), donnez le bon nom de fichier pour le champ filename.
- Créez les communications entre le fichier et les deux unités
- Créez la communication entre le champ de sortie de l'unité d'interpolation et l'unité de visualisation, avant cela remarquez que les deux plots sont jaunes et qu'il faut donc définir l'espace du champ de sortie. Pour cela cliquez sur le plot de sortie de l'unité d'interpolation puis double cliquez sur l'objet sélectionné dans la fenêtre des catégories. Un menu vous invitant à modifier l'espace vous est proposé, choisissez l'espace at42_lat_spc.at42_grids :
-



- Faites tourner l'application, pour vous aider, vous trouverez une copie du canevas sur la page suivante.

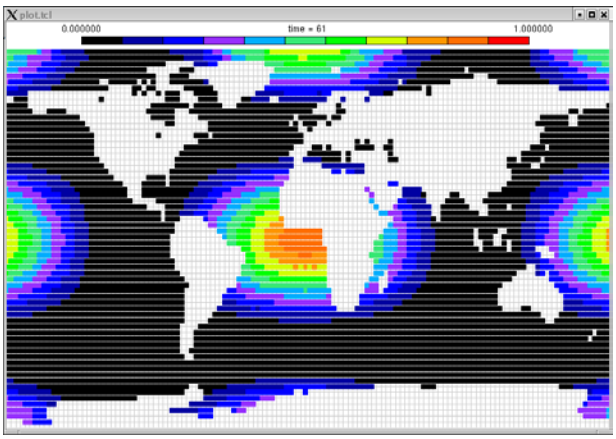


Champs calculés sur la grille Océan (orca 2)



Interpolation spatiale OASIS

Champs interpolés sur la grille ARPEGE AT42



13.1 Rappel des points de cette session

Le principal intérêt de cette session a été de vous apprendre comment utiliser des fichiers NetCDF pour relier des PALM_Get/Put de vos unités à des le lectures/écritures sur disque. La description des fichiers NetCDF se fait par le mécanisme des cartes d'identité de PALM.
Ce mécanisme, mis en œuvre pour des fichiers NetCDF, peut facilement être étendu à d'autres format de fichier auto descriptifs comme HDF5, GRIB, etc. Les utilisateurs intéressés par ces autres formats de fichiers sont fortement encouragés à contribuer à PALM !
Dans cet exemple vous avez également appris à utiliser une boite d'algèbre prédéfinie pour réaliser une interpolation spatiale de champs géophysiques.

14 Session 14 : Utiliser un minimiseur

Dans les boites d'algèbre prédéfinies, PALM propose des minimiseurs qu'il est utile de savoir utiliser pour certaines applications. Certains minimiseurs sont codés en « reverse communication », ils sont plus adaptés à PALM car ils ne monopolisent pas un processus de l'application.

Comme exemple, nous allons minimiser une fonction coût de la forme : $\boldsymbol{J(x) = x^T B^{-1} x}$. Le gradient de cette fonction est $\boldsymbol{\text{grad } J(x) = 2*B^{-1} x}$. Nous utiliserons le minimiseur CGPLUS de type gradient conjugué.

Nous n'aurons besoin que de trois unités utilisateur, la première nommée init donnera une première valeur de la fonction (un vecteur avec tous ses éléments égaux à 1). La seconde unité (compute) prendra un vecteur en entrée et le multipliera par une matrice diagonale $\mathbf{B}^{-1}$ (les éléments de la matrice déjà inversé $\mathbf{B}^{-1}$ sont 1,2,3,…,ip_vectsize) et retournera le résultat $\mathbf{B}^{-1}\mathbf{x}$. La troisième unité (result) ne fera qu'afficher le résultat.

Le gradient $\mathbf{2*B^{-1}x}$ peut être facilement calculé juste en multipliant $\mathbf{B}^{-1}\mathbf{x}$ par 2 avec la boîte d'algèbre DSCAL. L'expression $\mathbf{x}^T\mathbf{B}^{-1}\mathbf{x}$ se calcule avec le produit scalaire de la boite DDOT. Dans notre exemple, la solution analytique est $\mathbf{x = 0}$ à comparer avec le résultat donné par le minimiseur.

Pour simplifier les communications tous les PALM_Put/Get sont faits sans time et sans tag (PL_NO_TIME, PL_NO_TAG).

Le minimiseur CGPLUS est un processus itératif qui fonctionne comme ceci : à partir d'une première valeur de la fonction (f) et de son gradient (G), il calcule un nouveau point où la valeur de la fonction et son gradient doivent être calculés pour faire une nouvelle itération. À chaque itération, le minimiseur sort un drapeau pour dire si le processus de minimisation est terminé ou s'il faut continuer le processus. Le critère de convergence et le nombre maximum de cycle sont des entrées du minimiseur.

L'algorithme sera donc bâti avec une boucle autour du minimiseur (do while) qui ne s'arrêtera que lorsque la convergence sera terminée ou que le nombre d'itérations maximum sera dépassé. L'unité qui donne la première valeur de la fonction sera lancée en dehors de la boucle.

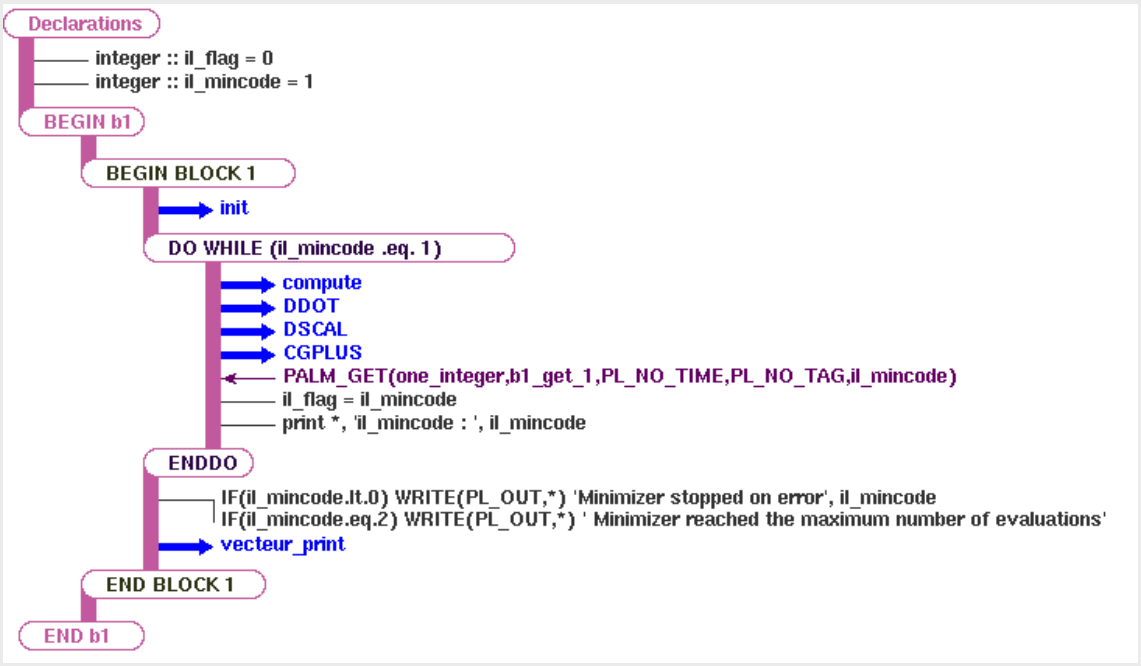
Minimisez !

- Ouvrez PrePALM et définissez ces constantes :

ip_vectsize	10
ip_iter	3*ip_vectsize/2
ip_eval	4*ip_vectsize

- Chargez les trois unités utilisateurs : **init**, **compute**, **vecteur_print**
- et les trois boites d'algèbre **DDOT**, **DSCAL**, **CGPLUS**

- Définissez l'algorithme suivant :



- Créez les communications :
  - De la première valeur du vecteur (first_guess de init) vers compute (vector), DDOT (X) et cgplus (x)
  - De compute (result) vers DDOT (Y) et DSCAL (X)
  - Le résultat de DDOT à CGPLUS (f)
  - Le résultat de DSCAL à CGPLUS (g)
  - Le résultat (x) de CGPLUS à compute (vector) et DDOT (X)
  - Le résultat (result) de CGPLUS à vecteur_print. Il faut également donner un espace correct à ces deux objets (espace défini à NULL), pour cela sélectionnez les objets et éditez les en double cliquant dessus dans la fenêtre de gauche de PrePALM :

no	name	space
25	result.CGPLUS	vect_space.compute

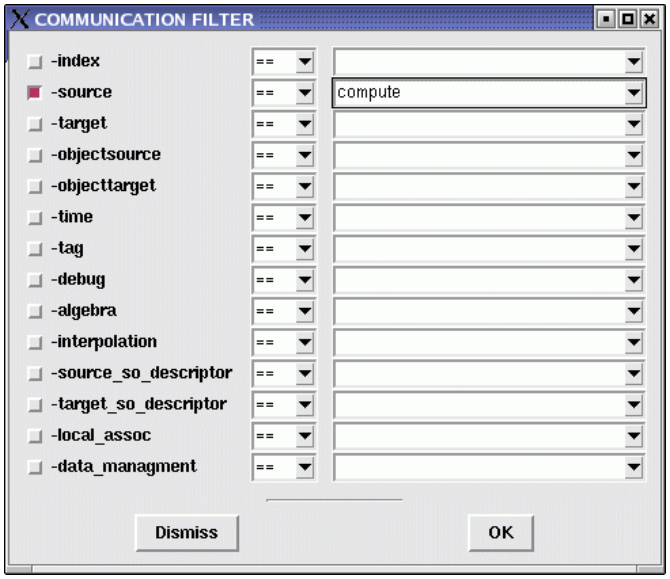
  - Le flag de CGPLUS au PALM_Get de la branche
- Pour les autres plots donnez des valeurs en dur
  - 2.0 pour ALPHA dans DSCAL
  - 1 et 0 pour iprint1 et iprint2 de CGPLUS
  - 1.d-12 pour eps de CGPLUS
  - il_flag pour iflag de CGPLUS
  - 0 pour irest, 2 pour method
  - ip_iter pour nbmaxiter
  - ip_eval pour nbmaxeval
  - .false. pour finish
- Y'a plus qu'à tester…

Profitons du fait que nous avons 11 communications (c'est encore peu, vous verrez qu'à l'usage sur de vraies applications on peut en avoir beaucoup plus), pour montrer quelques facilités de l'interface graphique.
Sélectionnez la catégorie communication de prepalm :

Elargissez la fenêtre des attributs (au détriment du canvas) avec le slicer en bas de PrePALM

Un clic sur le titre d'une des colonnes permet de trier l'ensemble des communications sur un des attributs. Essayez !

Essayez aussi le bouton Filter qui ouvre la boite de dialogue suivante :



Ceci vous permet de sélectionner une partie seulement de la liste sur un ou plusieurs critères, ci-dessus, on aura comme résultat toutes les communications dont l'unité source est compute, cette fonctionnalité, qui ne s'applique pas qu'aux communications, est très utile par exemple lorsque l'on reprend une application développée par une autre personne (ou par vous-même si vous ne l'avez pas ouverte depuis un moment) et comprendre exactement ce qui est fait, examiner certains attributs, etc.
Dans le menu Utilities vous avez aussi des opérations très intéressantes qui facilitent les actions répétitives. Vous pouvez par exemple mettre à TRACK_ON toutes les communications pour avoir plus de renseignement sur ce que fait PALM dans les fichiers de log. En général les problèmes que l'on rencontre avec PALM viennent de communications mal décrites dans PrePALM (champ time tag, …) ou mal codées dans les unités elle mêmes, utilisation d'un espace ou d'un nom d'objet différent de celui déclaré dans la carte d'identité, description d'un time ON, mais envoi avec PL_NO_TIME, etc. Vous pouvez enfin jouer sur le look graphique du canevas dans le menu settings => canevas settings : taille des plots, largeur des branches …

14.1 Rappel des points de cette session

Dans cette session vous avez appris à mettre en œuvre un minimiseur en "reverse communication". Le minimiseur est la base de beaucoup de processus d'optimisation, cette session vous sera particulièrement utile pour implémenter une chaîne d'assimilation de données ou contruire un algorithme d'optimisation autour d'un code de calcul.
Enfin vous avez appris quelques aspects pratiques de PrePALM pour mieux vous retrouver dans des applications complexes contenant de nombreuses unités et communications.

15 Mode MPI-1 de PALM

15.1 Généralités

PALM a été développé en se basant sur le standard MPI-2. En plus des fonctionnalités du mode MPI-1, il exploite deux fonctionnalités du mode MPI-2 :

- le lancement dynamique de processus (fonction MPI_Comm_spawn et fonctions associées),
- les mécanismes client/serveur (fonction MPI_Comm_connect et fonctions associées).

Ceci impose d'avoir une implémentation MPI-2, sinon complète, du moins qui accepte ces fonctionnalités pour le déploiement des applications PALM sur les calculateurs. La plupart des distributions domaine public telles que LAM/MPI, Open-MPI (ne pas confondre avec Open-MP ; cf. §4.5) ou MPICH2 proposent ces fonctionnalités et le bon fonctionnement de PALM a pu être validé avec ces distributions. Cependant, certaines distributions déployées par les constructeurs, en général optimisées sur leurs calculateurs, n'acceptent pas les fonctionnalités MPI-2. On peut par exemple citer le supercalculateur IBM BLUE-GENE L sur lequel il est même impossible d'installer une autre version MPI que celle fournie par IBM.

Pour remédier à ces problèmes de déploiement sur ce genre de machine, une version « dégradée » du coupleur PALM a été développée : c'est ce que l'on appelle le mode MPI-1 de PALM. Le choix de cette version se fait à l'installation de PALM en activant une clé au système de configuration automatique (configure --help pour plus de détails lors de l'installation de PALM) et dans l'interface graphique PrePALM pour les applications (cf. § 15.3).

Le principe de cette version repose sur le lancement de tous les programmes dès le début de l'application en exploitant le mode MPMD de MPI-1. Ce mode n'est pas standard au niveau de MPI-1, mais il est présent quasiment partout sur les différentes implémentations de MPI-1. Dans ce mode MPI-1 « étendu » plusieurs exécutables différents peuvent être lancés en parallèle en se partageant le même communicateur MPI : le communicateur MPI_COMM_WORLD. On est donc dans une configuration MPMD bien que MPI-1.

15.2 Restrictions au niveau du coupleur PALM

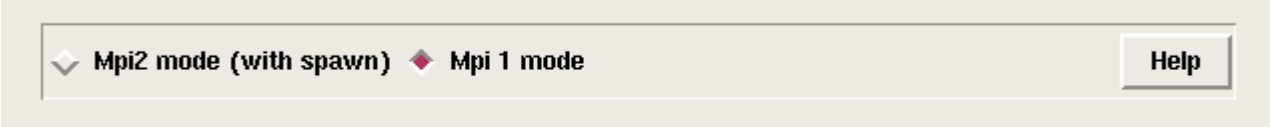
Contrairement au mode MPI-2, les exécutables PALM (unités et blocs), ne peuvent pas être relancés plusieurs fois au cours de la simulation, les boucles et autres structures de contrôles doivent donc être systématiquement encapsulées dans des blocs. On a vu dans la session 3 que cette contrainte d'encapsuler les boucles autour des exécutables n'était pas sans conséquence sur les applications. Si par exemple les programmes ne libèrent pas leur mémoire lors de deux exécutions consécutives on peut être amené à dépasser les capacités de la machine. Les codes de calcul sont rarement prévus pour fonctionner « en boucle » car cela implique un effort de programmation supplémentaire qui n'est pas forcément d'actualité dans les contraintes de développement.

Avant d'opter pour le mode de fonctionnement MPI-1 « étendu » de PALM, il faut donc analyser l'application envisagée pour voir si le couplage est réalisable. Pour des couplages du genre fluide / structure où les codes de calcul n'ont besoin d'être lancé qu'une seule fois en début de simulation, ce mode de fonctionnement est tout à fait envisageable et ce sans dégradation de l'application couplée.

Notons que pour les codes parallèles l'utilisation du communicateur PL_COMM_EXEC à la place de MPI_COMM_WORLD est nécessaire en mode MPI-1 car le communicateur MPI_COMM_WORLD concerne tous les exécutables de l'application y compris le driver de PALM. Ce n'est pas le cas en mode MPI-2 pour un programme « spawné » par le driver qui conserve un MPI_COMM_WORD qui ne concerne que lui, dans ce dernier cas le communicateur PL_COMM_EXEC est une copie du communicateur MPI_COMM_WORLD.

15.3 Lancement d'une application en mode MPI1

L'installation de PALM en mode MPI-1 se fait donc en utilisant l'option --with-mpi1mode lors du configure. Ce mode est activé dans l'interface graphique PrePALM au niveau du menu permettant de générer les fonctions de service (Make PALM files). La case à cocher du mode de fonctionnement désiré doit être activée. Un bouton d'aide permet de vous rappeler les caractéristiques de ce mode.



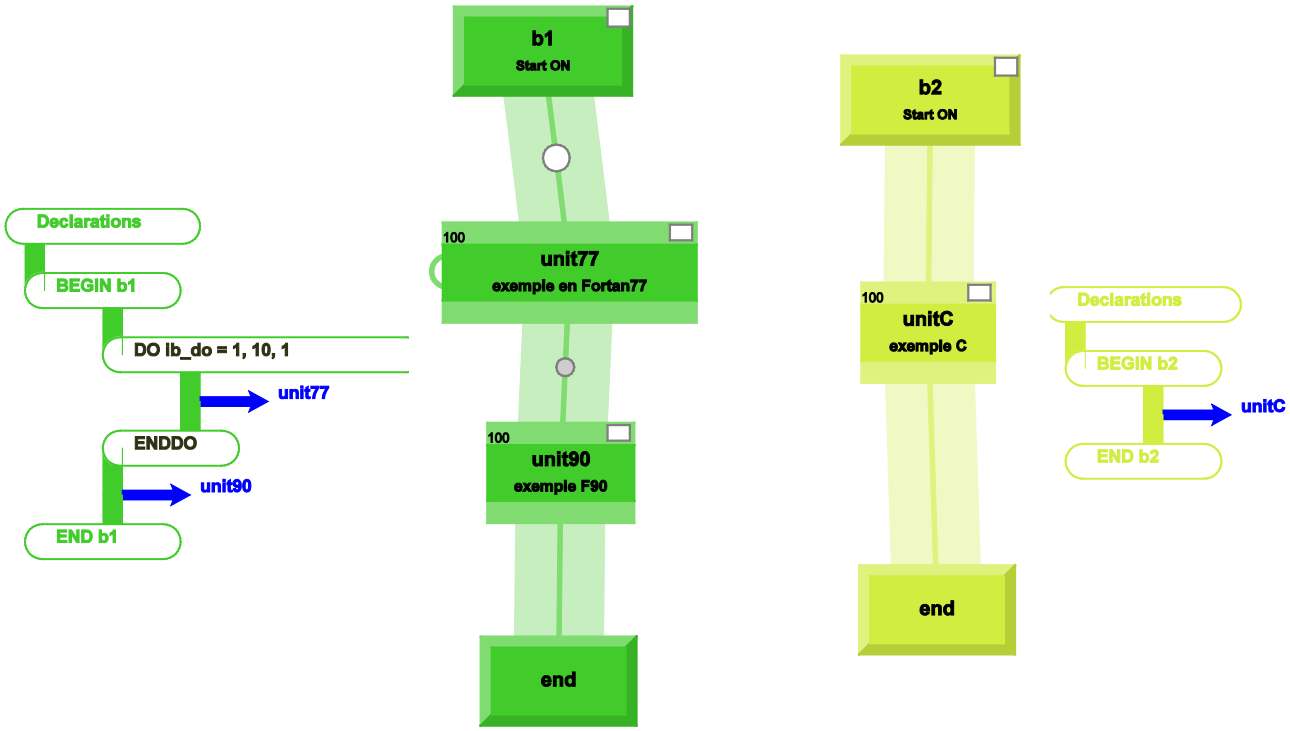
Attention ce choix doit être en totale cohérence avec la bibliothèque PALM que vous utilisez. Si vous préparez les fichiers PrePALM en mode MPI-1 il faut absolument que la bibliothèque PALM du champ PALMLIB du fichier Make.include se termine par l'extension _mpi1mode. Pour plus d'information sur la compilation de PALM reportez vous à la section correspondante (cf. chapitre 19).

Lors de la génération des fichiers de service, un nouveau fichier (run_mpi1.sh) a été créé par PrePALM. Ce fichier prend la syntaxe du mode d'exécution d'un programme compilé avec LAM/MPI, MPICH2 ou Open-MPI avec la commande mpiexec. Il est possible qu'elle doive être adaptée certains calculateurs. Ainsi, alors qu'en mode MPI-2, pour lancer une simulation, il fallait exécuter ./palm_main, en mode MPI-1, il faut exécuter ./run_mpi1.sh.

15.4 Exemple d'application en mode MPI-1

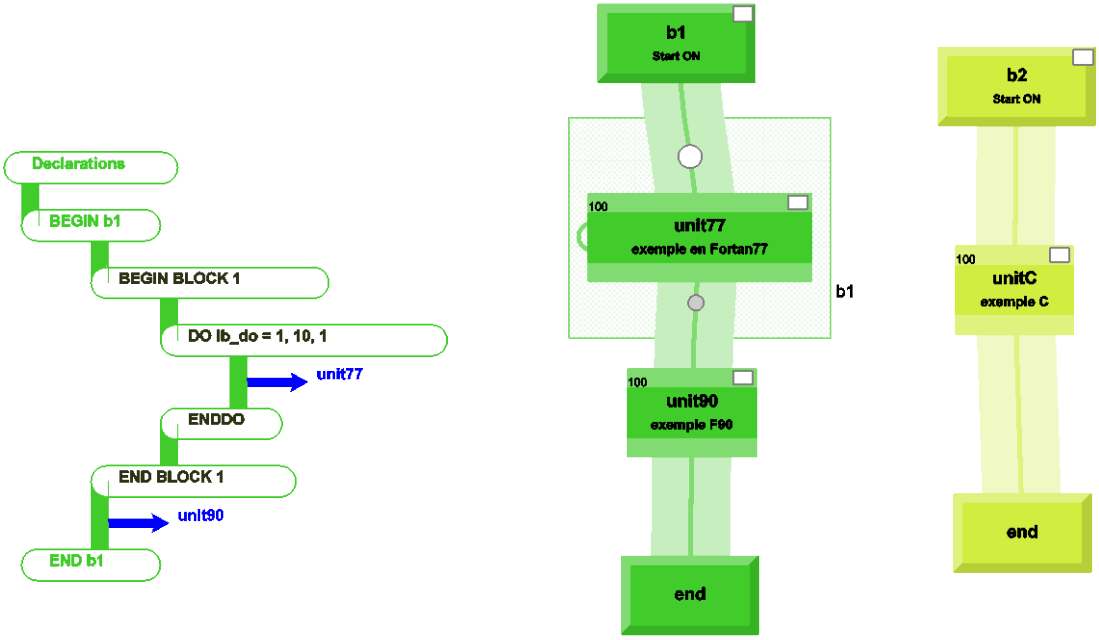
Reprenons les unités de la session 2 de ce manuel (unit77, unit90 et unitC) et examinons les différents cas de figures en mode d'exécution MPI-2 et MPI-1 qui peuvent se présenter pour illustrer les différences.

Soit l'application suivante en langage PrePALM :



En mode MPI-2, cette application comporte 3 exécutables (main_unit77, main_unit90 et main_unitC) plus le driver de PALM. Comme unit90 est sur la même branche que unit77, elle tournera à la suite de cette dernière en récupérant les ressources mémoire et processeur de unit77. Unit77 est lancée 10 fois dans une boucle, à chaque fois le programme est chargé et exécuté, il utilise les mêmes ressources. Cette application PALM a besoin de 3 processus pour tourner en mode MPI-2, un pour le driver et un par branche. Elle peut même tourner sur deux processus sans problème car il n'y a pas de communication entre les deux branches qui conduirait à un blocage.

En mode MPI-1, telle quelle, cette application ne pourrait pas fonctionner car l'unité unit77 est dans une boucle. La seule façon de la faire tourner consiste à mettre un block autour de la boucle. Comme ci-dessous :



Dans ce cas nous avons toujours trois exécutables : main_block_1 (qui a remplacé main_unit77), main_unit90 et main_unitC. En mode MPI-1 ces trois exécutables sont lancés dès le début de l'application par le script run_mpi1.sh qui contient :

```
mpiexec -np 1 ./palm_main : -np 1 ./main_block_1 : -np 1 ./main_unit90 : -np 1 ./main_unitC
```

On a donc besoin de 4 processus pour lancer cette application en mode MPI-1. Notons que l'unité unit90 même si elle est chargée en mémoire dès le début de l'application, n'est pas réellement exécutée, en fait elle attend l'ordre de démarrage du driver de PALM qui la débloquera lorsqu'elle interviendra dans l'algorithme définit par l'utilisateur donc après la boucle sur l'unité unit90. Le comportement de l'application en mode MPI1-1 est donc identique au mode MPI-2, c'est garantit par PALM !

Pour cette application il serait encore plus judicieux de faire rentrer unit90 dans le block pour n'avoir plus que trois exécutables : main_block_1 (qui contient unit77 et unit90) et main_unitC, à ce moment là la commande de lancement devient :

```
mpiexec -np 1 ./palm_main : -np 1 ./main_block_1 : -np 1 ./main_unitC
```

Et le nombre de processus nécessaires pour cette application passe à trois. Contrairement au mode MPI-2, pour gagner encore un processus et passer à 2, il serait nécessaire de faire passer unitC dans la branche b1 à l'intérieur du block.

Attention : Il est important de conserver l'ordre de lancement des exécutables donné dans le script run_mpi1.sh, PALM se base dessus pour créer les communicateurs MPI des différents exécutables, la moindre inversion aurait des effets catastrophiques.

Beaucoup d'applications PALM peuvent tourner en mode MPI-1 « étendu » moyennant des modifications mineures de l'algorithme défini dans PrePALM, la conséquence est hélas parfois un « gaspillage » de ressources. Certaines applications où les codes de calcul tournent en boucle sans possibilité de le mettre dans des blocs ne sont pas possibles en mode MPI-1. Le mode de fonctionnement MPI-2 reste donc le cas le plus général du coupleur dynamique PALM car il apporte beaucoup plus de souplesse.

15.5 Rappel des points de ce chapitre

Dans ce chapitre vous avez appris comment compiler et utiliser une version, certes "dégradée" de PALM, mais qui ne demande pas une implémentation MPI-2. Cette version est nécessaire sur certaines machines particulières pour des raisons de non implémentation du standard MPI-2 ou pour des problèmes de performances.

16 Interpolation de maillages avec la bibliothèque CWIPI

16.1 Généralités

La bibliothèque CWIPI développée à l'ONERA sous licence LGPL (sites.onera.fr/cwipi/) permet d'échanger de l'information entre des codes parallèles basés sur des maillages non structurés. CWIPI, qui fait partie de la distribution OpenPALM, peut être utilisé seul comme coupleur ou via le coupleur OpenPALM pour bénéficier à la fois des fonctionnalités des deux librairies PALM et CWIPI ainsi que de l'interface graphique PrePALM.

Le gros intérêt de CWIPI est qu'il permet d'échanger des champs de couplage sur des maillages différents coté source et coté cible en interpolant à la volée les grandeurs. Les fonctionalités de CWIPI sont basées sur une spatialisation 3D des données ; pour CWIPI un champ de couplage est associé à un maillage décrit sous forme d'éléments non-structurés. Notons qu'un maillage structuré peut parfaitement être décrit comme un maillage non-structuré moyenant le codage d'une interface. CWIPI présente également le gros avantage d'une gestion quasi transparente des codes parallèles s'appuyant sur une décomposition de domaine ; son schéma de communication est bien adapté pour coupler des codes massivement parallèles.

16.2 Bases des maillages non structurés dans CWIPI

Dans CWIPI, les maillages s'appuient sur la définition d'éléments de base tels que :

- des segments pour les éléments 1D,
- des triangles, des quadrangles ou des polygones pour les éléments 2D,
- des tétraèdres, des pyramides, des prismes, de hexaèdres ou des polyèdres pour les éléments 3D.

Tous ces éléments sont décrits dans un repère cartésien 3D commun aux différents codes. CWIPI a besoin d'informations géométriques qui doivent être décrites en mémoire via l'appel à des primitives spécifiques, il impose certaines contraintes sur la forme et le type des tableaux à lui communiquer. Il est rare de fournir directement à CWIPI les tableaux des structures de données internes du code décrivant le maillage car :

- dans le code ces structures de données ne sont pas forcément décrites comme CWIPI les attend,
- les zones de couplages sont généralement locales, par exemple un échange peut se faire juste sur une surface d'un maillage volumique.

16.3 Premiers pas avec CWIPI sous OpenPALM

Dans cet exemple nous allons réaliser un couplage en interpolant un champ 2D entre deux maillages surfaciques. Positionnez-vous dans le répertoire session_PCW/base, Voici (dans le fichier polyg.f90) le premier code que nous allons coupler, en commençant par sa carte d'identité :

```
 1 !PALM_UNIT -name polyg\
 2 !          -functions {f90 polyg}\
 3 !          -object_files {polyg.o}\
 4 !          -comment {CWIPI test fortran}
 5 !
 6 !PALM_OBJECT -name coord_id -space one_integer -intent IN\
 7 !            -closedlist {{1 : X coord} {2 : Y coord}}\
 8 !            -default 1\
 9 !            -comment {field to send}
10 !
11 !PALM_CWIPI_COUPLING -name cpl1
12 !
13 !PALM_CWIPI_OBJECT -name exch1\
14 !                  -coupling cpl1\
15 !                  -intent INOUT
```

11 : Un nouveau mot clé permet de décrire un couplage CWIPI, l'utilisateur peut décrire plusieurs couplages CWIPI dans le même code. Associé à ce couplage, on trouve des objets CWIPI (13) qui identifient les champs à envoyer ou à recevoir. Comme pour les espaces et les objets PALM, les noms donnés aux couplages et objets CWIPI sont internes à chaque code. Dans notre cas, l'objet exch1 possède l'attribut INOUT ce qui veut dire qu'associé à cet objet nous ferons un envoi et une réception dans le code. Comme pour PALM l'instrumention de chaque code est indépendante des autres codes à coupler ; c'est en décrivant l'application dans l'interface graphique PrePALM qu'on va relier un « couplage » d'un code à celui d'un autre code. Examinons maintenant le code FORTRAN :

```
17 subroutine polyg()
18
19   use cwipi
20   use palmlib
21   implicit none
22   include 'mpif.h'
23
```

19 : Un module cwipi est mis à disposition de l'utilisateur, ce module contient les constantes génériques pour l'appel aux primitives CWIPI. Viennent ensuite la déclaration des variables du code :

```
24   integer, parameter :: nvertex = 11, nelts = 5
25
26   double precision :: coords(nvertex*3)
27   integer :: connecindex(nelts+1)
28   integer :: connec(21)
29
30   double precision :: values(nvertex), localvalues(nvertex)
31
32   integer :: stride = 1
33   integer :: il_err, i, coord_id, nNotLocatedPoints
34
35   character(len=PL_LNAME) :: cl_space, cl_name
36   character(len=PL_LNAME) :: cl_coupling_name, cl_exchange_name
37   character(len=PL_LNAME) :: output_format, output_format_option
```

```
38   character(len=PL_LNAME) :: cl_sending_field_name, cl_receiving_field_name
39
```

Nous allons définir un maillage basé sur une collection de 11 points (nvertex) d'un repère cartésien 3D, ces points permettront de définir 5 éléments (nelts). Nous reviendrons plus loin sur les tableaux coords, connecindex, et connec qui serviront à définir le maillage.

```
40   ! coordinate to send
41   cl_space = 'one_integer'
42   cl_name = 'coord_id'
43   CALL PALM_get(cl_space, cl_name, PL_NO_TIME, PL_NO_TAG, coord_id, il_err)
44
```

En préambule aux échanges CWIPI, via un PALM_Get, le code demande comment initialiser le tableau de champs qu'il va envoyer, avec un choix possible entre envoyer un tableau contenant la coordonnée X ou la coordonnée Y (comme décrit dans la carte d'identité de l'unité (7)), l'envoi d'un champ dans lequel on a mis les valeurs d'une coordonnée du maillage permet de réaliser des tests simples avec un contrôle visuel rapide des champs échangés.

```
45   ! coupling initialization
46   CALL PCW_Init(il_err)
47   cl_coupling_name = "cpl1"
48   output_format = 'Ensight Gold'
49   output_format_option = 'text'
50   call PCW_Create_coupling(cl_coupling_name,  &
51                        cwipi_cpl_parallel_with_part, &
52                        2,                            &   ! Geom. ent. Dim.
53                        0.1d0,                        &   ! Geom.tolerance
54                        cwipi_static_mesh,            &   ! Mesh type
55                        cwipi_solver_cell_vertex,     &   ! Solver type
56                        1,                            &   ! Output frequency
57                        output_format,                &   ! Postpro. format
58                        output_format_option,         &
59                        il_err)
```

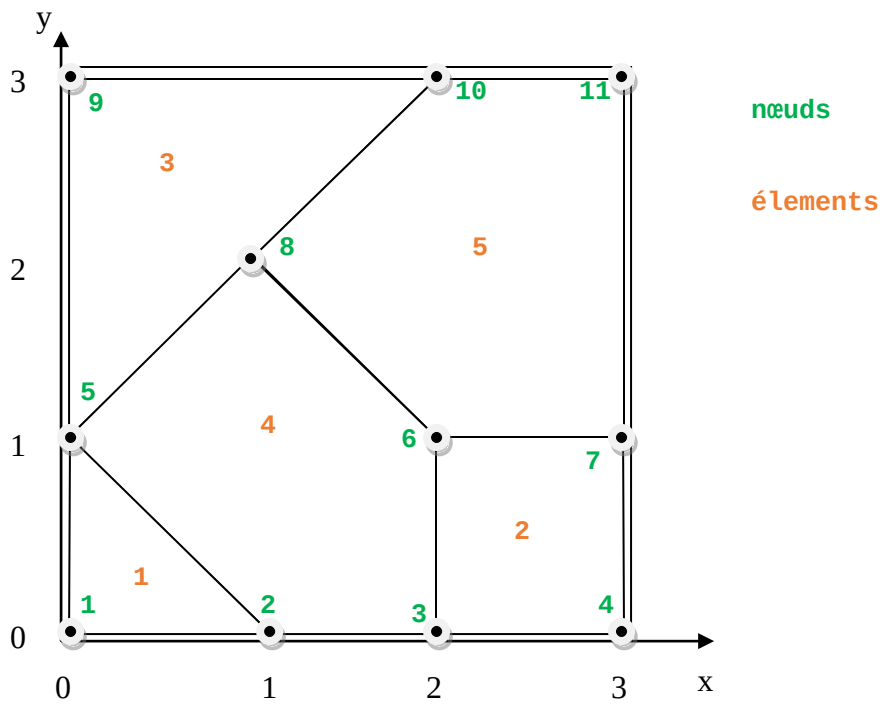
Toutes les primitives CWIPI interfacées pour OpenPALM commencent par PCW_. Le code doit appeler la primitive PCW_Init (46), puis il définit certaines caractéristiques du couplage « cpl1 » (définit dans la carte d'identité) via la primitive PCW_Create_coupling (50). Dans notre exemple le code informe cwipi :

- qu'il est parallèle avec une décomposition de domaine (51),
- qu'on va manipuler des éléments 2D, il s'agira donc d'un couplage surfacique, même si pour CWIPI les surfaces sont définies dans un repère 3D (52),
- qu'on admet une tolérance géométrique locale correspondant à 10% de la taille d'une maille pour la recherche de points à localiser dans ce maillage (53), on verra plus loin l'intérêt d'agir sur cette tolérance,
- que le maillage ne va pas bouger au cours de la simulation (54),
- que les champs à échanger seront définis sur les sommets des éléments (55),
- que nous voulons des sorties de contrôle des champs à chaque itération avec une définition des formats de sortie au format « Ensight » en ASCII, lisible par le logiciel de visualisation paraview.

On construit ensuite les trois tableaux nécessaires à CWIPI pour l'appel de PCW_Define_Mesh :

```
60
61    ! Mesh definition
62    coords = (/0,0,0,  1,0,0,  2,0,0,  3,0,0,  0,1,0,  2,1,0,&
63               3,1,0,  1,2,0,  0,3,0,  2,3,0,  3,3,0/)
64    connecindex = (/0,3,7,11,16,21/)
65    connec = (/1,2,5,   3,4,7,6,   5,8,10,9   ,5,2,3,6,8,   6,7,11,10,8/)
66
```

Le tableau coords (62) contient les coordonnées des nœuds du maillage (voir figure ci-dessous), c'est un tableau 1D de double précision de taille nvertex*3 ; les coordonnées sont entrelacées (x1, y1, z1, x2, y2, z2, …, xn, yn, zn). Le tableau d'entiers connecindex (64) de taille (nelts +1) contient les informations qui permettent de savoir quel type d'élément est décrit un à un dans le tableau connec, le premier indice de ce tableau est toujours 0, puis on cumule le nombre de sommets de l'élément suivant. Dans notre cas nous avons 1 triangle, puis 2 quadrangles, puis 2 surfaces à 5 sommets. Les éléments doivent toujours être décrits par ordre croissant de sommets. Enfin le tableau connec (65), de taille connecindex(nelts+1) contient la description des éléments un par un, dans cet exemple on a pris soin de séparer les 5 éléments par des espaces pour plus de lisibilité.




```
67    call PCW_Define_mesh(cl_coupling_name, &
68                         nvertex,          &
69                         nelts,            &
70                         coords,           &
71                         connecindex,      &
72                         connec,           &
73                         il_err)
74
```

Toutes les caractéristiques du maillage sont communiquées à CWIPI via la primitive PCW_Define_mesh (67).
Attention : dans CWIPI, les tableaux passés à la primitive PCW_Define_mesh sont « mappés » en mémoire, sans recopie. Dans l'étape PCW_Define_mesh CWIPI ne fait que conserver l'adresse mémoire des tableaux envoyés, il ne traite ces données qu'au moment où il déclenche l'algorithme de localisation qui, s'il n'est pas demandé explicitement, ne se produit que lors du premier échange sur ce couplage. Il ne faut donc surtout pas détruire ni modifier à tort les tableaux coords connec et connecindex avant la fin du couplage CWIPI.

```
75   ! initialization sending field
76   do i = 1, nvertex
77      values(i) = coords((i-1) * 3 + coord_id)
78   end do
79
80   cl_exchange_name = 'exch1'
81   if (coord_id .eq. 1) then
82      cl_sending_field_name = 'coox'
83   else
84      cl_sending_field_name = 'cooy'
85   end if
86   cl_receiving_field_name = 'recv'
87
```

Le tableau values (77) qui contient le champ à envoyer est initialisé soit à la valeur de la coordonnée X, soit à la valeur de la coordonnée Y (choix de l'utilisateur dans PrePALM). On passe ensuite à la phase d'échange des données (89) :

```
88   ! exchange
89   call PCW_Sendrecv (cl_coupling_name,            &
90                      cl_exchange_name,            &
91                      stride,                      &
92                      1,                           & ! step index
93                      0.1d0,                       & ! physical time
94                      cl_sending_field_name,       & !
95                      values,                      & ! sending field (IN)
96                      cl_receiving_field_name,     & !
97                      localvalues,                 & ! receving field (OUT)
98                      nNotLocatedPoints,           &
99                      il_err)
```

CWIPI peut faire simultanément un envoi et une réception avec un PCW_Sendrecv, l'entier stride (91), ici à 1, sert à envoyer plusieurs champs dans le même tableau. Si stride est supérieur à 1, il faut remplir le tableau en entrelaçant les champs, par exemple pour un stride de 3 avec des champs u, v et w, on aurait (u1, v1, w1, u2, v2, w2, …, un, vn, wn) dans les tableaux values et localcalues. Ce code va être couplé avec un autre qui définira lui aussi un maillage. Si les maillages des deux codes sont coïncidents (qu'ils recouvrent exactement la même surface géométrique, tous les point d'un maillage cible seront localisés dans le maillage source, sinon CWIPI retournera le nombre de points qui n'ont pas été localisés dans la variable nNotLocatedPoints (98). Un traitement particulier des points non localisés peut donc être nécessaire. Contrairement à notre exemple, dans un

couplage entre des vrais codes la phase d'envoi/réception de données est en général appelée plusieurs fois.

```
100
101   call PCW_Delete_coupling(cl_coupling_name,il_err)
102   call PCW_Finalize(il_err)
103
104 end subroutine polyg
```

Les deux dernières primitives CWIPI permettent de détruire le couplage et de terminer la session CWIPI (101 et 102).

Pour tester ce « code », on peut le coupler avec lui-même, pour cela il suffit de lancer deux instances de cette unité PALM dans deux branches différentes.

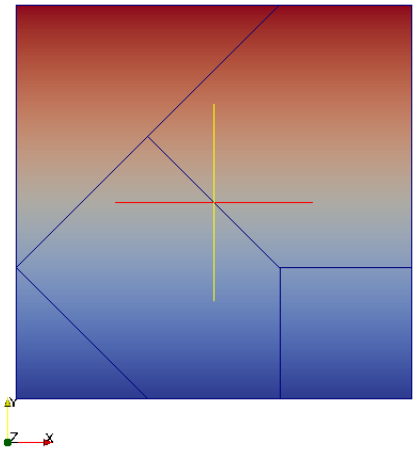
- Positionnez vous dans le répertoire session_PCW/base
- Créez deux branches avec PrePALM.
- Charger la carte d'identité de l'unité polyg.f90.
- Lancer une instance de polyg dans chacune des deux branches.
- Pour l'objet PALM coord_id, choisir (click droit sur le plot) 1 pour un des codes et 2 pour le second, ainsi les champs envoyés par les deux instances du code seront différents (coordonnée x ou y).
- Il vous faut maintenant préciser quel code communique via CWIPI avec quel autre, cliquez sur la zone CWIPI de l'une des unités, PrePALM vous propose d'insérer un couplage, choissisez le seul couplage qu'il vous propose.
- Un trait épais grisé reliant les deux unités apparait, cliquez sur ce trait et insérez la communication proposée.
- Générez les fonctions de service, compilez et faites tourner l'application.

Dans notre cas, nous avons demandé des sorties graphiques (Output frequency fixé à 1 du PCW_Create_coupling), ces sorties sont stockées dans un sous répertoire nommé cwipi contenant lui-même un répertoire par code et par nom d'objet d'échange. Il est conseillé de mettre cette fréquence des sorties à 0 une fois que le couplage est au point pour éviter des accès disques couteux en temps de calcul.

Pour visualiser le résultat du couplage avec le logiciel graphique paraview, positionnez-vous dans un des sous répertoire cwipi et lancez la commande :

```
> paraview --data=CHR.case
```

Une fois appuyé sur le bouton Apply de paraview vous pourrez voir le champ reçu ou le champ envoyé par CWIPI :



Nous allons maintenant coupler le code polyg avec un autre code qui définit un maillage différent.

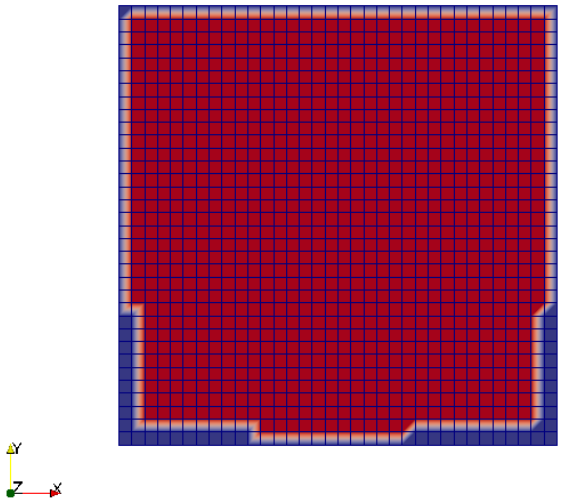
Exercice PCW base 1

Ouvrez le fichier rectangle.f90, répondez aux questions en regardant le code Fortran.
Questions :

- Quel type de maillage CWIPI est définit dans ce code , 1D, 2D ou 3D ?
- Combien de nœuds pour ce maillage ?
- Combien d'élements ?
- Quels types d'éléments ?
- Quelle est la portion de l'espace 3D couverte par ce maillage ?
- Quel est le champ envoyé par rectangle ?
- Que fait-on si des points sont non localisés ?

Remplacez l'une des deux instances de polyg par rectangle et testez.

Rectangle vous avertit qu'il trouve 175 points non localisés. Allez dans le sous répertoire de cwipi/cpl1.cpl1_rectangle_polyg, et lancer paraview. En visualisant le champ « location » vous verrez quels sont les points de rectangle non localisés dans polyg :



Points localisés par CWIPI

Lorsque CWIPI détecte des points non localisés, il crée un champ « location » dans les sorties graphiques, les points localisés ont une valeur de 1 et apparaissent en rouge avec paraview, les points non localisés ont une valeur de 0 et apparaissent en bleu. La présence de points non localisés vient du fait que le maillage définit par l'unité rectangle est plus étendu que celui définit par polyg, avec 2 mailles de rectangle supplémentaires tout autour. On voit sur la sortie graphique l'effet de la tolérance géométrique (fixée à 0.1 dans polyg.f90) ; on remarque cependant qu'avec cette tolérance certains points de rectangle en dehors du maillage de polyg sont tout de même localisés, et que d'autres points restent non localisés. On observe que cette tolérance géométrique est locale, elle est relative à chaque maille. Selon que l'élément de polyg le plus proche est plus ou moins grand certains points sont ou non localisés. Essayez maintenant d'agir sur cette tolérance pour faire en sorte de :

- ne localiser aucun point en dehors du maillage de polyg,
- puis de localiser tous les points.

Le choix que vous adopterez pour traiter ces points non localisés dans vos couplages dépendra certainement des problèmes traités mais une chose est certaine : il faut systématiquement vérifier et traiter les points non localisés quand on réalise un couplage avec CWIPI.

Notons aussi que la valeur de la tolérance géométrique conditionne le temps de calcul de CWIPI pour la recherche des voisins, plus elle est faible, moins CWIPI y passera de temps. Dans notre exemple on pourrait aller jusqu'à une tolérance très faible (0.0001 par exemple), une tolérance nulle conduit à ne rien localiser. Dans notre exemple les deux maillages sont inscrits dans le plan Z=0, si les maillages épousaient une forme 3D comme une sphère, il est probable qu'une tolérance trop faible conduirait à de nombreux points non localisés, le choix de la valeur de la tolérance géométrique est donc affaire d'expérience, en général on commence par une valeur de 0,1 puis on

ajuste en fonction des résultats obtenus, une bonne manière de procéder consiste à lire la tolérance géométrique dans un fichier de données pour éviter la phase de recompilation à chaque test.

Nous allons maintenant passer à une version parallèle du code rectangle. Pour faire simple la décomposition du domaine est réalisée uniquement sur l'axe des X, chaque processus aura le même nombre de mailles définit par défaut ou dans un fichier d'entrée de nom rectangle_par.mesh. Pour examiner les différences entre le code parallèle et le code séquentiel ouvrez les deux fichiers avec l'utilitaire tkdiff :

```
> tkdiff rectangle.f90 rectangle_par.f90
```

Vous pouvez observer:

- l'inclusion de mpif.h pour appeler les primitives MPI_Comm_rank et MPI_Comm_size,
- le re-calcul des coordonnées locales de chaque domaine xmin et xmax,
- qu'aucune des primitives PCW_ n'a été modifié.

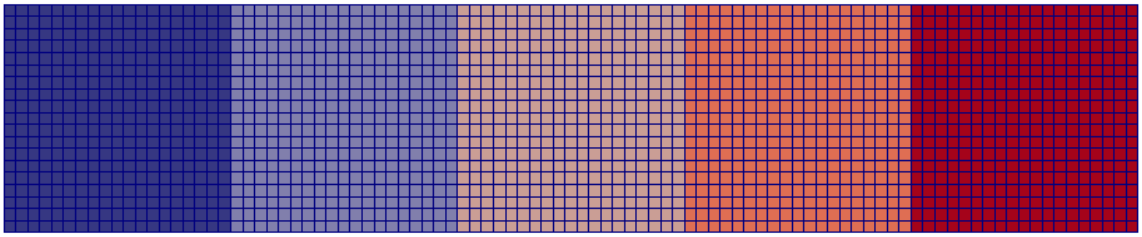
Exercice PCW base 2

- Construisez deux branches et lancez les unités rectangle et rectangle_par.
- Donnez 5 processus à rectangle_par.
- Dans settings -> palm execution settings donnez un nombre suffisant de processus pour l'application.
- Construisez les deux fichiers de définition du maillage par exemple comme ceci :

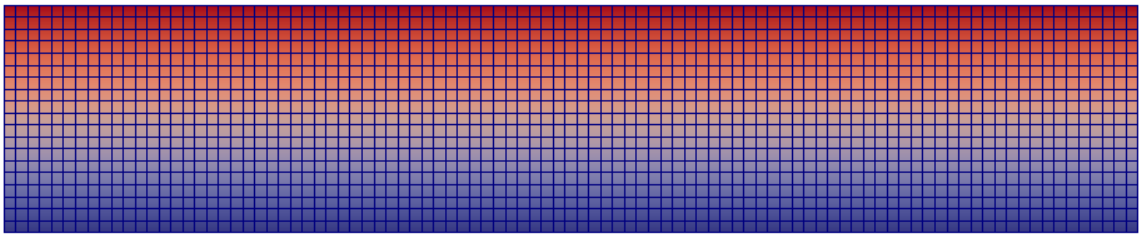
rectangle.mesh :	rectangle_par.mesh:
0. xmin	0. xmin
5. xmax	5. xmax
50 nx	20 nx
0. ymin	0. ymin
1. ymax	1. ymax
10 ny	20 ny

- lancer l'application et affichez les résultats de rectangle_par avec paraview.

Avec ces paramètres pour les maillages vous devriez obtenir ces sorties graphiques dans le répertoire de sortie du code parallèle :



Décomposition du domaine sur les 5 processus (champs partitioning) une couleur est affecté à chaque processus.



Champs recu et interpolé de rectangle (champ R_C0_exch1.exch1_co)

C’est parce que CWIPI travaille avec des maillages, donc dans un système de coordonnées commun à toutes les unités pour tous les processus, que l’instrumentation est quasi transparente entre un code parallèle et un code mono processus. Chaque processus définit uniquement la partie du maillage global qu’il connait. Une phase de localistion des points, réalisée lors du premier échange permet à CWIPI de savoir où trouver les informations pour interpoler les champs (quel processus et quelle maille pour chaque point à localiser). Une fois cette opération réalisée, CWIPI interpole les champs dans la maille en affectant des poids relatifs aux coordonnées baricentriques du point concerné dans la maille localisée. L’opération de localisation et de calcul des coordonnées baricentriques n’est effectuée qu’une seule fois ce qui permet d’aller beaucoup plus vite pour les échanges suivants s’ils se font en boucle, ce qui est généralement le cas pour les couplages multiphysiques. La phase de localisation, plus coûteuse que la phase d’interpolation et d’échange, est néanmoins optimisée dans CWIPI car elle s’appuie sur un algorithme d’octree subdivisant recursivement l’espace 3D de recherche. Si les valeurs des champs sont données sur les nœuds des mailles (CWIPI_SOLVER_CELL_VERTEX) le champ est interpolé, si ces valeurs sont données au centre des cellules (CWIPI_SOLVER_CELL_CENTER) le champ n’est pas interpolé.

La plupart des primitives PCW_ sont des opérations collectives donc synchronisantes entre les processus d’un même code et les autres processus du code avec lequel on échange de l’information. On a donc un fonctionnement assez différent par rapport aux primitives PALM_Get/Put. Avec

CWIPI on ne peut pas par exemple envoyer de l'information entre deux codes qui tourneraient en séquence sur la même branche comme on peut le faire avec des PALM_Get/put.

16.4 Un exercice plus complet

Cet exercice a pour but d'aller un peu plus loin dans la prise en main des primitives d'OpenPALM interfaçant les appels à la bibliothèque CWIPI. On apprendra à gérer les points non localisés ou à traiter des maillages mobiles. L'exercice consiste à échanger des données entre un code fortran (*fortran_surf.f90*) et un code C (*c_surf_coupling.c*). Les deux codes gérent des surfaces tridimensionnelles mobiles en temps caractérisées par (voir Figure 1) :

- $-10 \leq x \leq 10$
- $-10 \leq y \leq 10$
- $z = ampl*\cos(omega*time+phi)*(x^2+y^2)$

La coordonnée z subit donc une oscillation temporelle sinusoïdale d'amplitude *ampl*, de fréquence *freq* (avec omega = 2.pi.freq) et de déphasage *phi*. Ces grandeurs sont propres à chaque code et sont définies dans les fichiers de données *dataC.dat* pour le code C et *dataF.dat* pour le code Fortran.

Ce fichier d'entrée a cette forme :

```
1       ! itération de depart
10      ! itération finale
0.10d0  ! fréquence
0.012d0 ! amplitude
0.1d0   ! phase
0.1d0   ! Tolérance Géométrique
```

Notons que le pas de temps de l'évolution temporelle est de 0.1.

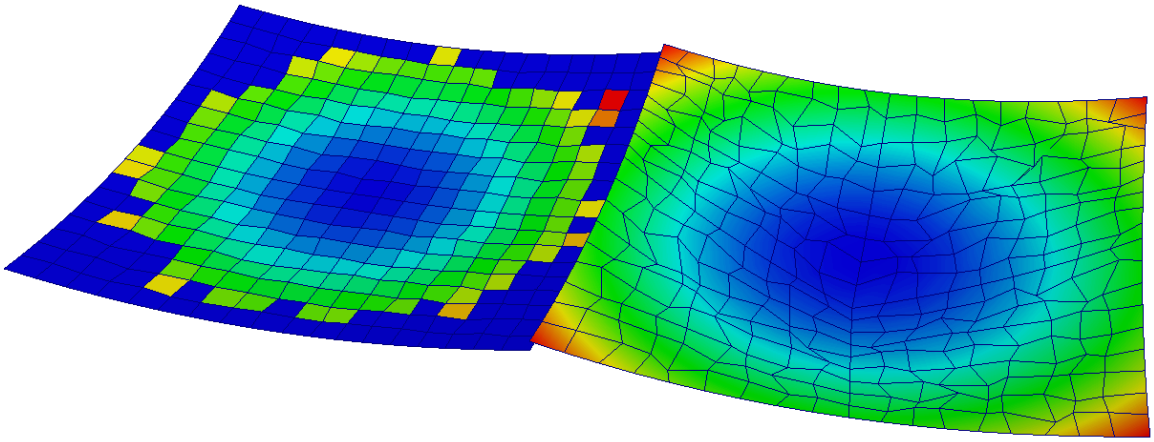


Figure 1. Surfaces d'échanges gérées par les codes: à gauche pour le code C et à droite pour le code Fortran.

Le maillage définit dans le code Fortran est de type *cell centered*, c'est à dire que les grandeurs calculées et envoyées sont stockées au centre des cellules. Le maillage du code C est quant à lui *cell vertex*, c'est à dire que les variables sont localisées aux nœuds du maillage. L'échange se fait au travers d'un *send/receive* croisé, ce qui permet d'avoir une symétrie sur l'instrumentation et l'exécution des codes. Les champs échangés sont de forme analytique, ce qui permet de valider très

facilement les résultats obtenus. Les codes s'échangent un champ scalaire qui a pour valeur leurs coordonnées z aux nœuds ou aux cellules.

Le tutorial va se dérouler en quatre exercices permettant d'appréhender au fur et à mesure les concepts important de l'utilisation des primitives OpenPALM PCW_.
Pour le premier exercice (qui sera le plus long et le plus important), nous allons traiter la mise en relation des codes Fortan et C en ne jouant qu'une seule itération temporelle.
Toujours sur la base d'une seule itération, le second exercice permettra d'appréhender des exemples d'actions à mener en cas de détections de points non localisés par le coupleur.
Le troisième exercice permettra de mettre en œuvre les itérations temporelles du couplage en considérant que les surfaces gérées par les codes sont statiques.
Enfin, le dernier exercice s'attachera à montrer une solution permettant de gérer les surfaces couplées et mobiles au cours du temps.

Les primitives utiles pour mener à bien cet exercice sont les suivantes :
→ Initialisation de l'application couplée
- PCW_Init
- PCW_Finalize
- PCW_Create_coupling
- PCW_Delete_coupling

→ Définition du maillage
- PCW_Define_mesh
- PCW_Update_location

→ Contrôle des communications
- PCW_Sendrecv
- PCW_Get_not_located_points

→ Contrôle de l'application
- PCW_Dump_application_properties
- PCW_Dump_notlocatedpoints
- PCW_Dump_status

Les arguments de ces primitives sont décrits à la fin de ce manuel.
Le dernier paragraphe de ce chapitre présente des primitives qui permettent d'aller plus loin dans l'utilisation de CWIPI.

16.5 Définition du couplage dans PrePALM

Pour ces exercices, l'application sous l'environnement PrePALM est exactement la même pour tous les exercices. Nous allons donc la construire une fois pour toutes.

Pour commencer, examinons les cartes d'identité des unités correspondant aux codes (ci-après). Ces cartes d'identités sont fournies dans les entêtes des codes. Comme pour tout code de calcul porté sur OpenPALM, on voit apparaître l'utilisation du mot clé *PALM_UNIT* permettant de renseigner le nom de l'unité ainsi que quelques paramètres que nous connaissons bien désormais. On voit ensuite apparaître 6 objets (mot clé *PALM_OBJECT*) reçus par les unités. Ces objets correspondent aux grandeurs à lire dans les fichiers de données *dataC.dat* et *dataF.dat*. Notons dès

à présent que la variable *geom_tol* correspond à une tolérance géométrique utilisée par CWIPI pour l'étape de localisation entre les maillages.
Totalement nouveau par rapport aux autres mots clés, le mot clé *PALM_CWIPI_COUPLING* permet de définir des environnements de couplage CWIPI pour les unités. A ces environnements on associe ensuite des objets d'échanges (mot clé *PALM_CWIPI_OBJECT*). Ces objets échangés peuvent être soit uniquement reçus (*intent IN*), soit uniquement produits (*intent OUT*) soit les deux (*intent INOUT*). Notons qu'il est possible d'associer plusieurs couplages à une unité. Par exemple un code fluide peut être pourvu de primitives pour échanger des données simultanément avec un code de conduction thermique, un de rayonnement etc …

Carte d'identité du code C :

```
/*PALM_UNIT -name c_surf_coupling\
            -functions {C c_surf_coupling}\
            -parallel mpi\
            -minproc 1\
            -maxproc 100000\
            -object_files {c_surf_coupling.o grid_mesh.o}\
            -comment {CWIPI test c_surf_coupling}
*/
/*PALM_OBJECT -name itdeb\
              -space one_integer\
              -intent IN\
              -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_OBJECT -name itend\
              -space one_integer\
              -intent IN\
              -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_OBJECT -name freq\
            -space one_double\
            -intent IN\
            -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_OBJECT -name ampl\
            -space one_double\
            -intent IN\
            -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_OBJECT -name phi\
            -space one_double\
            -intent IN\
            -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_OBJECT -name geom_tol\
            -space one_double\
            -intent IN\
            -localisation REPLICATED_ON_ALL_PROCS
*/
/*PALM_CWIPI_COUPLING -name c_surf_cpl
*/
/*PALM_CWIPI_OBJECT -name echange1\
                  -coupling c_surf_cpl\
                  -intent INOUT
```

```
*/
```

Carte d'identité du code Fortran :

```
!PALM_UNIT -name fortran_surf\
!          -functions {f90 fortran_surf}\
!          -parallel mpi\
!          -minproc 1\
!          -maxproc 100000\
!          -object_files {fortran_surf.o grid_mesh.o}\
!          -comment {CWIPI test fortran_surf}
!
!PALM_OBJECT -name itdeb\
!            -space one_integer\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!
!PALM_OBJECT -name itend\
!            -space one_integer\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!
!PALM_OBJECT -name freq\
!            -space one_double\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!
!PALM_OBJECT -name ampl\
!            -space one_double\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!
!PALM_OBJECT -name phi\
!            -space one_double\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!PALM_OBJECT -name geom_tol\
!            -space one_double\
!            -intent IN\
!            -localisation REPLICATED_ON_ALL_PROCS
!
!PALM_CWIPI_COUPLING -name test2D_3
!
!PALM_CWIPI_OBJECT -name echange1\
!                  -coupling test2D_3\
!                  -intent INOUT
```

Pour construire l'application commençons par charger les cartes d'identités qui sont localisées en entête des sources des deux codes (Figure 2), les informations trouvées concernant CWIPI sont décrites à la fin du compte rendu de lecture donné par PrePALM.

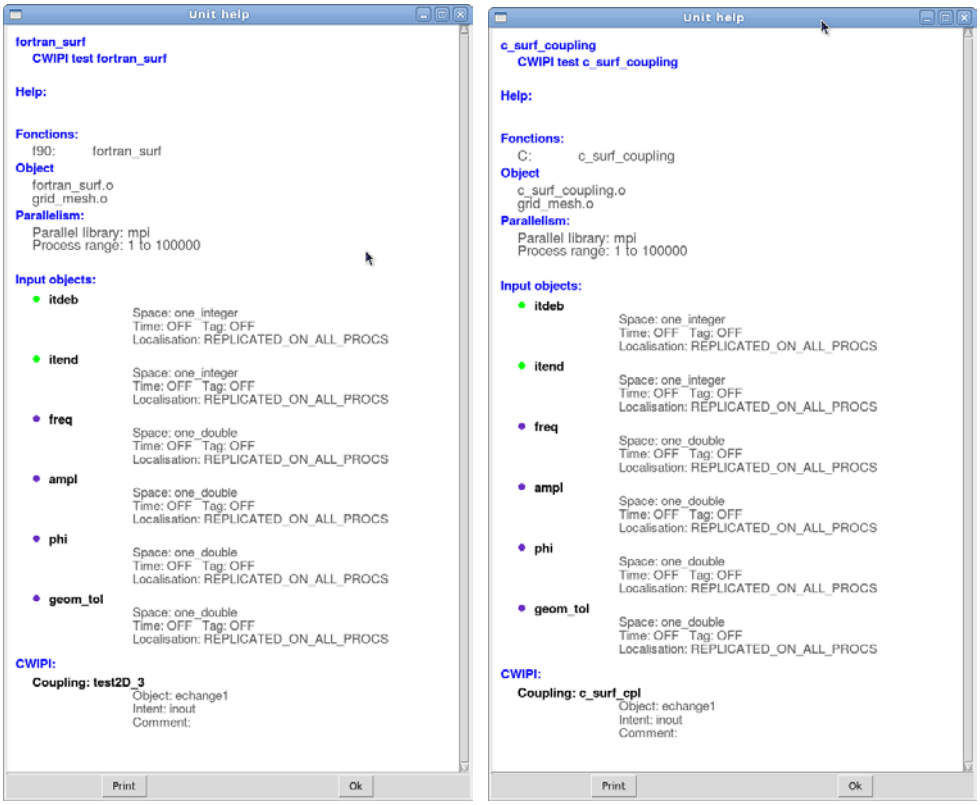


Figure 2. Cartes d'identités des unités chargées sous PrePALM.

Nous allons ensuite créer deux branches *b1* et *b2* qui lanceront respectivement les unités fortran_surf et c_surf_coupling. On constate sur la Figure 4 qu'un encadré CWIPI apparaît en bas à gauche des unités rappelant ainsi que ces unités peuvent communiquer via la bibliothèque CWIPI. Ces unités sont parallèles. Affecter 4 processus à une unité et un seul à l'autre (attention, même si ces unités sont parallèles, elle ne peuvent pas tourner sur un nombre quelconque de processeurs, un contrôle est fait à l'exécution dans le code). Enfin, dans l'onglet *Execution working directory*, spécifier le répertoire de travail *./CODEF* pour le code Fortran et *./CODEC* pour le code C (Figure 3). Assurez-vous que ces répertoires existent bien à l'emplacement où l'application sera exécutée. Cette opération permettra de faire en sorte que les codes lisent et génèrent des données dans des répertoires biens distincts ce qui est très utile lors de l'utilisation de codes complexes avec des jeux de données conséquents.

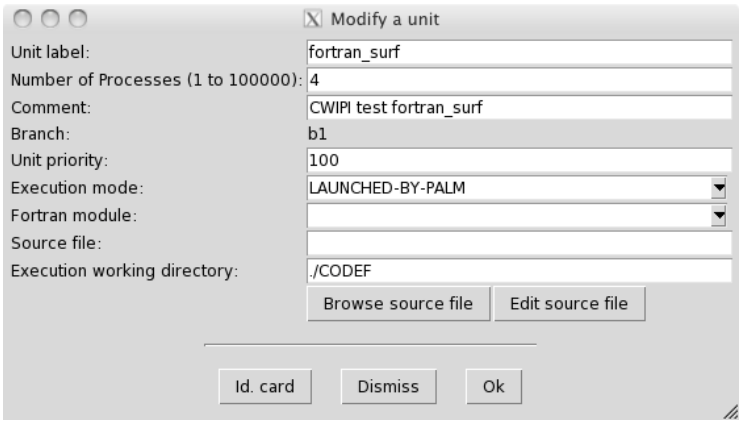


Figure 3. Edition des caractéristiques de l'unité Fortran.

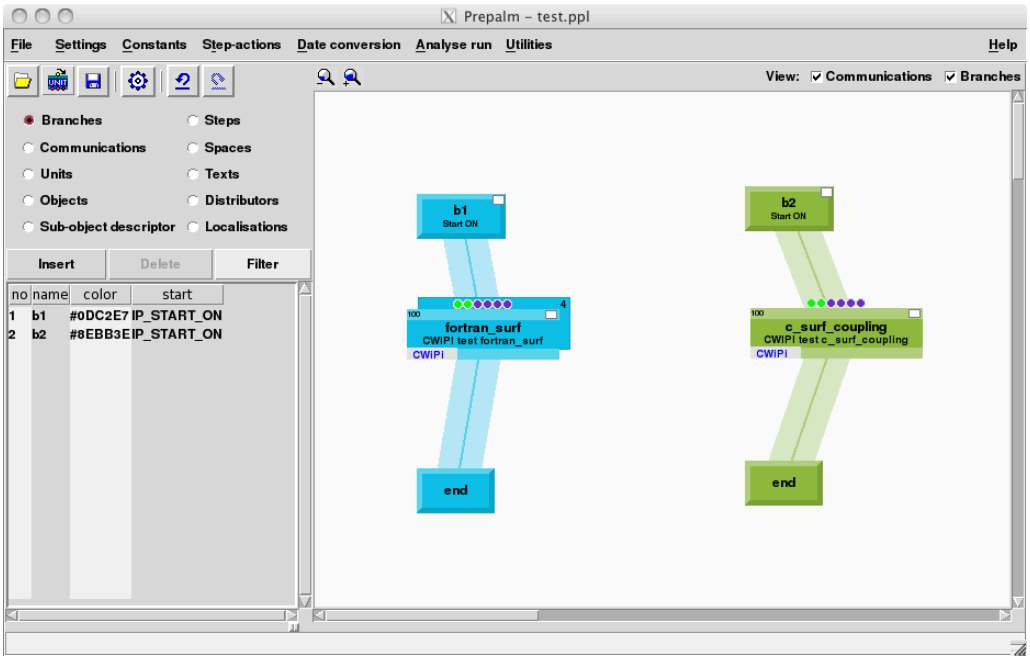


Figure 4. Canevas PrePALM avec les deux branches et les deux unités.

L'établissement de la connexion CWIPI entre les deux unités se fait ensuite en cliquant sur l'encadré CWIPI d'une des deux unités. PrePALM propose alors d'insérer un couplage entre les deux unités via les environnements de couplages décrits dans chacune des cartes d'identités. Un trait entre les unités est ainsi créé symbolisant les communications via CWIPI. En cliquant sur ce trait, on fait apparaître un nouveau menu qui permet de mettre en relation les objets de couplage des deux unités (Figure 5).

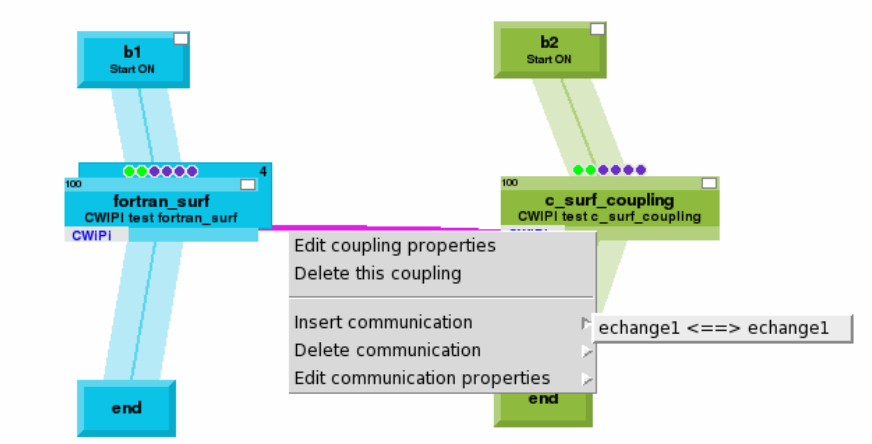


Figure 5. Mise en relation des objets de couplage des deux unités.

Dans le code des branches, définir les variables correspondant aux paramètres reçus par des PALM_Get dans les unités et insérer une région Fortran permettant le lire les fichiers contenant ces paramètres : *dataF.dat* (dans la branche de l'unité Fortran) et *dataC.dat* (dans la branche de l'unité C) comme indiqué sur la Figure 6.

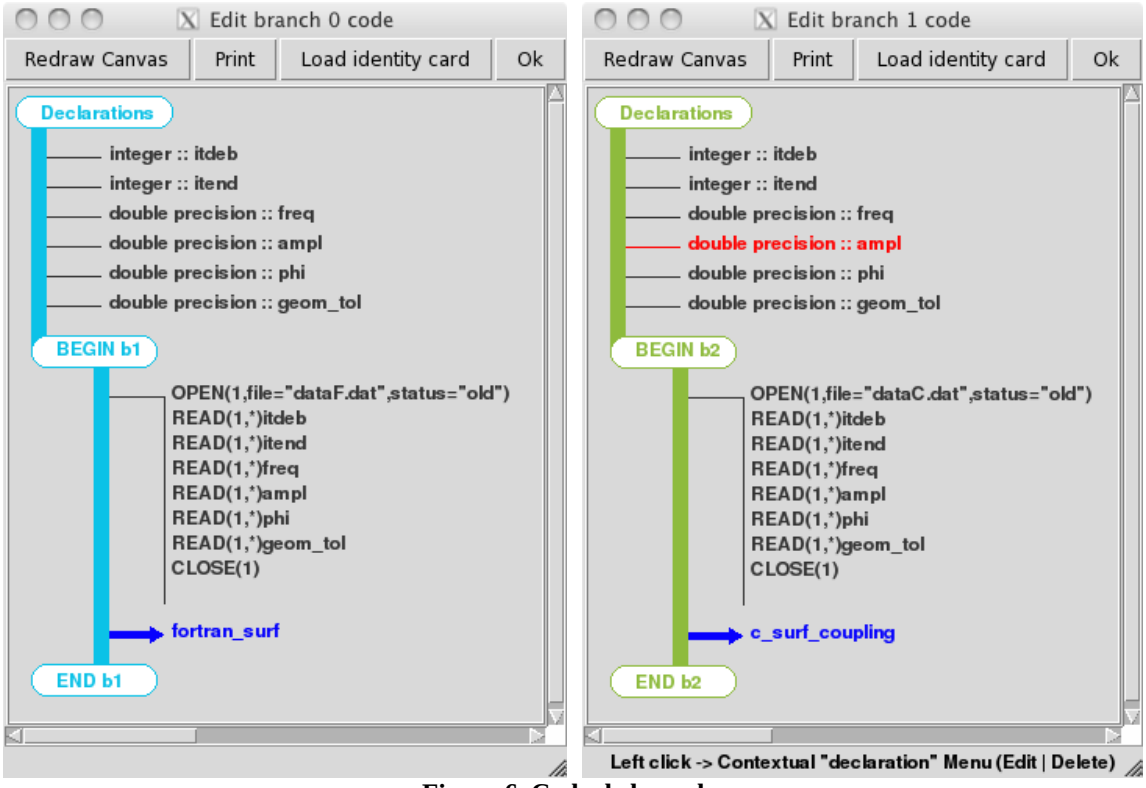


Figure 6. Code de branche

Afin de créer une communication directe entre les variables existant sur les branches et les unités, passer par une "imposition en dur" des objets (clic droit sur les plots correspondant aux objets, et *set* à la variable désirée).

L'application PrePALM est terminée. Il suffit comme pour une application OpenPALM standard de générer les fichiers de service (menu *File/Make PALM files*).

16.6 Exercice 1 : instrumentation initiale des codes à coupler

Dans le répertoire, les deux codes Fortran et C sont démunis de toute primitive de couplage. Etant donné qu'il serait fastidieux et répétitif d'instrumenter les deux codes dans le cadre de la formation, l'utilisateur peut choisir soit le code Fortran soit le code C selon ses affinités de codage. Pour la suite, nous nous concentrerons sur les explications basées sur le code Fortran. Les régions de commentaires balisés par des « To fill → End to fill » sont à instrumenter. La description des primitives PCW à utiliser se trouve à la fin de ce manuel.
Les différentes étapes de l'instrumentation sont :

- l'initialisation du couplage
- la création de l'environnement de couplage
- la définition du maillage auprès du coupleur
- l'échange croisé
- le traitement des données reçues
- la destruction de l'environnement de couplage
- la finalisation du couplage

16.6.1 Initialisation du couplage

L'initialisation du couplage est réalisée via la primitive *PCW_Init*. La primitive PCW_Dump_application_properties appelée à tout moment de l'application permet d'écrire dans les fichiers de log d'OpenPALM les propriétés de l'environnement CWIPI :

```
Local application properties

'fortran_surf' properties
  - Ranks in global MPI_comm : 0 <= ranks <= 3
  - Int Control Parameter :
  - Double Control Parameter :
  - String Control Parameter :

Distant application properties

'c_surf_coupling' properties
  - Ranks in global MPI_comm : 4 <= ranks <= 4
  - Int Control Parameter :
  - Double Control Parameter :
  - String Control Parameter :
```

16.6.2 Création de l'environnement de couplage

La création de l'environnement de couplage se fait via la primitive *PCW_Create_coupling*. Pour rappel, les codes sont parallèles, les entités échangées ont une dimension 2 et les maillages sont

statiques. Les variables suivantes définies dans le code Fortran sont utilisables directement dans l'appel de la fonction *PCW_Create_coupling* :

```
cl_coupling_name = "test2D_3"
output_format = 'Ensight Gold'
output_format_option = 'text'
```

Il est important de conserver la variable *cl_coupling_name* qui contient le nom de l'environnement de couplage. Notons que ce nom correspond à celui donné dans la carte d'identité. Ce nom sera à préciser dans diverse utilisations de primitives pour des opérations se rapportant à cet environnement.
Enfin, pour commencer, utiliser une tolérance géométrique de 0.1.

16.6.3 Définition du maillage auprès du coupleur

Dans CWIPI, il existe pour le moment une association univoque entre un maillage et un couplage. L'attache d'un maillage à un couplage est réalisée via la primitive *PCW_Define_mesh* dans laquelle on précise le nom du couplage, le nombre de sommets du maillage (*nvertex*), son nombre de cellules (*nelts*) ainsi que les tableaux de coordonnées des sommets (*coords*) et de connectivité (*connecindex* et *connec*).
Prenez le temps d'identifier comment est construit le tableau de coordonnées. Sa taille est 3*nvertex (CWIPI travaille toujours en considérant un repère cartésien de dimension 3). Les coordonnées en x, y, z du $i^{ième}$ point sont alors stockées :

- *x : coords((i-1)*3+1)*
- *y : coords((i-1)*3+2)*
- *z : coords((i-1)*3+3)*

Le tableau *connecindex* dont la dimension est le nombre de cellules plus un donne le nombre de sommet de chaque élément. Le nombre de sommet de l'élément i est donné par

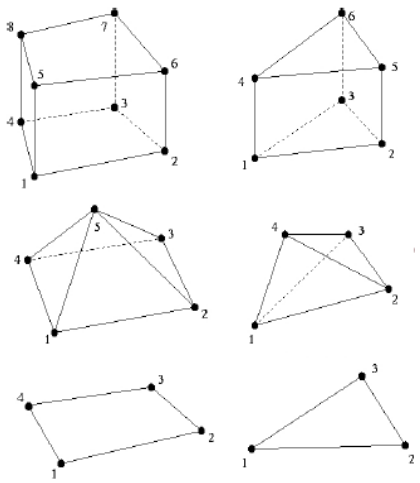
n = connecindex(i+1)-connecindex(i)

Le tableau *connec* contient pour chaque élément la référence des nœuds qui le compose.

Lorsque le maillage contient plusieurs types d'éléments, ceux-ci doivent être triés de la manière suivante :

- Eléments linéaires : arêtes
- Eléments surfaciques : triangles, quadrangles, polygones,
- Eléments volumiques : tétraèdres, pyramides, prismes, hexaèdres

La connectivité interne des éléments est la suivante :



Il est également possible de définir des polyèdres. Ceci s'effectue à l'aide de fonction *PCW_Add_polyhedra.* La définition des polyèdres est à part car plus complexe. La description des polyèdres se fait en deux temps :

- Description de la connectivité des faces polygonales composants les polyèdres (définition identique à celle effectuée pour les polygones dans *PCW_Define_mesh*)
- Description de la connectivité des polyèdres par les faces les constituant. Attention, cette connectivité doit être orientée :
  - Si la normale de la face est orientée vers l'extérieur du polyèdre, on indiquera « *numf* » où « *numf* » est le numéro de face.
  - Si la normale de la face est orientée vers l'intérieur du polyèdre, on indiquera « *-numf* »

16.6.4 Echanges croisés

Sur une même entité de couplage, et donc un même maillage, il est possible de réaliser divers échanges correspondant à des envois/réceptions de diverses variables ou des mêmes variables à des temps différents. L'envoi croisé se fait par la primitive *PCW_Sendrecv* à laquelle on spécifie :

- le nom du contexte de couplage,
- le nombre de variables contenues dans le tableau d'échange. Attention, ce tableau est entrelacé, de la même manière que les coordonnées du maillage. Si on échange 2 variables, alors (i-1)*2+1 sera la première variable associé au nœud i si on est en *cell vertex* (ou à l'élément i si on est en *cell centered*) et (i-1)*2+2 à la deuxième variable,
- l'instance de l'échange ainsi que le temps associé,
- les noms et les tableaux de champs reçus et envoyés.

En plus des champs reçus, la primitive renvoie le nombre de points non localisés. De plus, le code d'erreur de la primitive *PCW_Sendrecv* permet de savoir si l'échange de données a connu un souci majeur ou non. Ce code est interprétable par la primitive *PCW_Dump_status* qui permet d'écrire dans les fichiers de log d'OpenPALM si les échanges se sont ou non bien passés.

16.6.5 Traitement des données reçues

Dans le cas de cet exemple, le traitement des données reçues consiste à créer des sorties graphiques au format du logiciel graphique Ensight. Pour ce premier exemple, rien n'est à réaliser pour ce point.

16.6.6 Destruction de l'environnement de couplage

La destruction de l'environnement de couplage est obtenue par la primitive *PCW_Delete_coupling* en spécifiant le nom du couplage, puis la primitive *PCW_Finalize* permet de mettre fin à l'environnement CWIPI.

16.6.7 Lancement de l'application et analyse des résultats

Une fois cette instrumentation réalisée, compilez l'application en tapant *make*. Si vous êtes en mode MPI1 de PALM, le lancement de l'application se fait au travers d'un script généré directement par PrePALM *run_mpi1.sh* :

```
#!/bin/sh
mpiexec -np 1 ./palm_main : -np 1 ./main_c_surf_coupling : -np 4
./main_fortran_surf
```

Si vous êtes en mode MPI2 de PALM lancez simplement la commande :

```
> mpirun -np 1 ./palm_main
```

Lancez ce script. Vérifiez que tout s'est bien déroulé, notamment en analysant le fichier *palmdriver.log* :

```
*************************************************
 *******  Palm MP driver v4.1.0  version  *******
 *****************  MPI-1 MODE  *****************
 *************************************************


*************** Driver setup done, Beginning of PALM session ***************


 0 warning(s) has(have) been generated during execution


 Direct communications nb     : 0
 Indirect communications nb   : 0
 Explicit bufferised communications nb : 0


************************ PALM session complete ************************
```

Le fichier *palmdriver.log* informe que la session a terminé normalement et qu'il n'y a eu aucune communication qui a sollicité la bibliothèque PALM. Effectivement, les *sets* des objets par des

clics droits sur les plots n'apparaissent pas dans ce décompte, et les communications par la bibliothèque CWIPI non plus.
Le code Fortran a généré des fichiers de visualisation des données reçues qui sont stockées dans le répertoire *./CODEF/*. De plus, CWIPI permet de sortir les champs échangés (reçus et envoyés) ainsi que les partitionnements des domaines pour les unités parallèles avec décomposition de domaine. Ceux-ci se trouvent dans les répertoires

- ./CODEF/cwipi/ test2D_3.c_surf_cpl_fortran_surf_c_surf_coupling/ pour le code Fortran
- ./CODEC/cwipi/ test2D_3.c_surf_cpl_fortran_surf_c_surf_coupling/ pour le code F

Pour visualiser ces champs, utilisez le logiciel paraview (Figure 7). Notons que si le code Fortran est lancé sur 4 processus, alors il y aura 4 fichiers à charger (CHR_0000.case à CHR_0003.case), en effet ces fichiers sont générés par le code Fortran et non par CWIPI qui lui regroupe les résultats dans un même fichier.

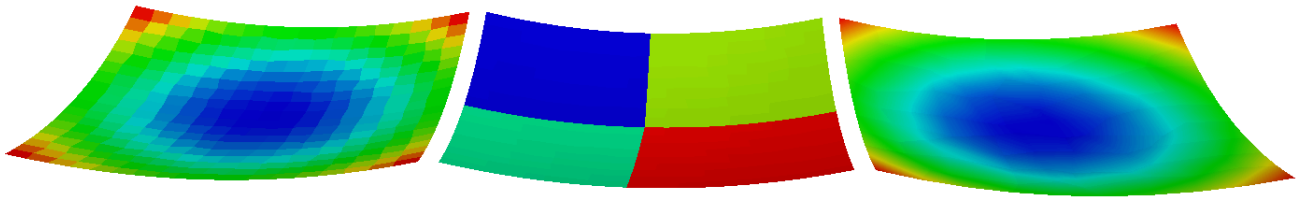


Figure 7. Surface d'échange avec de gauche a droite : champs reçus par le code Fortran (CHR_000*.case), partitionnement dans le code Fortran (données CWIPI) et champs envoyés par le code C (données CWIPI).

Pour aller plus loin et comprendre comment tout ceci fonctionne, exercez-vous à changer de variable à échanger puis à passer non pas un scalaire (stride=1) mais un vecteur de deux composantes ou plus (stride=2).

16.7 Exercice 2 : détection des points non localisés

Reprenez le couplage réalisé précédemment et dans le fichier *dataF.dat* passez l'amplitude de 0.012 à 0.015. Appréciez la surprise lorsque vous ouvrez les fichiers sortis par le code (CHR.case) : le champ est complètement perturbé (Figure 8). Ceci est lié à l'existence de points non localisés que l'on peut observer via les fichiers sortis par CWIPI (Figure 8).

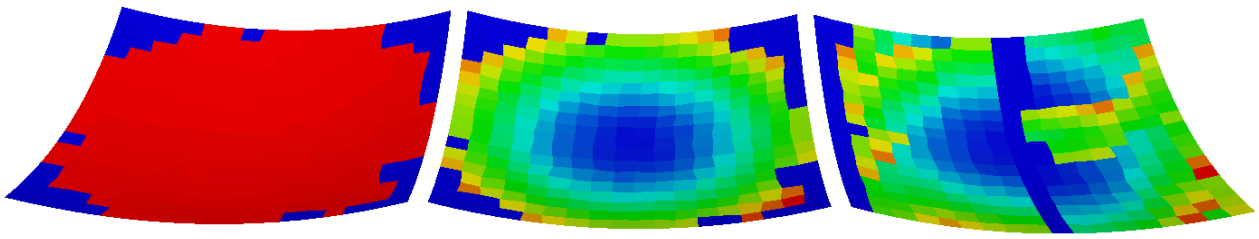


Figure 8. Surface d'échange du code Fortran avec de gauche a droite : points localisés (en rouge) et non localisés (en bleu) provenant des fichiers CWIPI, valeurs du champs aux points localisés dans les fichiers CWIPI, valeurs sorties par le code.

Après un échange de données, il est très important de vérifier si tous les points ont été bien localisés sur le maillage source. Pour cela, il est possible de faire un test sur la variable rendue par la primitive *PCW_Sendrecv*.
La primitive *PCW_Dump_notlocatedpoints* permet d'écrire dans les fichiers de log d'OpenPALM les indices des points non localisés. De plus, la primitive *PCW_Get_not_located_points* permet de récupérer dans le code source les indices des points non localisés en passant par un tableau d'entier préalablement dimensionné par rapport aux nombres de points non localisés. Il est important de noter que pour des soucis de performance, la variable *PCW_Sendrecv* ne retourne que les valeurs des points localisés, perturbant ainsi l'indexage dans le tableau de retour des grandeurs. Pour clarifier les choses, si on considère par exemple qu'on cherche à recevoir les valeurs d'une variable sur 5 nœuds et que le nœud 3 n'est pas localisé, le tableau rendu par la primitive *PCW_Sendrecv* contiendra :

- indice 1 : valeurs de la variable au nœud 1
- indice 2 : valeurs de la variable au nœud 2
- indice 3 : valeurs de la variable au nœud 4
- indice 4 : valeurs de la variable au nœud 5
- indice 5 : 0

Si l'existence de points non localisés peut ne pas poser pas de problème pour l'algorithme de couplage, il est quand même strictement nécessaire de traiter le tableau en retour en tenant compte de cet ordre de réception des données.

Pour l'exercice en cours, faire un test afin de connaître l'existence et le nombre de points non localisés. Affichez les nœuds non localisés et afficher la valeur aux nœuds localisés sous la forme : x y valeur. Notons que les maillages sont construits aléatoirement, vous n'obtiendrez donc pas forcement les mêmes nœuds pour deux lancements consécutifs.

Selon l'application, il peut être indispensable que tous les points soient localisés. Afin de lever la contrainte sur les points non localisés, augmentez la tolérance géométrique dans la primitive *PCW_Create_coupling*. Essayez de quantifier l'effet de cette augmentation de la tolérance sur le temps d'initialisation de l'échange.
Dans les cas où les domaines de calcul ne se recouvrent pas entièrement (si par exemple une surface de couplage d'un code est un sous-ensemble de la surface de couplage d'une autre), il est alors indispensable de réordonner les valeurs du tableau de réception pour qu'elles correspondent bien aux nœuds ou cellules du maillage traité. A l'aide de la primitive *PCW_Get_not_located_points*, reconstruisez le tableau *relocalvalues* qui permet de faire la sortie graphique des données dans le code Fortran.

16.8 Exercice 3 : évolution temporelle du couplage

Repartez du couplage développé précédemment et dans les fichiers de données, augmentez la valeur de la variable *itend* (gardez les paramètres identiques à ceux de l'exercice 1 pour garantir la localisation des points). Attention, la valeur doit être la même dans les deux fichiers sinon le code ayant le plus grand *itend* va se mettre en attente alors que l'autre aura fini son exécution. En exécutant l'application directement le message d'erreur suivant apparaît :

```
coupling.cxx:651: Fatal error.
coupling mesh is already created
```

Ce message vient du fait que dans cet exemple, la définition du maillage au coupleur est réalisée dans la boucle temporelle. Lors du traitement de la deuxième itération, CWIPI nous fait savoir qu'il y a un problème car un maillage a déjà été défini.
Faites le nécessaire pour ne définir qu'une seule fois le maillage au cours du calcul.
Une fois que l'application tourne, chargez les fichiers de résultats sortis par CWIPI. Notez que le nombre de solutions sorties par CWIPI dépend de ce que vous avez renseigné pour les fréquences de sortie dans la primitive *PCW_Create_coupling* (1 pour sortir les grandeurs à tous les couplages, 2 une fois tous les deux couplages, etc). De plus, il faut bien prendre garde d'incrémenter les champs time step et time values de la primitive *PCW_Sendrecv* sinon le coupleur sort avec un message d'erreur.
Remarquez que la surface sortie par CWIPI est fixe. Celle sortie par le code Fortran est mobile au cours du temps. Néanmoins, CWIPI ne voit pas cette mobilité car il n'est jamais notifié à CWIPI que la surface se déplace. Vous pouvez le vérifier en introduisant des fréquences de battement différentes dans les codes Fortran et C et vérifier qu'il n'y a jamais de points non localisés.

16.9 Exercice 4 : évolution temporelle du couplage avec surface mobile

A partir du couplage précédent, cherchez et implémentez une méthode pour prévenir le coupleur que les surfaces d'échange ont changé.

16.10 Pour aller plus loin avec CWIPI

16.10.1 Définition des points d'interpolation

Il est possible de surcharger la définition des points d'interpolation qui sont par défaut les centres des cellules ou les sommets suivant la nature du solveur. Cette fonctionnalité est utile par exemple pour un solveur de type éléments finis qui souhaite obtenir des valeurs aux points de Gauss pour alimenter un calcul d'intégrale ou encore lorsqu'il est nécessaire qu'un solveur mette à disposition un maillage différent des points sur lesquels il souhaite avoir de l'information en retour (exemple de mise à disposition de la totalité d'un maillage 3D et récupération de données uniquement sur une surface).
Cette surcharge se fait par appel à la fonction *PCW_Set_points_to_locate*.

16.10.2 Communications asynchrones

Les communications asynchrones permettent d'optimiser les échanges et de mieux contrôler les points de synchronisation entre les codes. Un échange croisé *PCW_Sendrecv* peut être remplacé par la séquence d'appels suivante :

```
PCW_irecv(…)
PCW_issend(…)

…
Autre code source
…

PCW_Wait_irecv(…)
PCW_Wait_issend(…)
```

Les points de synchronisation se situent dans les fonctions *PCW_Wait*. Pour optimiser le temps de calcul de l'application, du code source peut être défini entre les appels d'envoi et de réception (PCW_issend et PCW_irecv) et les fonctions *PCW_Wait*. Ce mécanisme est basé sur des communication asynchrones MPI, en théorie ceci permet de "recouvrir" les temps de calcul et les temps dédiés aux communications, mais dépend fortement de l'application et de l'implémentation MPI utilisée. L'autre intérêt de ce mécanisme est d'éviter les *blocages* éventuels résultant d'appels simultanés de deux envois ou de deux réceptions dans les codes couplés.

16.10.3 Définition d'une interpolation utilisateur

Il est également possible de surcharger la méthode d'interpolation en définissant une fonction utilisateur en C ou FORTRAN respectant le prototype suivant :

en langage C/C++ :

```
static void _userInterpolation(const int entities_dim,
                               const int n_local_vertex,
                               const int n_local_element,
                               const int n_local_polhyedra,
                               const int n_distant_point,
                               const double local_coordinates[],
                               const int local_connectivity_index[],
                               const int local_connectivity[],
                               const int local_polyhedra_face_index[],
                               const int local_polyhedra_cell_to_face_connectivity[],
                               const int local_polyhedra_face_connectivity_index[],
                               const int local_polyhedra_face_connectivity[],
                               const double distant_points_coordinates[],
                               const int distant_points_location[],
                               const float distant_points_distance[],
                               const int distant_points_barycentric_coordinates_index[],
                               const double distant_points_barycentric_coordinates[],
                               const int stride,
                               const cwipi_solver_type_t  solver_type,
                               const void *local_field,
                               void *distant_field)
```

ou en langage FORTRAN :

```
subroutine  userInterpolation(entitiesDim, &
                                    nLocalVertex, &
                                    nLocalElement, &
                                    nLocalPolyhedra, &
                                    nDistantPoint, &
                                    localCoordinates, &
                                    localConnectivityIndex, &
                                    localConnectivity, &
                                    localPolyFaceIndex, &
                                    localPolyCellToFaceConnec, &
                                    localPolyFaceConnecIdx, &
                                    localPolyFaceConnec, &
                                    disPtsCoordinates, &
                                    disPtsLocation, &
                                    disPtsDistance, &
                                    disPtsBaryCoordIdx, &
                                    disPtsBaryCoord, &
                                    stride, &
                                    solverType, &
                                    localField, &
                                    distantField)
```

Les données fournies par CWIPI sont toutes transmises par arguments. L'utilisateur a notamment accès au résultat de la localisation géométrique.

Une fonction utilisateur est prise en compte par CWIPI en faisant appels à :

- *PCW_Set_interplation_function* pour une fonction utilisateur définie en C
- *PCW_Set_interplation_function_f* pour une fonction utilisateur définie en Fortran

Cette fonctionnalité est intéressante pour les solveurs éléments finis ou Galerkin discontinu utilisant des éléments d'ordres élevés. Dans ce cas, la définition d'une fonction utilisateur permettra de réaliser des interpolations plus fines basées sur les fonctions de base des éléments. Une autre application est la définition d'une interpolation d'ordre élevée sur grilles cartésiennes par interpolation de Lagrange.

16.10.4 Interface CWIPI pour Python

Tout comme Palm, l'extension CWIPI peut être utilisée pour des unités Python grâce à l'interface écrite avec Cython. Les échanges d'informations se basent sur les buffers des tableaux numpy. Toutes les primitives sont regroupées dans la classe Coupling.

L'extrait de code ci-dessous devrait donner une bonne idée comment il faut utiliser les appels de primitives dans Python. Pour le détail de l'interface Python, se référer au chapitre 18 sur Python. Python utilise un objet Coupling qui contient l'attribut coupling_id, donc les méthodes PCW le connaissent déjà, on n'a pas besoin de fournir cet argument lors de l'appel aux primitives.

```
import mpi4py.MPI as MPI
import numpy as np
import palm
import PCW
import palm_user_param as pu


PCW.init()
cl_coupling_name = "test"
output_format = 'Ensight Gold'
output_format_option = 'text'
cp = PCW.Coupling(cl_coupling_name,
      PCW.COUPLING_PARALLEL_WITH_PARTITIONING,
      2, geom_tol, PCW.MOBILE_MESH, PCW.SOLVER_CELL_CENTER,
      1, output_format, output_format_option)

[....]

cp.define_mesh(nvertex, nelts, coords, connecindex, connec)

[...]

cp.sendrecv(cl_exchange_name, stride, it, time, cl_sending_field_name, values,
cl_receiving_field_name, localvalues)

cp.get_n_not_located_points()
cp.get_not_located_points()
```

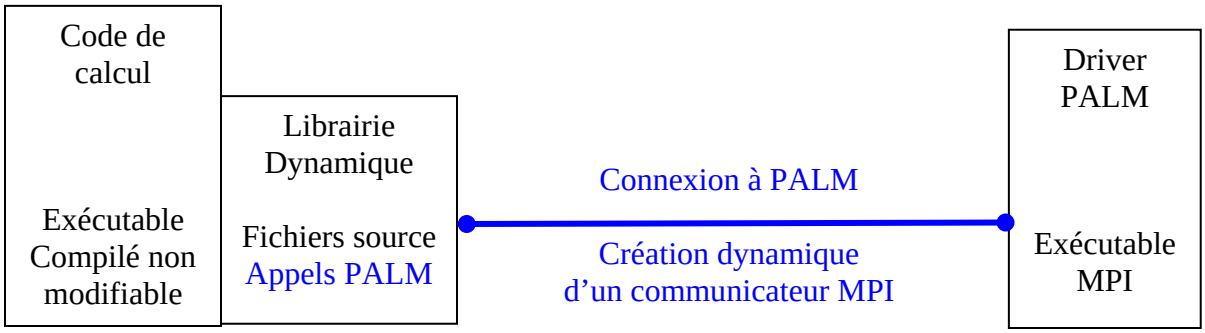
```
print 'not located: ', cp.n_not_located_points
```

Un exemple complet écrit avec Python basé sur l'application de ce chapitre se trouve dans le répertoire corrige_python.

17 Connexion d'un code externe à une application PALM.

17.1 Généralités

Suite à la difficulté de « PALMer » certains codes de calcul dont on n'a pas les programmes sources, un mode de fonctionnement par librairie dynamique a été développé pour le coupleur PALM. Certains codes commerciaux ont en effet ce mode d'ouverture par des fonctions utilisateur compilées a postériori sous forme de librairie dynamique.



Rappelons qu'une librairie dynamique (fichier .so dans le monde unix, .dll dans le monde Windows) est une collection de sous-programmes liée à une application par son nom (et éventuellement son chemin), seul le lien et le prototype des fonctions sont connus par l'exécutable. Lors de l'exécution du programme principal, la librairie est chargée dynamiquement en mémoire lors du premier appel de celle-ci (elle peut tout aussi bien ne pas être chargée si le programme ne l'appelle pas lors de son processus d'exécution). Cette technique permet entre autre de pouvoir proposer des fonctions qui sont modifiables par l'utilisateur dans la mesure où elles n'interviennent pas directement dans le programme principal. Un autre avantage des librairies dynamiques est de conduire à des exécutables plus légers en occupation disque car il n'y a pas recopie du code binaire de la librairie dans le programme exécutable.

On a vu jusqu'à présent qu'une unité PALM était un subroutine fortran ou une fonction C sans arguments. Le programme principal est créé par PrePALM (fichier de service main_*.c), ce qui a l'avantage (du point de vu utilisateur) de ne pas avoir à appeler des fonctions telles que PALM_Init ou PALM_Finalize. En fait, ces appels sont fait directement par le main créé par PrePALM. On a vu également que cette technique permettait aussi à PALM de constituer des blocs autour de plusieurs unités, les blocs permettent de concaténer plusieurs unités dans un seul exécutable, ce qui serait impossible à faire si les codes contenaient leur programme principal. Cette encapsulation a une forte contrainte pour les codes car elle suppose de disposer des programmes source du programme principal, elle peut donc paraître relativement intrusive pour adapter les codes de calcul à PALM.

Pour coupler des codes de calcul dont on ne pourrait pas disposer du programme principal, l'idée est de les lancer indépendamment du coupleur PALM (sans utiliser la primitive MPI_Comm_Spawn) et de créer un communicateur MPI entre le coupleur et la librairie dynamique du code de calcul comme présenté sur la figure ci-dessus.

17.2 Principe de fonctionnement

La connexion à PALM se fait par la primitive PALM_Connect() et la déconnexion par la primitive PALM_Disconnect(), pour les codes parallèles ces deux primitives doivent être appelées par tous les processus du code de calcul. Une fois la primitive PALM_Connect appelée, l'utilisateur bénéficie de l'environnement PALM et peut donc appeler toutes les autres primitives PALM comme les Get/Put, les écritures dans les fichiers de sortie, etc.

Remarque : PALM_Connect repose sur les fonctionnalités client/serveur de MPI-2, seul le mode MPI-2 de PALM est supporté.

Contraintes :
1) Si le code de calcul fait des appels à MPI, les librairies dynamiques doivent être compilées avec la même version de bibliothèque MPI. C'est une contrainte forte car les distributions du code de calcul « commercial » ne sont hélas pas forcément disponibles avec des versions de MPI compatibles avec PALM. Si plusieurs codes de calcul sont couplés de la sorte, il faut que tous les codes soient compilés avec la même version de MPI.

2) La couche MPI que vous utilisez doit être compilée avec l'option qui implémente les librairies dynamiques, **attention c'est souvent une option à l'installation de MPI**.

3) PALM doit être installé avec l'option **--with-shared_lib** au configure de PALM (se reporter au chapitre sur l'installation de PALM)

Pour illustrer le fonctionnement, vous trouverez dans le répertoire chapter_16 deux exemples complets de couplage d'un code externe avec PALM. Que le code de calcul soit parallèle ou non, le principe est identique.

17.3 Connexion d'un code non parallèle avec PALM.

Le code de calcul est un modèle jouet qui simule l'appel à des fonctions utilisateur. Vous trouverez le code source de ce modèle jouet écrit en langage C dans le fichier code.c du répertoire chapter_16/connect_code. Le fichier source de code.c est le suivant :

```
/* Ce code est independant de PALM, aucune reference à PALM n'apparait

   Il appelle une librairie dynamique udf_* (pour User Defined Function)
   c'est dans ces fonctions que les appels à PALM sont effectues
```

```
   ce code possède une boucle interne sur le temps, dont le nombre d'itérations
   est retourné par la fonction udf_init

   une fois ce nombre d'iterations connu il calcule un champs (100 réels)
   à chaque iteration chaque valeur du champs est incrémentée de 1.

   Pour compiler ce code, il faut déjà compiler les fonctions utilisateur (vierge
de tout appel à PALM)
*/

/* #include "mpi.h" */
#include <stdio.h>
#include <string.h>


int main(int argc, char *argv[] ) {
  int i, il_time, il_err;
  int il_maxtime;
  float rla_field[100];

  printf("CODE : debut\n");


  /* appel de la librairie dynamique à l'initialisation */
  il_err = udf_init(&il_maxtime);

  /* initialisation du champ à t = 0 */
  for (i = 0; i < 100; i++) {rla_field[i] = i-1;}


  /* boucle sur le temps */
  for (il_time = 0 ; il_time <= il_maxtime ; il_time++) {

       for (i = 0; i < 100; i++) {rla_field[i] = rla_field[i]+1.;}
       printf("CODE : iter %i rla_field[0] = %f , rla_field[1] = %f\n",
            il_time, rla_field[0],rla_field[1]);

       /* appel de la librairie dynamique à chaque itération */
       il_err = udf_inloop(rla_field, &il_time);

  }


  /* appel de la librairie dynamique à la fin du programme */
  il_err = udf_end();

  printf("CODE : j'ai fini\n\n");

}
```

Ce code de calcul appelle trois fonctions utilisateur, la première à l'initialisation, la seconde à chaque itération d'une boucle sur le temps, la dernière à la fin du programme. Ces fonctions utilisateurs sont compilées sous forme de librairie dynamique contenue dans le fichier udf_vierge.c. Au départ ni le

code ni la librairie ne font référence à PALM. Le fichier make_code permet de compiler ce code qui, comme il n'est pas parallèle, ne fait pas d'appels MPI.

Vous pouvez compiler ce code en entrant la commande :

```
>make -f make_code
```

Et lancer le programme par la commande :

```
>./code
```

Pour connecter ce code il est maintenant nécessaire de modifier les fonctions utilisateur et de créer la librairie dynamique. Pour commencer, comme toute unité PALM, l'utilisateur doit définir une carte d'identité, elle est quasiment identique aux unités classique sauf pour le champ -functions où il faut définir la commande à lancer pour lancer le code de calcul, on utilise ici la commande mpirun car même si le code ne faisait pas référence à MPI au départ, il va devenir un code MPI via la librairie dynamique.

```
/*PALM_UNIT -name code\
           -functions {SH {mpirun -np 1 ./code&}}\
           -parallel no \
           -comment {test code independant}
*/
```

La syntaxe de lancement du code (mpirun –np 1 ./code) dépend bien entendu de la librairie MPI utilisée, on tourne ici avec lammpi, cette commande est donnée à titre d'exemple. On peut très bien donner au champ –functions le nom d'un script de lancement ou même rien du tout et faire en sorte que le programme à lancer soit exécuté au moment opportun (par exemple avant le lancement de PALM).
Le reste de la carte d'identité est tout à fait classique avec des déclarations d'espaces et d'objets PALM :

```
/*PALM_SPACE -name vect_real\
             -shape (100)\
             -element_size PL_REAL\
             -comment {100 simple precision}
*/


/*PALM_OBJECT -name max_time\
              -space one_integer\
              -intent IN\
              -comment {time for get}
*/

/*PALM_OBJECT -name vector\
              -space vect_real\
              -intent INOUT\
              -time ON\
              -comment {vector inout in code}
*/

/*PALM_OBJECT -name status\
              -space one_integer\
              -intent OUT\
```

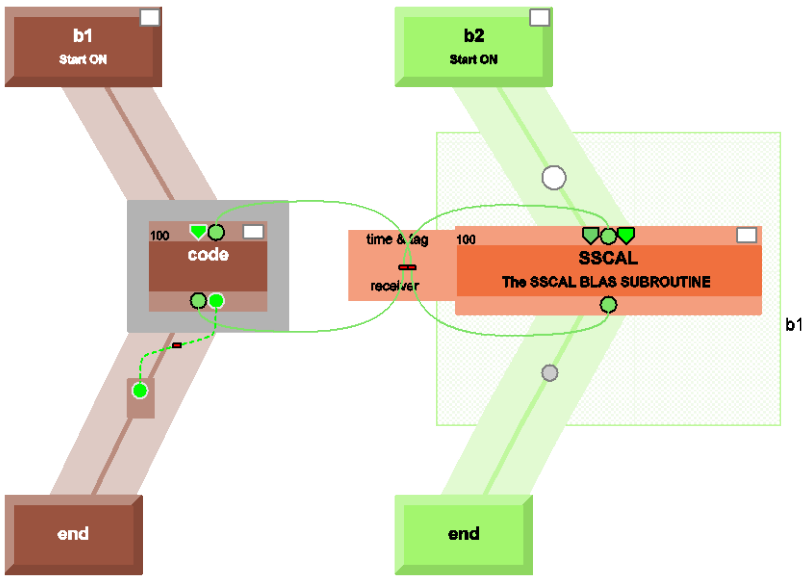
```
              -comment {unite termine}
*/
```

Dans le corps des fonctions utilisateur ci dessous il faut maintenant ajouter les appels à MPI et à PALM, les parties en bleu on été ajoutées :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

/* interface PALM */
#include "palmlibc.h"

/* variables destinées aux appels PALM */
int il_tag = PL_NO_TAG;
int il_time = PL_NO_TIME;
int il_err = 0;
char cla_obj[PL_LNAME], cla_space[PL_LNAME];


/* fonction utilisateur destinées à retourner le nombre d'itérations */
/* appelée une seule fois par le code, à l'initialisation            */

int udf_init (int *max_time) {
  int argc=0;
  char **argv;

  /* initialisation MPI */
  il_err = MPI_Init(&argc, &argv);
  /* connection du code à PALM */
  il_err = PALM_Connect();

  /* valeur defaut si le get n'est pas connecté */
  *max_time = 15;
  /* remarquer qu'après le PALM_Connect, on dispose de PL_OUT, fichier de sortie
de PALM */
  PALM_Write(PL_OUT,"==== > udf_init : max_time (valeur defaul) = %i",*max_time);

  /* appel classique d'un PALM_GET */
  sprintf(cla_obj,"max_time");
  sprintf(cla_space,"one_integer");
  il_err = PALM_Get(cla_space, cla_obj, &il_time, &il_tag, max_time);
  PALM_Write(PL_OUT,"==== > udf_init : max_time apres get = %i",*max_time);
  return 0;

}

/* fonction utilisateur              */
/* appelée a chaque itération du code */

int udf_inloop(float *rda_field, int *id_time) {

  sprintf(cla_obj,"vector");
  sprintf(cla_space,"vect_real");
  /* simple get put du champ */
  il_err = PALM_Put(cla_space, cla_obj, id_time, &il_tag, rda_field);
  il_err = PALM_Get(cla_space, cla_obj, id_time, &il_tag, rda_field);
  return 0;

}
```

```
/* fonction utilisateur*/
/* appelée une seule fois par le code, à la fin du programme */

int udf_end() {
  int status;

  sprintf(cla_obj,"status");
  sprintf(cla_space,"one_integer");
  il_time = PL_NO_TIME;
  status = 0;
  il_err = PALM_Put(cla_space, cla_obj, &il_time, &il_tag, &status);

  /* fin de la connexion avec PALM */
  il_err = PALM_Disconnect();
  /* fin de MPI */
  il_err = MPI_Finalize();
  return 0;

}
```

Les appels à PALM_Put/Get, PALM_Write, etc, sont strictement identiques à des unités classiques, il n'y aucune limite avec ce mode de fonctionnement. Au niveau de l'application PrePALM, la seule chose à préciser est le type d'exécution de l'unité. Dans la boite de dialogue de l'unité, il faut choisir EXTERN-TO-CONNECT pour le champ Execution mode. Ceci a pour effet de faire apparaitre une bordure grise autour de l'unité code.



Dans notre cas les champs produits par le modèle sont multipliés par une constante à chaque itération sur le temps, on utilise une boite d'algèbre SSCAL pour faire ce travail.

Remarque : Cette technique de connexion est basée sur les fonctionnalités client/serveur de MPI_2, sur certaines machines (NEC par exemple) l'implémentation repose sur un couche Tcp/ip, l'application n'aura donc pas nécessairement les mêmes performances que celles observées avec une unité PALM classique en remplaçant Program par Subroutine. Dans le cas où l'utilisateur dispose des codes source, il est donc plus judicieux de procéder autrement que par une connexion. En outre les unités « connectées » ne peuvent pas être mises dans des blocs.

17.4 Connexion d'un code parallèle avec PALM.

Dans le répertoire chapter_16/connect_code_par vous trouverez un exemple complet de connexion d'un code parallèle. Les différences avec l'exemple précédent sont :

- la présence d'un distributeur (dist_d3x100.f90) pour le tableau à échanger car le vecteur est distribué sur tous les procs.
- L'absence d'appel à MPI_Init et MPI_Finalize dans la fonction de la librairie dynamique car ces appels sont réalisés dans le programme principal qui est parallèle.

```
/*PALM_UNIT -name code\
           -functions {SH {run_code&}}\
           -object_files {code.o} \
           -parallel mpi \
           -minproc 1\
           -maxproc 100\
           -comment {test code independant}
*/

/*PALM_SPACE -name vect_real\
             -shape (100*ip_nbproc)\
             -element_size PL_REAL\
             -comment {100 simple precision par proc}
*/


/*PALM_DISTRIBUTOR -name d3x100\
                 -type custom\
                 -shape (ip_nbproc*100)\
                 -nbproc ip_nbproc\
                 -function d3x100\
                 -object_files {dist_d3x100.o}\
                 -comment {}
*/

/*PALM_OBJECT -name max_time\
              -space one_integer\
              -intent IN\
              -localisation REPLICATED_ON_ALL_PROCS\
              -comment {time for get}
*/
```

```
/*PALM_OBJECT -name vector\
              -space vect_real\
              -distributor d3x100\
              -localisation DISTRIBUTED_ON_ALL_PROCS\
              -intent INOUT\
              -time ON\
              -comment {inout distributed vector in code}
*/

/*PALM_OBJECT -name status\
              -space one_integer\
              -intent OUT\
              -localisation REPLICATED_ON_ALL_PROCS\
              -comment {unite termine}
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

/* interface PALM */
#include "palmlibc.h"

/* variables destinées aux appels PALM */
int il_tag = PL_NO_TAG;
int il_time = PL_NO_TIME;
int il_err = 0;
char cla_obj[PL_LNAME], cla_space[PL_LNAME];


/* fonction utilisateur destinée à retourner le nombre d'itérations */
/* appelée une seule fois par le code, à l'initialisation           */

int udf_init (int *max_time) {

  /* connection du code à PALM */
  il_err = PALM_Connect();

  /* valeur defaut si le get n'est pas connecté */
  *max_time = 15;
  /* remarquer qu'après le PALM_Connect, on dispose de PL_OUT, fichier de sortie
de PALM */
  PALM_Write(PL_OUT,"==== > udf_init : max_time (valeur defaul) = %i",*max_time);

  /* appel classique d'un PALM_GET */
  sprintf(cla_obj,"max_time");
  sprintf(cla_space,"one_integer");
  il_err = PALM_Get(cla_space, cla_obj, &il_time, &il_tag, max_time);
  PALM_Write(PL_OUT,"==== > udf_init : max_time apres get = %i",*max_time);
  return 0;

}

/* fonction utilisateur               */
/* appelée a chaque itération du code */

int udf_inloop(float *rda_field, int *id_time) {
  int il_rank;

  sprintf(cla_obj,"vector");
  sprintf(cla_space,"vect_real");
  il_err = MPI_Comm_rank(MPI_COMM_WORLD,&il_rank);
```

```
  /* simple get put du champ */
  il_err = PALM_Put(cla_space, cla_obj, id_time, &il_tag, rda_field);
  il_err = PALM_Get(cla_space, cla_obj, id_time, &il_tag, rda_field);
  return 0;

}



/* fonction utilisateur*/
/* appelée une seule fois par le code, à la fin du programme */

int udf_end() {
  int status;
  int il_rank;
  il_err = MPI_Comm_rank(MPI_COMM_WORLD,&il_rank);
  sprintf(cla_obj,"status");
  sprintf(cla_space,"one_integer");
  il_time = PL_NO_TIME;
  status = 0;
  il_err = PALM_Put(cla_space, cla_obj, &il_time, &il_tag, &status);
  /* fin de la connexion avec PALM */
  il_err = PALM_Disconnect();
  return 0;

}
```

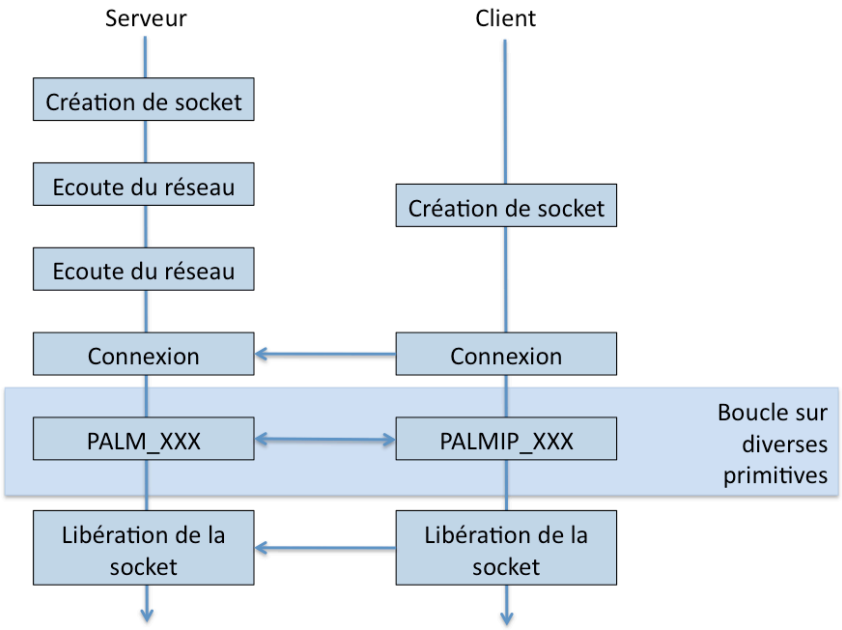
17.5 Pour aller plus loin : connexion par IP d'un code externe

Comme nous l'avons vu, une limitation forte du type de connexion présentée précédemment vient du fait que si le code de calcul fait des appels à MPI, les librairies dynamiques de ce code et de PALM doivent être compilées avec la même version de bibliothèque MPI. Afin de lever cette contrainte, la connexion d'un code à l'environnement PALM a été étendue via le passage par les sockets IP.
La notion de sockets a été introduite dans les distributions de Berkeley (un fameux système de type UNIX, dont beaucoup de distributions actuelles utilisent des morceaux de code), c'est la raison pour laquelle on parle parfois de sockets BSD (*Berkeley Software Distribution*). Il s'agit d'un modèle permettant la communication inter processus afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP. On distingue deux modes de communication :

- le mode connecté qui utilise le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, l'adresse de destination n'est pas à re-spécifier à chaque envoi de données.
- Le mode non connecté qui utilise le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, aucun accusé de réception n'est donné.

C'est le mode connecté qui est utilisé dans PALM. Comme dans le cas de l'ouverture d'un fichier, la communication par socket utilise un descripteur pour désigner la connexion sur laquelle on envoie ou reçoit les données. Ainsi, la première opération à effectuer consiste à appeler une fonction créant un socket et retournant un descripteur identifiant de manière unique la connexion. Un socket est la combinaison d'une adresse IP de machine et d'un numéro de port, adresse de connexion sur la machine. Cette combinaison devient alors une adresse unique au monde permettant une connexion univoque.

Un serveur est alors à l'écoute de messages éventuels. Comme expliqué par la suite, ce serveur est une unité miroir du code externe qui est intégré au schéma de couplage à la place du code. Le code externe joue le rôle de client : il envoie des requêtes au serveur. Afin de bien distinguer un code client, les primitives PALM_XXX ou PCW_XXX ont été interfacées en PALMIP_XXX ou PCWIP_XXX. Lors de l'appel d'une primitive PALMIP_XXX, le client envoie au serveur une requête lui ordonnant d'exécuter en *remote* la primitive PALM_XXX. Le serveur exécute les primitives PALM et retourne le code d'erreur éventuel au client. De plus, les arguments *IN* des primitives PALM sont envoyés au serveur et les arguments *OUT* des primitives PALM sont reçus du serveur. La fonction PALM_Write(PL_OUT,...) fonctionne pour le C mais l'écriture Fortran Write(PL_OUT,...) ne peut pas fonctionner car le fichier PL_OUT est ouvert par le serveur, écrit en C.
Dans le cas d'une communication PALM locale, l'utilisateur a la responsabilité de gérer la taille du buffer utilisé pour les primitives Get/Put/Dump pour éviter de dépasser la mémoire allouée. Pour l'interface Palm par IP, il faut en plus gérer la taille du message passé à travers le socket. C'est pour cela que la primitive PALMIP_Get/Put/Dump_sized prend en plus un argument *size* qui spécifie la taille (en octets) du message passé par socket. Celui-ci doit correspondre à la taille du buffer local pour éviter des fautes de segmentation. Si l'application n'utilise pas d'objets distribués ou des sous-objets, on peut utiliser la primitive standard PALMIP_Get/Put/Dump qui récupère la taille statique de l'espace par un appel à PALM_Space_get_size. Par contre dans le cas d'objets distribués ou de sous-objets, la taille du Get/Put/Dump ne correspond pas à la taille de l'espace et il faut éviter d'utiliser les primitives simples PALMIP_Get/Put/Dump.
La figure ci dessous présente le principe général de fonctionnement d'une application PALM avec connexion d'un code externe par la couche IP. Notons dès à présent que le code externe est représenté dans l'application PALM par une unité *miroir*. Cette unité *miroir* joue le rôle de serveur au sein de l'application PALM, répondant aux requêtes du client, le code externe. L'utilisateur n'a pas à écrire l'unité miroir, cette dernière est fournie par PrePALM.



Principe de fonctionnement d'une application PALM avec connexion d'un code externe via la couche IP.

Pour illustrer l'utilisation de la couche IP dans PALM, cette section reprend le cas de la section 17.4, « Connexion d'un code parallèle avec PALM ». Dans le répertoire chapter_17/connect_code_par_IP, vous trouverez l'exemple complet détaillé ci-dessous. Dans ce tutorial, l'application PALM et le code à connecter tournent sur la même machine.

Dans le répertoire chapter_17/connect_code_par_IP/code se trouvent les sources du code. Il s'agit strictement du même code que celui de la section 17.4. La compilation s'effectue simplement en tapant *make*. L'exécutable *code* et la librairie dynamique *udf_so.so*.

Les fonctions utilisateur sont modifiées de telle sorte à faire appel aux fonctions de la libraire IP pour générer la connexion à l'application et procéder aux échanges. La carte d'identité de l'unité est rappelée ci dessous. Comparée à celle de la section 17.4, seuls les champs *functions* et *object_files* de l'attribut *PALM_UNIT* sont modifiés. Comme nous l'avons mentionné, la partie serveur de la couche IP est gérée par une unité miroir qui fait le pont entre le monde PALM et l'extérieur. De ce fait, la description de l'unité indique de manière standard les accès aux fichiers correspondant *mirror_code.c* :

```
/*PALM_UNIT -name code\
           -functions {C mirror_code}\
           -object_files {mirror_code.o} \
           -parallel mpi \
           -minproc 1\
           -maxproc 100\
           -comment {test code independant}
*/

/*PALM_SPACE -name vect_real\
             -shape (100*ip_nbproc)\
             -element_size PL_REAL\
             -comment {100 simple precision par proc}
*/


/*PALM_DISTRIBUTOR -name d3x100\
                 -type custom\
                 -shape (ip_nbproc*100)\
                 -nbproc ip_nbproc\
                 -function d3x100\
                 -object_files {dist_d3x100.o}\
                 -comment {}
*/

/*PALM_OBJECT -name max_time\
              -space one_integer\
              -intent IN\
              -localisation REPLICATED_ON_ALL_PROCS\
              -comment {time for get}
*/

/*PALM_OBJECT -name vector\
              -space vect_real\
              -distributor d3x100\
              -localisation DISTRIBUTED_ON_ALL_PROCS\
              -intent INOUT\
              -time ON\
              -comment {inout distributed vector in code}
*/
```

```
/*PALM_OBJECT -name status\
              -space one_integer\
              -intent OUT\
              -localisation REPLICATED_ON_ALL_PROCS\
              -comment {unite termine}
*/
```

Le code de l'unité *miroir* est identique quelque soit l'application. Les parties en bleu soulignent les champs qui peuvent être modifiés par l'utilisateur. On note notamment la définition des ports de la machine sur lesquels l'unité serveur va attendre les requêtes. Chaque processus d'un code parallèle se verra attribuer un port particulier, distingué ici par le numéro de rang MPI.

```
#include "iplib_server.h"
#include "mpi.h"
#include "palmlibc.h"
static int listen_port=5000;
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
static t_server s_server;
void mirror_code() {
  int il_err,rank;
  il_err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("mirror_code started, rank %i\n",rank);
  fflush(stdout);
  fflush(stdout);
/*    if(Palm_On_Ip_CreateServer(listen_port,PALMONIP_SVRFLAG_VERBOSE,&s_server))
{ */

  if(Palm_On_Ip_CreateServer(listen_port+rank,0,&s_server)) {
    return;
  }
  if(Palm_On_Ip_Run(&s_server)) {
    return;
  }
  if(Palm_On_Ip_KillServer(&s_server)) {
    return;
  }
  PALM_Write(PL_OUT,"mirror_code stopped\n");
  fflush(stdout);
}
```

Les fonctions utilisateur sont présentées ci dessous. On remarque notamment que pour établir une connexion de manière univoque, il est important de préciser l'adresse IP de la machine où le serveur est exécuté ainsi qu'un numéro de port. Dans cet exemple, nous verrons que l'application PALM génère le fichier *code.palm_connect* qui contient l'adresse IP de la machine sur laquelle le serveur tourne. Ce choix est particulièrement intéressant pour les calculs sur les machines parallèles où l'adresse IP est celle du nœud où l'application tourne et est donc susceptible de changer d'une exécution à l'autre. De plus, dans le cas de cet exemple, le port a été imposé en dur coté serveur et client. Une paramétrisation et un passage de l'information par fichier est tout à fait envisageable pour permettre de simplifier la gestion de cas complexes. Concernant les appels aux primitives PALM, on constate qu'il s'agit des même appels avec, comme énoncé précédemment, la spécification *IP* dans le nom de la primitive qui permet d'identifier que l'on traitre un code client d'une application PALM.

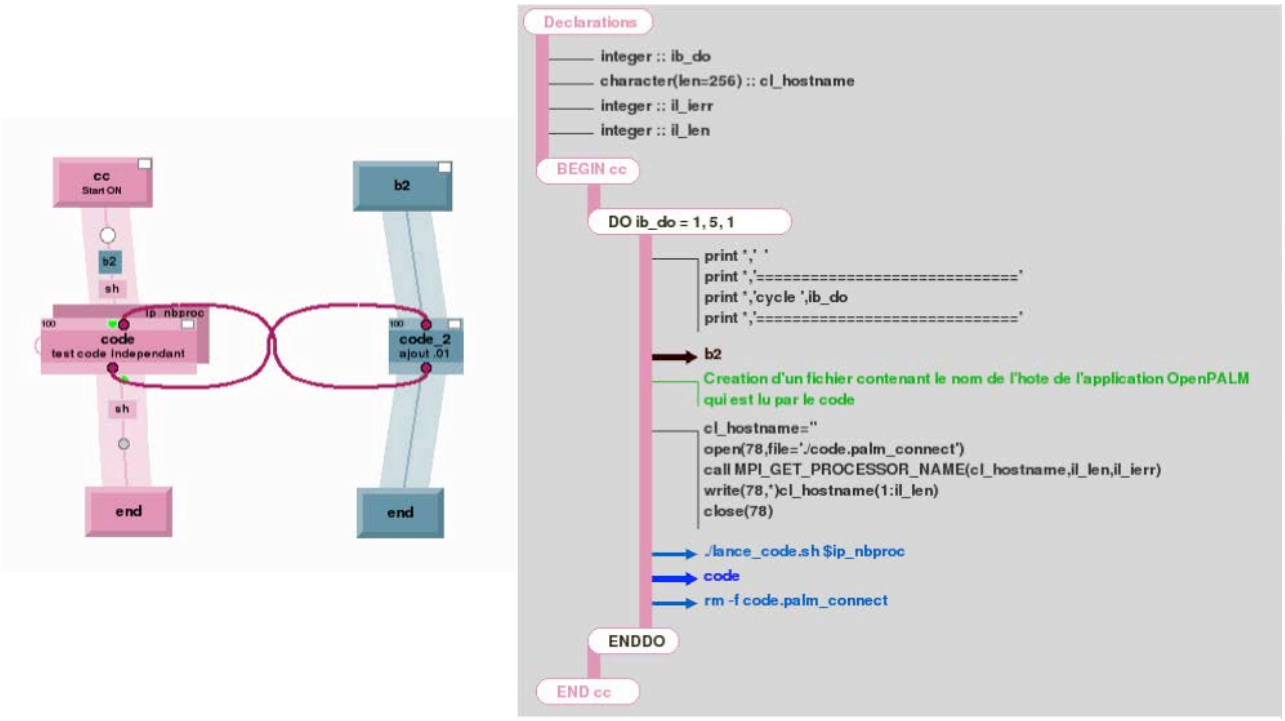
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

/* interface PALM */
```

```
#include "iplib_client.h"

/* variables destinées aux appels PALM */
int il_tag = PL_NO_TAG;
int il_time = PL_NO_TIME;
int il_err = 0;
char cla_obj[PL_LNAME], cla_space[PL_LNAME];


/* fonction utilisateur destinées à  retourner le nombre d'itérations */
/* appelée une seule fois par le code, à l'initialisation            */

int udf_init (int *max_time) {
  char palm_on_ip_host[PL_LNAME];
  int palm_on_ip_port;
  int il_rank;
  int lev;
  FILE * pFile;

  pFile = fopen ("code.palm_connect","r");
  fscanf(pFile, "%s\n", palm_on_ip_host);
  printf("Code indep connecte avec : %s\n", palm_on_ip_host);
  fclose (pFile);

  il_err = MPI_Comm_rank(MPI_COMM_WORLD,&il_rank);
  palm_on_ip_port = 5000 + il_rank;


  /* connection du code à PALM */
  il_err                                                                    =
PALMIP_Connect(palm_on_ip_host,palm_on_ip_port,PALMONIP_CLIENTFLAG_VERBOSE);

  il_err = PALMIP_Verblevel_get(PL_VERB_COMM, &lev);
  PALMIP_Write(PL_OUT,"=====>verbosite communications : %i ",lev);
  lev = 50;
  il_err = PALMIP_Verblevel_set(PL_VERB_COMM, &lev);
  PALMIP_Write(PL_OUT,"=====>verbosite communications : %i ",lev);


  /* valeur defaut si le get n'est pas connecté */
  *max_time = 15;
  /* remarquer qu'après le PALM_Connect, on dispose de PL_OUT, fichier de sortie
de PALM */
  il_err  =  PALMIP_Write(PL_OUT,"====  >  udf_init  :  max_time  (valeur  defaul)  =
%i\n",*max_time);

  /* appel classique d'un PALM_GET */
  sprintf(cla_obj,"max_time");
  sprintf(cla_space,"one_integer");
  il_err = PALMIP_Get(cla_space, cla_obj, &il_time, &il_tag, max_time);
  PALMIP_Write(PL_OUT,"==== > udf_init : max_time apres get = %i\n",*max_time);
  return 0;
}

/* fonction utilisateur              */
/* appelÃ©e a chaque itÃ©ration du code */

int udf_inloop(float *rda_field, int id_size, int *id_time) {
  int il_rank;

  sprintf(cla_obj,"vector");
  sprintf(cla_space,"vect_real");
  il_err = MPI_Comm_rank(MPI_COMM_WORLD,&il_rank);
```

```
  /* simple get put du champ */
  il_err  =  PALMIP_Put_sized(cla_space,  cla_obj,  id_time,  &il_tag,  rda_field,
&id_size);
  il_err  =  PALMIP_Get_sized(cla_space,  cla_obj,  id_time,  &il_tag,  rda_field,
&id_size);
  return 0;
}


/* fonction utilisateur*/
/* appelÃ©e une seule fois par le code, Ã  la fin du programme */

int udf_end() {
  int status;
  int il_rank;
  il_err = MPI_Comm_rank(MPI_COMM_WORLD,&il_rank);
  sprintf(cla_obj,"status");
  sprintf(cla_space,"one_integer");
  il_time = PL_NO_TIME;
  status = 0;
  il_err = PALMIP_Put(cla_space, cla_obj, &il_time, &il_tag, &status);
  /* fin de la connexion avec PALM */
  il_err = PALMIP_Disconnect();
  return 0;
}
```

Pour compiler les fonctions utilisateurs ainsi modifiées et créer la librairie dynamique correspondante *udf_so.so*, taper *make lib_dyn_udf* dans le terminal pointant vers la distribution LAMMPI.

L'application PALM, connect_code_par_IP/application_openpalm/boucle.ppl, est éditée à l'aide de PrePALM. Comparée à celle de la section 17.4, le code indépendant est représenté directement par l'unité miroir dans le caneva. L'appel à la primitive *MPI_GET_PROCESSOR_NAME* dans une région fortran de la branche sur laquelle tourne le code serveur permet d'identifier l'adresse IP et de créer le fichier *code.palm_connect*. Le code de calcul client est lancé avec le script *lance_code.sh*.



Application PrePALM pour la connexion par couche IP

Le script *lance_code.sh* permet de créer un lien symbolique vers la librairie dynamique dans le répertoire d'exécution de l'application (cette procédure peut être évitée en positionnant correctement la variable d'environnement LD_LIBRARY_PATH).

```
#!/bin/sh
#
if test -f udf_so.so
then
  echo "la librairie dynamique est presente"
else
  ln -sf ../code/udf_so.so .
fi
mpirun -np $1 ../code/code &
```

La compilation de l'application se fait de manière standard dans le répertoire application_openpalm. Pour exécuter ce couplage, il suffit d'exécuter le driver de PALM.
> mpirun –np 1./palm_main

A noter que ce mode de connexion fonctionne aussi bien en mode MPI1 que MPI2 de PALM, en prenant garde à remplacer MPI_COMM_WORLD par PL_COMM_EXEC dans le code miroir avec une inclusion de la bibliothèque PALM.

17.6 Cas de couplage par IP OpenMPI - OpenMPI

Une limitation de la librairie OpenMPI consiste à empecher l'appel imbriqué de runmpi. Un runmpi ne peut pas en lancer un deuxième. Il faut donc utiliser un programme avec un appel à MPI_spawn.

Le répertoire openmpi_spawn contient un exemple de couplage par IP avec un spawn du code externe fait dans la branche PALM.

17.7 Exportation de la librairie client IP pour OpenPALM

Pour un fonctionnement sur deux machines hétérogènes, il n'est pas nécessaire d'installer la librairie PALM sur le client, il suffit juste d'installer le client IP.

Une fois la distribution de palm désarchivée, vous pouver trouver l'archive de ce qui est juste nécessaire pour le client dans le fichier :

PALM_MP/SRC/IPLIB/IPLIB_CLIENT.tgz

Pour son installation sur la machine distante, commencez par désarchiver ce fichier.

```
> tar xvfz PALM_MP/SRC/IPLIB/IPLIB_CLIENT.tgz
> cd IPLIB_CLIENT
```

Editez le fichier Makefile pour renseigner les bons noms et les bonnes options du compilateur, en accord avec le code que vous devez utiliser. Puis compilez la libairie client en entrant make.

Pour compiler le client IP utilisable depuis un code Python, il faut disposer de cython, numpy et faire :

```
> make palmip4py.so
```

Des exemples de couplage d'un code Python avec la librairie IP sont donnés dans le repertoire :

PALM_MP/TEST/z_iplib/d_python/

17.8 Rappel des points de ce chapitre

Dans ce chapitre vous avez vu comment connecter un code "externe" dans une application PALM. Par externe on entend que ce code ne peut pas être encapsulé dans un exécutable (unité ou block) lancé par le coupleur pour définir l'algorithme de couplage. Le code peut par exemple être une boite noire comme un code commercial, à condition qu'il autorise l'appel de routines utilisateur sous forme de librairies dynamiques. Nous avons montré la différence entre un code séquentiel et un code parallèle. Notons que cette manière de fonctionner impose certaines contraintes sur le code lui-même comme l'accès à une version du code compatible avec la version MPI-2 du reste de l'application.

Si le Code n'est pas compatible avec la version MPI de PALM, ou s'il est nécessaire de tourner en mode MPI-1, la solution consiste à utiliser la connexion par IP et unité miroir. Notons que cette solution autorise également des couplages sur un parc hétérogène de machines pour peu qu'elles soient visibles sur le même réseau.

18 Ecriture d’unités PALM en langage Python

Initialement basée sur l'outil SWIG décrit dans le chapitre suivant, l'interface OpenPALM pour Python a été réécrite depuis la version 4.1.4 d'OpenPALM afin de répondre au besoin de codes de calcul parallèles dont le programme principal est un assemblage de modules Python. En effet la solution basée sur SWIG, ne permet les couplages qu'en mode client/serveur MPI et oblige donc l'utilisateur à utiliser la version MPI2 d'OpenPALM.

La solution proposée ici, qui fonctionne aussi bien en mode MPI1 et MPI2 d'OpenPALM, s'appuie sur les outils NUMPY [1] pour la manipulation de tableaux, MPI4PI [2] pour l'interface MPI (outils les plus couramment utilisés pour les codes de calcul basés sur une interface python), et enfin CYTHON [3] pour construire l'interface PALM. Il est donc indispensable de vérifier que ces outils sont installés sur la machine cible, le cas échéant on peut se procurer ces outils open source et les installer.

[1] http://numpy.scipy.org/
[2] http://mpi4py.scipy.org/
[3] http://www.cython.org/

Les fichiers d'interface (interface_palm.pyx pour PALM et interface_pcw.pyx pour CWIPI) écrits en CYTHON sont fournis dans le répertoire PrePALM_MP/TEMPLATE. Ils sont automatiquement recopiés dans le répertoire utilisateur lorsque ce dernier construit les fichiers de service de l'application PALM. Le fichier Makefile de l'application PALM permet de construire le module palm.so pour le code python.

Il est nécessaire d'ajouter les chemins vers python, cython et les librairies dans le fichier Make.include, par exemple :

```
PYTHON = python
CYTHON = cython
PYTHON_INCLUDE=/chemin/include/python2.7
MPI4PY_INCLUDE=/chemin/python2.7/site-packages/mpi4py/include
NUMPY_INCLUDE=/chemin/python2.7/site-packages/numpy/core/include/
```

On peut facilement déterminer ces chemins avec des commandes python:

```
$python -c 'from distutils import sysconfig; print( sysconfig.get_python_inc() )'
$python -c 'import mpi4py; print( mpi4py.get_include() )'
$python -c 'import numpy; print( numpy.get_include() )'
```

18.1 Unité Python

Pour créer une unité Python, il faut renseigner la carte d'identité pour que PrePALM puisse reconnaitre l'unité Python:

```
#PALM_UNIT -name test_send\
#          -functions {python test_send}\
#          -object_files {}\
#          -comment {exemple python put}\
#          -help {No help available}
```

Après chargement de la carte d'identité, l'unité peut être insérée dans une branche à l'aide de PrePALM tout comme une unité C ou FORTRAN.

Les objets OpenPALM sont définis de la même manière que dans les unités C et FORTRAN:

```
#PALM_SPACE -name mat2d\
#           -shape (:,:)\
#           -element_size PL_DOUBLE_PRECISION\
#           -comment {matdbl}
#
#PALM_OBJECT -name dynmat\
#            -intent OUT\
#            -space mat2d\
#            -comment {test}
```

PrePALM génère automatiquement le Makefile qui permet de compiler l'interface Cython en librairie dynamique qui sera chargée par le script Python. Après exécution de *make*, on trouve le fichier palm.so dans le répertoire courant.

Il faut ensuite inclure cette librairie dynamique (générée par Cython) dans le code de l'application Python ainsi que numpy qui permet de gérer le tableau de data.

```
import palm
import numpy as np
```

Pour accéder aux constantes de PrePALM, il faut importer le module palm_user_param.py:
Il y a deux variantes:

- sous un namespace séparé:
  ```
  import palm_user_param as pu
  ```
  on a alors accès aux constantes à travers le namespace *pu*:
  ```
  print pu.const1
  local_var = pu.const2 + pu.const3
  ```
- dans le namespace principal:
  ```
  from palm_user_param import *
  ```
  ce qui rend les constantes directement accessibles dans le code:
  ```
  print const1
  local_var = const2 + const3
  ```

On va généralement opter pour la deuxième variante, sous réserve de faire un choix judicieux de noms de constantes et noms de variables permettant d'éviter des conflits de nommage.

Le corps de l'application est mis dans une procédure portant le nom défini dans l'attribut -functions de la carte d'identité:

```
def test_send():
```

18.2 Interface orientée objet pour Python

Dans Python, le non-typage des variables laisse peu de liberté pour les arguments dans l'appel à des fonctions. Les objets sont passés par référence, alors que les variables de type élémentaire (int, float) sont passées par valeur.
L'interface Python de OpenPALM consiste donc en classes pour simplifier l'utilisation d'OpenPALM dans Python avec le passage de paramètres de sortie à travers les attributs de la classe.

D'abord, il faut créer un Objet Palm qui sera utilisé pour les échanges. On peut tout de suite lui fournir les paramètres pour initialiser ses attributs. Les attributs manquants sont initialisés à leur valeur par défaut.

```
po = palm.PalmObject(object = "dynmat", space = "mat2d",
                          rank = 2, shape = [dim1,dim2])
```

Ici, on omet délibérément les attributs time et tag, puisque leurs valeurs par défaut PL_NO_TIME, PL_NO_TAG conviennent parfaitement pour cet exemple.
A tout moment, on peut changer les attributs de l'objet iniliasisés lors de la création:

```
po.object = "dynmat2"
po.time = 3
```

Dans l'exemple, on utilise un espace dynamique ici, il faut donc d'abord communiquer la forme du tableau de données au moteur de Palm. Tous les attributs (rank et shape) ont déjà été initialisés à la création de l'objet PALM, on peut donc appeler la fonction directement:

```
po.space_set_shape()
```

Ensuite, on génère un tableau numpy de type double des entiers de 0 à *dim1*dim2* et on peut l'envoyer directement puisque l'objet po contient déjà tous les attributs nécessaires pour *put*:

```
matrix = np.arange(dim1*dim2, dtype = np.float64)
po.put(matrix)
```

18.3 Communication dynamique à travers OpenPALM

Mettons alors cette unité d'envoi test_send.py dans une branche PrePALM, suivie d'une unité test_receive.py qui récupère les données.

Dans cet exemple, nous avons besoin de 2 constantes dans PrePALM:
dim1: entier, valeur = 7
dim2: entier, valeur = 5

Le module test_receive.py doit récupérer les dimensions de l'espace dynamique et recevoir les data. Il utilise aussi un objet PalmObject, mais fait appel, cette fois-ci, à l'espace 'NULL':

```
#PALM_UNIT -name test_receive\
#          -functions {python test_receive}\
#          -object_files {}\
#          -comment {exemple python get}\
#          -help {No help available}
#
#PALM_OBJECT -name mat_in\
#            -intent IN\
#            -space NULL\
#            -comment {test}
```

La classe PalmObject permet d'enchainer directement les déterminations du nom, du rang et du shape de l'espace, puisque les méthodes utilisent un attribut *dynspace* dédié aux espaces dynamiques, alors que l'attribut *space* garde la valeur 'NULL'.

```
po = palm.PalmObject(object = "mat_in", space = "NULL")
po.object_get_spacename()
po.space_get_rank()
po.space_get_shape()
```

On crée un tableau numpy vide de dimensions données par get_shape pour recevoir les données échangées et le tour est joué:

```
matr = np.empty(po.shape, dtype = np.double)
po.get(matr)
```

18.4 Codes parallèles : Get MPI communicator

Dans le cas d'une unité Python parallèle basée sur MPI, l'interface Python s'appuie sur MPI4PY. Dans ce cas les différents processus doivent partager un même communicateur MPI. OpenPALM fournit ce communicateur grâce à la fonction get_mycomm.

L'object communicateur MPI est utilisable après import de mpi4py:

```
import mpi4py.MPI as MPI

Mycomm=MPI.Comm()
palm.get_mycomm(Mycomm)
```

Les commandes MPI peuvent maintenant être appelées sur ce communicateur MPI de Palm:

```
rank = Mycomm.Get_rank()
size = Mycomm.Get_size()
```

18.5 Fonction d'aide Python

Une fois compilé, le module palm comporte une aide d'utilisation intégrée qui est accessible depuis le shell interactif de Python:

```
>import palm
>help(palm)
```

Cette commande affiche la liste de toutes les fonctions OpenPALM et de leur utilisation dans Python. Cette meme aide peut être obtenue depuis la ligne de commande via:

```
pydoc palm
```

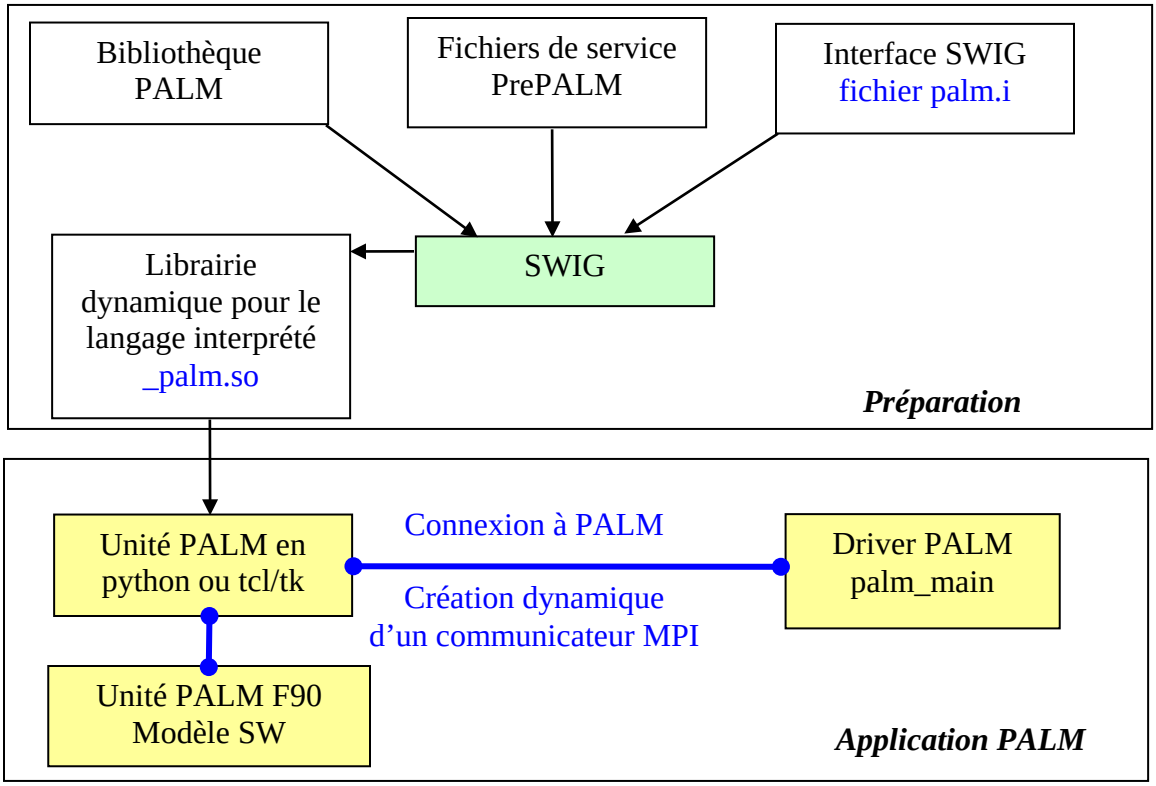
19 Ecriture d'unités PALM en langages interprétés comme perl ou tcl/tk

19.1 Généralités

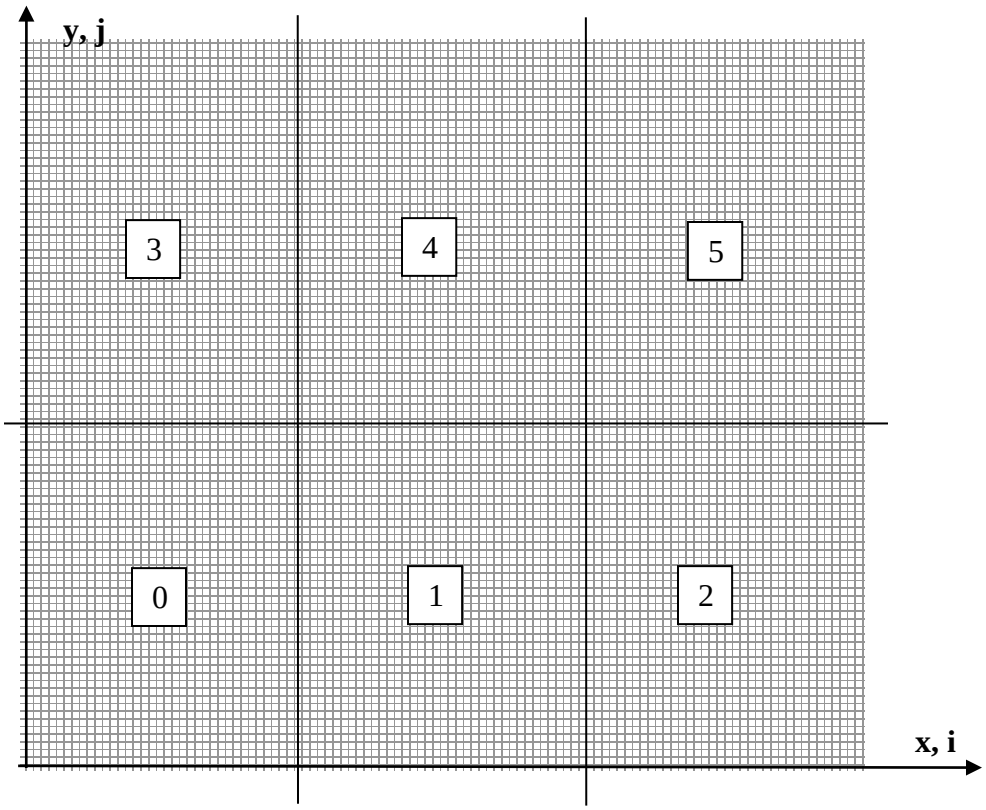
La plupart des langages interprétés tels que TCL/TK, PERL, Octave, autorisent le chargement et l'exécution de code compilés sous forme de librairie dynamiques (fichier .so sous Linux ou .ddl sous Windows). En exploitant cette possibilité, il est possible d'écrire des unités PALM dans différents langages interprétés. L'idée est de proposer les mêmes fonctionnalités PALM dans ces langages pour envoyer et recevoir des données de mémoire à mémoire via les primitives PALM_Put/Get et toute l'API PALM. Ceci présente par exemple un intérêt pour faire des près ou post traitement parallèles ou pour piloter un code de calcul à partir d'une interface graphique. Le mécanisme repose sur la connexion dynamique (par les fonctionnalités MPI-2) du script à PALM (voir chapitre précédent) et par l'utilisation de l'outil d'interfaçage SWIG (que l'on peut se procurer gratuitement sur le site suivant : www.swig.org). La connaissance approfondie de SWIG n'est pas nécessaire car le fichier d'interfaçage (fichier palm.i) est fournit avec PALM. Il est par contre indispensable d'installer cet outil sur votre machine pour pouvoir écrire des unités PALM en langages tcl/tk, etc..

Comme exemple d'utilisation, à partir du même code parallèle « PALMé », le modèle SW, nous allons construire trois applications en écrivant une unité PALM en langage python, la même unité PALM en langage Perl et une unité PALM en langage tcl/tk. Le principe est strictement identique pour ces trois langages et le fichier d'interface de SWIG **palm.i** est le même.

Les programmes source des deux application et le fichier palm.i sont disponibles dans les répertoires chapter_17/unit_python chapter_17/unit_perl et chapter_17/unit_tcl.



Le modèle SW (pour Shallow-Water) que nous allons coupler simule un bassin d'eau « peu profonde ». C'est un modèle explicite, basé sur les équations de Saint-Venant, intégrant dans le temps une hauteur d'eau (H) et les composantes U et V du vecteur vitesse des courants. Les champs sont discrétisés sur une grille structurée (i,j), régulière, modélisant un bassin rectangulaire. La taille du domaine, le nombre de mailles, la durée de la simulation et différents paramètres du modèle sont déterminés par des constantes de PrePALM. Le parallélisme est codé en découpant le modèle sur les deux axes x et y, paramètres eux-mêmes déterminés dans les constantes de PrePALM. Le fonction de distribution est codée en fonctions de ces paramètres, elle est générique quelque soit le nombre de mailles et le nombre de découpes selon les deux axes.



Exemple de décomposition du domaine sur 6 processeurs
(ip_nbproc_x = 3 et ip_nbproc_y = 2 dans les constantes de PrePALM)

La carte d'identité de l'unité SW est la suivante :

```
!================== sw ========================================================
!PALM_UNIT  -name Model\
!           -functions {F90 m}\
!           -object_files {sw.o}\
!           -parallel mpi \
!           -minproc 1\
!           -maxproc 8000\
!           -comment {SW MODEL}
!
!======== espaces ============================================================
!
!PALM_SPACE -name h -shape (ip_i+1,ip_j+1) -element_size PL_DOUBLE_PRECISION -
comment {h variable in model state field}
!
!======== distributeurs ======================================================
!
!PALM_DISTRIBUTOR -name h_distrib\
!                 -type custom\
!                 -shape (ip_i+1,ip_j+1)\
!                 -nbproc ip_nbproc\
!                 -function h_distrib\
!                 -object_files {swparal.o}\
!                 -comment {distributeur de h}
!
!======== objets =============================================================
!
!
```

```
!PALM_OBJECT -name putflag  -intent IN  -localisation SINGLE_ON_FIRST_PROC -space
one_integer\
!            -comment {flag of fields to be sent.}
!
!PALM_OBJECT -name time -intent OUT  -localisation SINGLE_ON_FIRST_PROC -space
one_integer\
!            -comment {Time iteration}
!
!PALM_OBJECT -name tend -intent OUT  -localisation SINGLE_ON_FIRST_PROC -space
one_integer\
!            -comment {final time step}
!
!PALM_OBJECT -name hn  -intent OUT -distributor h_distrib  -localisation
DISTRIBUTED_ON_ALL_PROCS\
!            -space h  -time ON -comment {h variable in model forecast}

!============================================================================
```

Comme objet en entrée, l'unité SW attend à chaque pas de temps de sa boucle interne le drapeau *putflag* lui indiquant si elle doit faire ou non un PALM_Put du champ *hn* contenant la valeur des hauteurs d'eau. En sortie, au début du programme, l'unité SW envoie le nombre total d'itérations qu'il va effectuer. Ensuite, à l'intérieur se sa boucle temporelle interne, il envoie le pas de temps courant : objet *time*

Du point de vue PALM l'unité SW fonctionne comme ceci :

- au départ du programme, envoi du nombre d'itérations total de la simulation,
- à chaque itération temporelle :
  - envoi du numéro de l'itération courante,
  - réception d'un drapeau lui indiquant une action à effectuer,
  - si le drapeau est 1, envoi du champs hn à cette itération.

19.2 Unité PALM écrite en perl

L'unité perl que nous proposons va simplement demander quelques valeurs temporelles du champ H du modèle SW pour les imprimer dans les fichiers de log de PALM. L'application proposée se trouve dans le répertoire **chapter_17/unit_perl**.

Après avoir reçu le nombre total d'itération de l'unité SW, l'unité écrite en perl effectue une boucle interne sur les mêmes itérations que le SW, dans cette boucle, elle reçoit du SW à chaque pas de temps l'itération courante, si celle-ci correspond à une valeur où le champ H doit être sorti, elle envoie un drapeau à l'unité SW pour l'informer de faire un PALM_Put du champ H qu'elle reçoit ensuite avec un PALM_Get pour l'afficher. Pour plus de généricité, l'espace du champ à recevoir dans l'unité perl est déclaré à NULL, on utilise donc les primitives adéquates dans l'unité unit_perl.pl pour récupérer les tailles des espaces à manipuler.

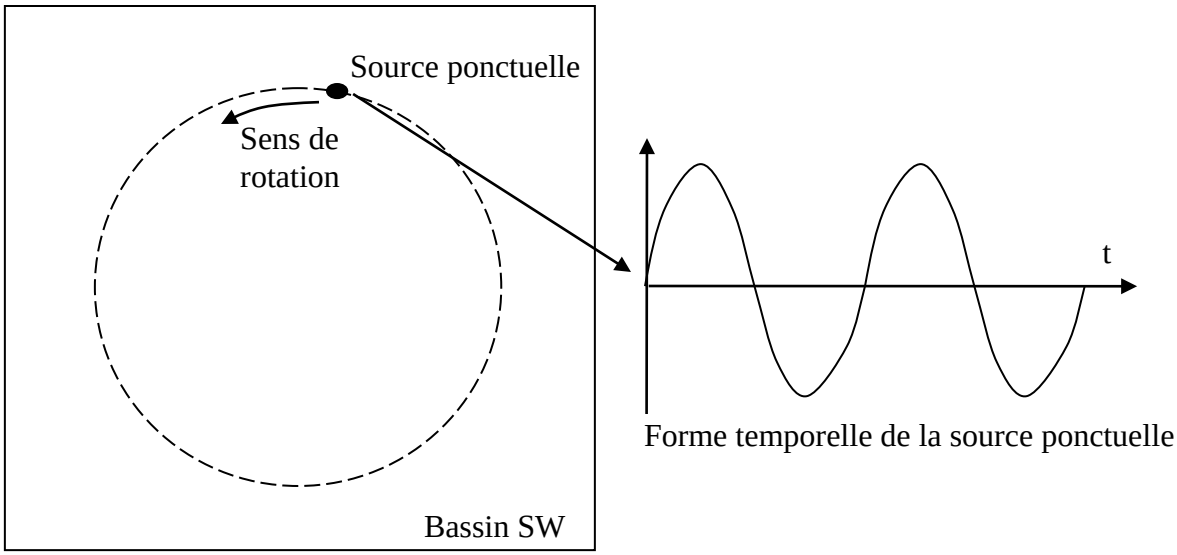
Unit_perl.pl :

```
#PALM_UNIT -name unit_perl\
#          -functions {SH run_unit_perl.sh&}\
#          -comment {unit_perl}
#PALM_OBJECT -name putflag\
#            -space one_integer\
#            -intent OUT\
#            -comment test
#PALM_OBJECT -name time\
#            -space one_integer\
#            -intent IN\
#            -comment modeltime
#PALM_OBJECT -name tend\
#            -space one_integer\
#            -intent IN\
#            -comment {end time}
#PALM_OBJECT -name hfield\
#            -space NULL\
#            -intent IN\
#            -comment hfield

use palm;


$err = palm::PALM_Mpi_init();
$err = palm::PALM_Connect();

$time_p = palm::new_int($palm::PL_NO_TIME);
$tag_p  = palm::new_int($palm::PL_NO_TAG);
$tend_p = palm::new_int(0);
$tcur_p = palm::new_int(0);
$flag_p = palm::new_int(0);
$ila_shape = palm::new_array_int(2);

$err = palm::PALM_Get("one_integer", "tend", $time_p, $tag_p, $tend_p);

$tend = palm::get_int($tend_p,0);

$time_sortie = 422;

$t = 0;
while ($t < $tend) {
    $err = palm::PALM_Get("one_integer", "time", $time_p, $tag_p, $tcur_p);
    $t= palm::get_int($tcur_p,0);
    print "$t \n";
    palm::PALM_Print("iteration : $t");
    if ($t == $time_sortie) {
        palm::set_int($flag_p,0,1);
    } else {
        palm::set_int($flag_p,0,0);
    }

    $err = palm::PALM_Put("one_integer", "putflag", $time_p, $tag_p, $flag_p);

    if ($t == $time_sortie) {

        $space_name = "  ";
        $err = palm::PALM_Object_get_spacename( "hfield", $space_name);
        $err = palm::PALM_Space_get_shape($space_name,2,$ila_shape);
        $nx =  palm::get_int($ila_shape,0);
        $ny =  palm::get_int($ila_shape,1);
        $nxy = $nx*$ny;
        $dla_field = palm::new_array_double($nxy);
        $err = palm::PALM_Get("NULL","hfield",$tcur_p,$tag_p,$dla_field);
```

```
        palm::PALM_Print("tableau hfield recu au temps $t");
        for ($i = 0; $i <= $nxy; $i++) {
          $field = palm::get_double($dla_field,$i);
            palm::PALM_Print("Field( $i) = $field");
        }
        palm::delete_double($dla_field);
    }
}

palm::delete_int($time_p);
palm::delete_int($tag_p);
palm::delete_int($tend_p);
palm::delete_int($tcur_p);
palm::delete_int($flag_p);

$err = palm::PALM_Disconnect();
$err = palm::PALM_Mpi_finalize();
print "\n====> unit_perl end\n";
```

19.3 Unité PALM écrite en TCL/TK

Cet exemple ressemble beaucoup aux précédents, le modèle à coupler est strictement le même, mais pour faire différent, nous allons exploiter les possibilités orientées « interface utilisateur du langage tcl/tk » pour coder une petite interface graphique permettant de faire une représentation sommaire du champ 2D H calculé dans le modèle SW. Nous irons même un peu plus loin que de faire de la visualisation parallèle interactive de données en modifiant un paramètre du SW. Le bassin de forme carré est excité par une source se déplaçant à vitesse constante le long d'un cercle contenu dans le bassin. Cette Source modifie ponctuellement la hauteur d'eau en forçant le champ H selon une fonction sinusoïdale qui varie au cours du temps (voir dans le code source pour plus de détails). Pour illustrer une interaction forte entre le modèle et l'interface graphique, l'utilisateur pourra inverser le sens de rotation de la source par un simple clic sur un bouton de l'interface graphique et visualiser l'effet sur le champ obtenu.



La construction de l'application PALM est identique à celle décrite pour le langage Python.

Voici maintenant le code de l'unité unit_tcl.tcl, petite interface graphique, écrite en tcl/tk avec l'utilisation de primitives PALM dans le langage tcl :

```
#!/bin/sh
# the next line restarts using wish\
exec wish "$0" "$@"

#PALM_UNIT -name unit_tcl\
#          -functions {SH unit_tcl.tcl&}\
#          -comment {unit_tcl}
#PALM_OBJECT -name putflag\
#            -space one_integer\
#            -intent OUT\
```

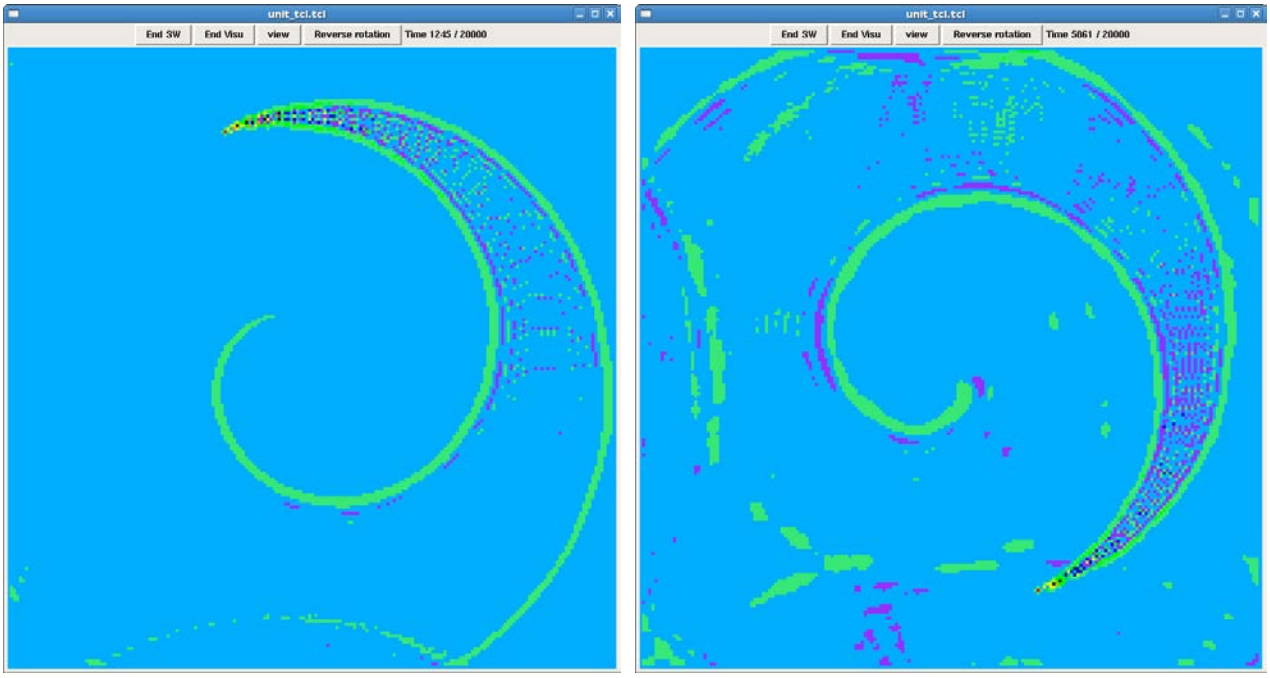
```
#            -comment test
#PALM_OBJECT -name time\
#            -space one_integer\
#            -intent IN\
#            -comment modeltime
#PALM_OBJECT -name tend\
#            -space one_integer\
#            -intent IN\
#            -comment {end time}
#PALM_OBJECT -name hfield\
#            -space NULL\
#            -intent IN\
#            -comment hfield

# Cette unité PALM est écrite en TCL/TK langage de script associé
# à une librairie graphique pour le développement d'interface.
#
# On illustre ici la possibilité de faire des appels PALM
# dans des langages interprétés en utilisant l'interfaceur SWIG
#
# Notons que les variables tcl ne sont pas typées alors que
# les primitives PALM travaillent elles sur des types C comme
# des entiers des réels ou des doubles.
# Pour cela on est amené à utiliser des fonctions qui permettent
# de déclarer de tels types, d'autres fonctions permettent
# de passer d'un type c à une variable tcl représentant les données
#

# Définition de quelques procédures, le programme principal est
# définit à la fin du script.

# Chargement dynamique de la librairie PALM
# initialisation du contexte MPI et connexion
# de l'unité à PALM
proc init {} {
    # palm.so est la librairie dynamique à charger en mémoire
    # pour les unités tcl
    # elles est construite par le fichier Makefile make_swig
    # remarque : cette librairie est spécifique à chaque
    # application PALM, car elle fait intervenir les
    # fonctions de service écrites par PrePALM
    load ./palm.so palm
    # Il est nécessaire d'initaliser MPI, plutôt que de créer
    # un module MPI interfacé avec tcl, PALM propose une primitive
    # qui fait un appel à MPI_Init
    PALM_Mpi_init
    # Une fois le contexte MPI_2 initialisé, on appelle
    # la primitive PALM_Connect qui crée dynamiquement
    # les communicateur entre le driver de PALM et l'unité
    set err [PALM_Connect]
}

# déconnexion de l'unité PALM
# arrêt de MPI et sortie du programme
proc finalize {} {
    set err [PALM_Disconnect]
    PALM_Mpi_finalize
    exit
}

# le pilotage du modèle SW est basé sur l'envoi à chaque itération
# d'un signal (un entier) selon la valeur de ce signal
# le SW exécutera des actions différentes
# flag (p_flag en variable C) vaut
```

```
#       0 -> aucune action
#       1 -> signal d'envoi du tableau 2D de champs H
#     999 -> arrêt du programme SW
#     444 -> inversion du sens de rotation de la source circulaire
# Les trois routines suivantes positionnent ce signal
# sur réception d'un événement TK associé aux boutons de l'interface
# graphique

# positionnement du flag à 1 -> signal de sortie de H pour le SW
proc need_field_signal {} {
    set_int $::p_flag 0 1
}

# positionnement du flag à 999 -> signal de fin pour le SW
proc exit_sw_signal {} {
    set_int $::p_flag 0 999
}

# positionnement du flag à 444 -> signal d'inversion du sens de rotation
proc revert_rotation_signal {} {
    set_int $::p_flag 0 444
}

# procédure graphique d'affichage du champs h

proc draw {curtime} {
    global larg haut
    PALM_Print "Plot de la hauteur d'eau demande au temps : $curtime"
    set time [new_int $::PL_NO_TIME]
    set tag  [new_int $::PL_NO_TAG]
    # réception de la taille effective du champ h
    set cl_space "==================="
    set err [PALM_Object_get_spacename hfield $cl_space]
    set il_shape [new_array_int 2]
    set err [PALM_Space_get_shape $cl_space 2 $il_shape]
    set nx [get_int $il_shape 0]
    set ny [get_int $il_shape 1]
    set nxy [expr $nx*$ny]
    #puts "taille du tableau=====>$nxy"
    #reception du champ de hauteur d'eau
    set hfields [new_array_double $nxy]
    set err [PALM_Get NULL hfield $::timemodel $tag $hfields]


    # partie purement graphique
    set nbniv 11
    set color(0) black   ; set color(1) #0000a2 ; set color(2) blue
    set color(3) #9630fe ; set color(4) #00aefe ; set color(5) #38e876
    set color(6) green   ; set color(7) #d2fe00 ; set color(8) #fe9600
    set color(9) #fe6e00 ; set color(10) red

    set fmin 496.
    set fmax 504.
    for {set j 0} {$j < $ny} {incr j} {
      for {set i 0} {$i < $nx} {incr i} {
          set i1 [expr $i/$nx.*$larg]
          set i2 [expr ($i+1)/$nx.*$larg]
          set j1 [expr (1-$j/$ny.)*$haut]
          set j2 [expr (1.-($j+1)/$ny.)*$haut]
          set ind [expr $i+$j*$ny]
          set f [get_double $hfields $ind]
          set i_f [expr int(($f-$fmin)/($fmax-$fmin)*($nbniv-1) -.49999)]
          if {$i_f < 0} {set i_f 0}
          if {$i_f > 10} {set i_f 10}
```

```
          .c  create  rectangle  $i1  $j1  $i2  $j2  -fill  $color($i_f)  -outline
$color($i_f)
          incr ind
        }
    }
    # deallocation
    delete_double $hfields
    delete_int $time
    delete_int $tag
    delete_int $il_shape
}

# --------------------
# programme principal
# interface graphique
# --------------------

# une frame pour contenir les boutons
set f .f; frame $f;pack $f -side top

# un canvas graphique pour dessiner le champs H
global larg haut ;# taille de la fenetre graphique
set larg 800
set haut 800
set c .c ; canvas $c  -background white -width $larg  -height $haut
pack $c -side top

# quelques boutons pour l'utilisateur
button .f.quit -text "End Visu" -command finalize
button .f.view -text view -command need_field_signal
button .f.endsw -text "End SW" -command "exit_sw_signal"
button .f.revert -text "Reverse rotation" -command "revert_rotation_signal"
label .f.time -text "no time"
pack .f.endsw .f.quit .f.view .f.revert .f.time  -side left

# Initialisation du contexte PALM
init

#get du temps final du modèle
set time [new_int $::PL_NO_TIME]
set tag  [new_int $::PL_NO_TAG]
set p_tend [new_int 1]
set p_flag [new_int 0]
set err  [PALM_Get "one_integer" "tend" $time $tag $p_tend]
set tend [get_int $p_tend 0]

set curtime 0

# boucle sur les itérations du modèle SW

while {$curtime < $tend} {
    # Réponse aux évènements utilisateurs
    update
    # réception du temps courant envoyé par le modèle
    set timemodel [new_int 0]
    # réception de l'itération courante du SW
    set err [PALM_Get "one_integer" "time" $time $tag $timemodel]
    set curtime [get_int $timemodel 0]
    # mise à jour de l'itération courante dans l'interface graphique
    .f.time configure -text "Time $curtime / $tend"
    # envoi du signal au SW
    set err  [PALM_Put "one_integer" "putflag" $time $tag $p_flag]
    set flag [get_int $p_flag 0]
    if {$flag == 1}   {draw $curtime}
```

```
    if {$flag == 999} {finalize}
    # remise à 0 du flag si celui-ci a été modifié
    # pour les itérations suivantes
    set_int $p_flag 0 0
}

# Fin du contexte PALM
finalize
```

Une fois l'application compilée, on peut lancer l'application PALM qui lance la petite interface graphique utilisateur :



unit_tcl.tcl : Visualisation interactive de la hauteur d'eau avant et après inversion du sens de rotation de la source.

Cette interface n'a aucune prétention, elle est là pour montrer à quel point il est facile d'interfacer des scripts ou des logiciel graphiques pour en faire des unités PALM. Le mécanisme peut facilement être copié pour interfacer des logiciels graphiques plus ambitieux. En résumé, on utilise dans cet exemple plusieurs fonctionnalités de PALM :

- la redistribution de données parallèles, quel que soit le nombre de proc du modèle SW, le programme de visualisation représente la totalité du domaine même si le champ initial est calculé sur plusieurs processus,
- l'héritage de la taille des espaces : l'interface graphique est capable de visualiser toute taille de grille,
- la connexion dynamique de processus à une application PALM.

L'échange de données se fait à chaque fois de mémoire à mémoire via MPI ce qui garanti l'efficacité de l'application.

19.4 Rappel des points de ce chapitre

Ce chapitre traite le cas particulier d'unités PALM écrites dans des langages interprétés en utilisant la fonctionnalité de connexion dynamique des programmes au coupleur PALM comme expliqué dans le chapitre précédent. Des exemples de couplage avec un modèle écrit en Fortran 90 sont donnés pour les langages Perl et Tcl/Tk.

20 Installation du logiciel PALM

20.1 Généralités

Le logiciel PALM est distribué avec les programmes source sous licence LGPL. Après avoir récupéré l'archive, a première chose à effectuer est de la décompresser avec la commande :

> tar -xvfz distrib.tgz

Cette commande exécutée, deux répertoires sont créés dans le répertoire courant : PrePALM_MP et PALM_MP. Le premier répertoire contient l'interface graphique PrePALM, le second la bibliothèque PALM. L'interface graphique peut être installée en local sur le poste de travail de l'utilisateur, la bibliothèque PALM doit être installée sur les différents calculateurs où sont déployées les applications PALM.

20.2 Installation de l'interface graphique PrePALM

20.2.1 Pré requis

L'interface graphique PrePALM est développée en langages Tcl/Tk et C, il est donc nécessaire de disposer d'une installation de ces deux produits sur votre station de travail. PrePALM est validé avec les version 8.3 et ultérieures de Tcl/Tk. PrePALM peut fonctionner sous Windows moyennant l'installation d'un émulateur linux comme Cygwin

Le langage C est utilisé pour la partie STEPLANG, il est donc nécessaire de compiler ce composant sur certaines machines uniquement car une version compilée de STEPLAN est fournie pour les plateformes linux x86_64.
Certaines bibliothèques de la boite à outil d'algèbre peuvent être nécessaires pour vos applications. Les bibliothèques blas lapack et scalapak ainsi que les minimiseurs, même s'ils sont interfacés par PrePALM, ne sont pas fournis avec la distribution de PALM. Il convient donc à l'utilisateur de se procurer ces bibliothèques et de les installer si elles sont nécessaires aux applications. La bibliothèque d'interpolation géophysique basée sur le coupleur OASIS, elle-même basé sur la bibliothèque domaine public scrip est fournie dans la distribution de d'OpenPALM dans le répertoire PrePALM_MP/ALGEBRA/Interpolation/Geophysic/DSCRIP_lib.

20.2.2 Définition de la commande prepalm

L'interface graphique étant développée dans un langage interprété il n'y a pas de compilation du programme. Chaque utilisateur doit cependant définir une variable d'environnement contenant le chemin du logiciel et un alias pointant sur la commande prepalm.

Si le répertoire absolu ou PrePALM est installé se nomme chemin_de_PrePALM, chaque utilisateur, selon le shell utilisé, doit définir :

csh, tcsh :
setenv PREPALMMPDIR chemin_de_PrePALM
alias prepalm '$PREPALMMPDIR/prepalm_MP.tcl \!* &'

sh, bash :
export PREPALMMPDIR=chemin_de_PrePALM
alias prepalm=' $PREPALMMPDIR/prepalm_MP.tcl'

Ces commandes sont à ajouter au fichier de login utilisateur .cshrc .tcshrc .bashrc … selon le cas.

La variable d'environnement PREPALMEDITOR peut également être positionnée à l'éditeur de votre choix, par défaut c'est l'éditeur vi qui est utilisé dans les menus de PrePALM comme l'édition des programmes sources des unités. Par exemple si vous désirez utiliser emacs à la place de vi, rentrer cette commandes dans votre fichier de configuration :

csh, tcsh :
setenv PREPALMEDITOR emacs

sh, bash :
export PREPALMEDITOR=emacs

20.2.3 Installation de STEPLANG

Steplang est le langage de commande utilisé pour la gestion explicite des objets dans le buffer. Pour compiler ce langage, positionnez vous dans le répertoire STEPLANG de PrePALM :
> cd PrePALM_MP/STEPLANG/

Sous Linux entrez les commandes :
> make clean
> make
Le résultat de la commande make doit conduire à la création du fichier steplang-i386. Sous d'autre systèmes unix il peut être nécessaire de modifier le fichier Makefile présent dans ce répertoire pour changer le nom du compilateur et de l'exécutable. Inspirez vous simplement des lignes en commentaire pour trouver les bonnes options.

20.2.4 Installation de la bibliothèque OASIS le cas échéant

La bibliothèque OASIS permet d'interpoler des champs géophysiques dans un repère à coordonnées sphériques. Cette bibliothèque traite les grilles structurées ou non les plus répandues dans les codes de calcul utilisés pour la modélisation du climat. Pour plus d'information sur les méthodes d'interpolation et l'utilisation de cette bibliothèque, il convient de se référer à la documentation du coupleur OASIS du CERFACS.
Les programmes sources de cette bibliothèque sont présents dans la distribution de PALM dans le répertoire : PrePALM_MP/ALGEBRA/Interpolation/Geophysic/DSCRIP_lib, dans ce répertoire la commande make permet de générer la bibliothèque. Selon les machines il peut être nécessaire de modifier le fichier Makefile.

20.3 Installation de la bibliothèque PALM

20.3.1 Pré requis

La bibliothèque PALM contient les éléments nécessaires à la constitution du driver de PALM (palm_main) et des entités utilisateurs (unités et blocks). Cette bibliothèque doit être compilée sur le calculateur où doit tourner les applications PALM. L'installation du logiciel est basée sur l'outil standard de configuration automatique « autoconf ». PALM est développé en langages C et FORTRAN 90.

Pour installer PALM il est nécessaire de disposer :

-d'un compilateur FORTRAN 90 et d'un compilateur C compatible avec ce dernier,

-d'une bibliothèque MPI qui implémente la morme MPI_2 (sauf cas particulier de la version PALM dégradée basée sur MPI_1). La bibliothèque MPI doit être compilée avec les mêmes compilateurs que la bibliothèque PALM.

Optionnellement, selon les fonctionnalités de PALM utilisées, il sera ou non nécessaire de disposer :

-des bibliothèques de calcul scientifique BLAS et LAPACK (ou équivalents constructeur) pour les boîtes d'algèbre monoprocessus,

-des bibliothèques PBLAS et SCALAPACK pour les boîtes d'algèbre parallèles,

-de la bibliothèque NETCDF pour la gestion des fichiers de ce type,

-des programmes source des minimiseurs dont l'interface est disponible dans la boite à outils d'algèbre.

L'installation de PALM ne requière pas d'être root sur la machine d'installation.

20.3.2 Installation

Dans le répertoire PALM_MP, l'installation de PALM se fait en trois commandes :

```
> ./configure [OPTION]... [VAR=VALUE]...
> make
> make install
```

Le plus délicat est de trouver les bonnes options à donner à la commande configure, ces options dépendent des compilateurs, de l'adressage, de la distribution MPI utilisée et enfin de la version de PALM à déployer (mode monoproc, MPI_2 ou MPI_1).

L'ensemble des options du configure de PALM est donné par la commande ./configure --help, le résultat de cette aide en ligne est le suivant :

```
Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print `checking...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for `--cache-file=config.cache'
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or `..']

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [NONE]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [PREFIX]

By default, `make install' will install all the files in
`NONE/bin', `NONE/lib' etc.  You can specify
an installation prefix other than `NONE' using `--prefix',
for instance `--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR           user executables [EPREFIX/bin]
  --sbindir=DIR          system admin executables [EPREFIX/sbin]
  --libexecdir=DIR       program executables [EPREFIX/libexec]
  --datadir=DIR          read-only architecture-independent data [PREFIX/share]
  --sysconfdir=DIR       read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR   modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR    modifiable single-machine data [PREFIX/var]
  --libdir=DIR           object code libraries [EPREFIX/lib]
  --includedir=DIR       C header files [PREFIX/include]
  --oldincludedir=DIR    C header files for non-gcc [/usr/include]
  --infodir=DIR          info documentation [PREFIX/info]
  --mandir=DIR           man documentation [PREFIX/man]

System types:
  --build=BUILD     configure for building on BUILD [guessed]
  --host=HOST       cross-compile to build programs to run on HOST [BUILD]

Optional Features:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-64bits         Use 64 bits addressing (default on sgi and fujitsu)
  --enable-promote-real   Promote REAL fortran data type to DOUBLE PRECISION
  --enable-blasopti       Use BLAS optimization (default on scalar computers)
```

```
  --enable-mpi_softwait    Use non CPU hogging mpi_wait (default on sgi, sun,
nec, linux)

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --without-mpi           Use Monoprocessing without MPI
  --with-mpich=MPICH_ROOT mpich for MPI (default=no)
  --with-lam=LAMMPI_ROOT  lam for MPI (default=no)
  --with-openmpi=OPENMPI_ROOT  OpenMPI for MPI (default=no)
  --with-mpi_path=path    Path of the MPI implementation
  --with-F90=F90          F90 compiler
  --with-CC=CC            C compiler
  --with-fopt=OPT         Option for Fortran Compiler
  --with-copt=OPT         Options for C compiler
  --with-debug=EXTRA_FLAGS    enable debugging (default debug flag  is -g)
  --with-fortran_underscore   Underscore at end of fortran functions
  --with-fortran_main=MAIN    internal name of main FORTRAN routine
                              (default value depends on system type)
  --with-roundtrip-delay=roundtrip-delay        *100 MPI_Iprobes (default~100)
  --with-mpi_comm_free=mpi_comm
  --with-leak_mem_ctl      To detect memory leak
  --with-shared_lib      Compile shared libraries
  --with-mpi1mode          using mpi1 mode (no spawn)
  --with-mpi2win           using mpi2 windows

Some influential environment variables:
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you have
              headers in a nonstandard directory <include dir>
  CPP         C preprocessor

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.
```

Seules les options en caractères gras de couleur bleu sont exploitables pour les utilisateurs, les autres options sont réservées à l'équipe de développement de PALM.
Dans tous les cas il convient de choisir les compilateurs FORTRAN 90 et C, la plupart des compilateurs fonctionnent avec PALM, notons par exemple :

-gcc et gfortran, suite GNU,
-gcc et g95,
-pgcc et pgf90 suite PGI,
-suite de compilateurs intel,
-suite de compilateurs pathscale,
-xlc et xlf90 sur IBM,
-sxmpif90 et sxmpicc sur NEC.

Les versions testées les plus intensivement (car utilisées pour le développement de PALM) sont (pgcc, pgf90), (gcc, gfortran) et (icc, ifort).
Le plus important au niveau du choix du compilateur est de respecter la cohérence avec ces dernier pour toute l'application PALM en respectant cet ordre :

- compilation de la bibliothèque MPI,
- compilation de la bibliothèque PALM,

- compilation des unités PALM (dans le cas de bibliothèques),
- compilation des applications PALM.

Une fois les compilateurs sélectionnés, il faut choisir la distribution de MPI compatible avec PALM, les versions du domaine public compatibles et testées avec PALM en mode MPI2 sont les suivantes :

- lammpi version 6.x.x et suivantes	--with-lam=chemin où lammpi est installé
- openmpi version 1.2.7 et suivantes	--with-openmpi=chemin où openmpi est installé
- mpich2 version 1.0.7 et suivantes	--with-mpich=chemin où mpich2 est installé

En mode MPI_1 n'importe quelle bibliothèque implémentant correctement cette norme est suffisante.

20.3.3 Exemple d'installation sur une station de travail Linux

Supposons qu'on ait une station de travail linux adressage 64 bits avec les compilateurs GNU installés (commande gfortran et gcc) mpich installé avec ces compilateurs dans le répertoire /usr/local/mpich/, qu'on veuille installer les librairies dynamiques de PALM, qu'on ne compile pas en fortran avec la promotion des réels simple précision en double précision, la commande à lancer est la suivante :

./configure --with-mpich=/usr/local/mpich/ --with-shared_lib --with-F90=mpif90 --with-CC=mpicc

20.4 Rappel des points de ce chapitre

Ce chapitre résume la manière d'installer PALM et son interface graphique. Toutes les plateformes ont leurs propres spécificités. L'équipe de développement de PALM vous encourage à faire remonter les difficultés que vous pouvez avoir rencontré et les solutions que vous avez trouvées pour en faire bénéficier d'autres utilisateurs.

21 Fonctions utiles, plus ou moins spécifiques

21.1 Choix par défaut et liste de choix pour les types simple en entrée des unités

Nous avons vu dans les précédents chapitres que les objets décrits en entrée des unités dans les cartes d'identité (donc reliés aux PALM-Get dans le code) n'étaient pas forcement des champs de couplage mais souvent des drapeaux ou autres informations pour gérer un paramètre ou un mode de fonctionnement de l'unité. Ces données peuvent être renseignées par des communications, ou plus simplement « en dur » avec un simple click droit de la souris.

Si cela à un sens, il est possible, au niveau de la carte d'identité, de définir une valeur par défaut pour ces objets. Dans ce cas, lors de l'utilisation de l'unité dans PrePALM, le plot correspondant à cet objet sera directement mis à une valeur en dur correspondant à cette valeur par défaut. Prenons l'exemple de l'unité producteur de la session 5, et ajoutons une entrée qui correspond à un mode de fonctionnement de l'unité. Si nous voulons que la valeur défaut de cet objet soit 1, on ajoutera simplement les lignes suivantes dans la carte d'identité :

```
!PALM_OBJECT -name run_mode\
!            -space one_integer\
!            -intent IN\
!            -default 1\
!            -comment {mode de fonctionnement de l'unite}
!
```

Objet « run_mode » dont on fera un PALM_Get à l'endroit opportun du code.

Pour aller un peu plus loin, il est également possible de définir une liste prédéfinie de valeurs que peut prendre cet objet. Imaginons que nous ayons 3, et seulement 3, choix possibles pour ce mode de fonctionnement, dans ce cas nous pouvons ajouter l'attribut –closedlist pour définir ces choix. L'utilisateur de l'unité sera ainsi obligé de choisir uniquement dans cette liste, ceci permet de réduire le risque d'erreurs de saisie dans l'interface graphique.

```
!PALM_OBJECT -name run_mode\
!            -space one_integer\
!            -intent IN\
!            -closedlist { {1 : mode normal} {2 : option 1} {3 : option 2}}\
!            -default 1\
!            -comment {mode de fonctionnement de l'unite}
```

Chaque élément de la liste doit être définit à l'intérieur d'accolades. Pour une option, la valeur prise par l'objet sera le premier mot de l'expression, le reste de l'expression est considéré comme un commentaire pour informer l'utilisateur.

21.2 Gestion du champ time, association de dates

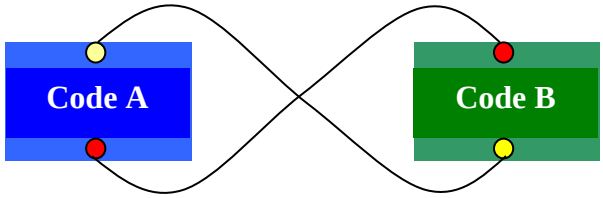
21.2.1 Généralités

Le champ « time » des primitives PALM_Put/Get est géré comme un entier pour la simplicité et la généricité de l'interface PALM. Dans les sessions consacrées aux communications, nous avons insisté sur l'importance de cet attribut car il permet :

- de gérer avec souplesse les boucles internes et externes des programmes à coupler,
- d'effectuer l'interpolation temporelle des données le cas échéant,
- de définir des fréquences de couplage différentes dans différentes applications couplées avec une instrumentation unique du code simplement en jouant sur l'attribut « time list » de la communication.

En pratique, pour définir des couplages entre différents codes ayant des pas de temps différents, plusieurs stratégies peuvent être adoptées selon la nature du couplage à mettre en œuvre.

21.2.2 Le couplage fort

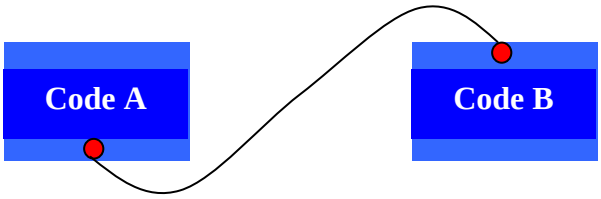


Couplage fort : *Les codes A et B tournent en parallèle, la synchronisation entre les deux codes se fait par les communications*

En général, pour du couplage fort, i.e. échange bilatéral entre les codes dans un processus itératif, l'utilisateur est quasiment obligé de trouver une base de temps commune à tous les modèles en se calant sur des multiples du modèle ayant le plus petit pas de temps. Notons que dans ce cas les programmes doivent tourner nécessairement en parallèle donc sur deux branches différentes. L'échange étant bilatéral et les PALM_Get étant bloquants, la synchronisation entre les codes se fait implicitement lors de ces échanges car au PALM_Get chaque code attend le résultat de l'autre pour continuer. En toute logique l'utilisation du champ time ne semble donc pas nécessaire dans ce cas de figure. Elle est tout de même conseillée pour des raisons de généricité de l'instrumentation. Soit l'utilisateur gère lui-même la fréquence des échanges avec le mécanisme de son choix (fichier de configuration, constantes de PrePALM ou plots PALM pour demander un temps de début, un temps de fin et une fréquence des PALM_Put/get, dans ce cas on peut envoyer les champs avec l'attribut PL_NO_TIME. Soit il utilise le champ « time » et la fréquence des échanges peut être gérée avec l'attribut « time list » de la communication qui agit comme un filtre, dans ce cas, les échanges ne se font qu'aux instants décrits dans les communications.
Bien qu'il soit possible d'associer des temps différents à l'envoi et à la réception, il est vivement conseillé, pour la lisibilité de l'application, de trouver une base de temps commune entre les

différents codes participant au couplage, en général on se base sur le modèle ayant le plus petit pas de temps.

21.2.3 Le couplage faible ou forçage



Couplage faible : *Les données transitent uniquement du code A vers le code B, si l'échange doit avoir lieu à plusieurs instants, il est important d'associer un champ « time » pour différencier les différentes instances du champ ou de prévoir un autre mécanisme de synchronisation.*

Le cas du couplage faible, ou forçage est un peu différent. Dans ce cas les transferts de données ne se font que d'un code vers l'autre, le code qui reçoit peut éventuellement tourner à la suite de celui qui envoie. Si les objets sont tous envoyées avec l'attribut PL_NO_TIME (même si les codes tournent en parallèle sur deux branches différentes) il y a risque d'écraser les données avant qu'elles n'aient été traitées par le code devant les recevoir car le PALM_Put n'est pas bloquant. En envoyant les champs avec un attribut « time » différent de PL_NO_TIME ce risque est éliminé car il n'y aura pas de confusion possible entre les différentes instances temporelles de l'objet. Il faut tout de même faire attention à ne pas saturer la mailbuff de PALM, ce qui peut se produire si le code qui envoie les données est plus rapide que celui qui les reçoit. Attention : un couplage qui va fonctionner sur une configuration donnée sur une machine donnée peut ne pas fonctionner dans une autre configuration ou sur une autre machine uniquement par ce que l'application n'est pas synchronisée. Il faut donc toujours garder à l'esprit que le couplage se fait dans un contexte parallèle et qu'il peut être judicieux d'ajouter des mécanismes de synchronisation entre les deux codes, soit par l'ajout de communications spécifiques (se reporter au le chapitre sur l'interpolation temporelle à ce sujet) soit par l'emploi de Steps bloquants.

21.2.4 La conversion d'entier en date ou réciproquement

Principalement pour les applications concernant les sciences de la terre, où l'utilisateur relie ses sorties de modèle à des dates effectives, ce dernier peut vouloir gérer (plutôt qu'un entier) des dates réelles rapportées à une échelle de temps (un calendrier), une unité de temps (seconde, minute, heure ou jour), et une origine (à quelle date correspond l'entier 0). PALM offre des primitives pour gérer cela, l'échelle, la base et l'origine sont précisées une fois pour toute pour tous les modèles dans l'interface graphique PrePALM. Dans le menu « Date conversion » l'utilisateur doit tout d'abord choisir un calendrier parmi Standard (le calendrier officiel, utilisé le plus couramment), NoLeap (sans jours fériés, années de 365 jours), Julian (calendrier Julien qui traite les années bissextiles différemment du calendrier standard) et 360 (calendrier ne comportant que 360 jours

utilisé par certains modèles climatiques). Il doit ensuite choisir sa base de temps qui peut aller de la seconde au jour en passant par l'heure et la minute. Par exemple si l'heure est choisie, deux entiers successifs utilisés dans les primitives PALM_Put/Get correspondront à deux heures successives. Une fois ces données entrées dans PrePALM, l'utilisateur a accès à un petit utilitaire interactif permettant de convertir les dates en entier et réciproquement (dans le menu Date conversion). Cet utilitaire permet aussi de calculer des plages de valeur pour les entiers selon l'échelle choisie, il peut par exemple retourner tous les entiers qui correspondent au premier jour du mois sur une période définie. Dans les codes, la primitive PALM_Time_convert (voir son prototype dans la liste des primitives PALM) permet de passer d'un entier à une date et réciproquement.

21.3 Gestion dynamique de la verbosité

La gestion assez fine de la verbosité de PALM a été mise en place pour déboguer les applications. Selon le niveau de verbosité choisi, les fichiers de sortie de PALM (palmdriver.log et branche_XXX.log) peuvent être plus ou moins volumineux. Un niveau de verbosité important peut s'avérer nécessaire sur des grosses applications comportant de nombreuses communication mais peut conduire à l'écriture d'énormes quantités de données si le problème apparait relativement tard dans la simulation. Pour contourner ce problème PALM offre des primitives qui permettent d'agir dynamiquement dans les codes sur le niveau de verbosité. Une fois le problème cerné dans telle ou telle unité, éventuellement à tel ou tel endroit, il est possible d'augmenter ou de diminuer le niveau de verbosité afin d'économiser de la place sur disque. Cette action se fait par un simple appel à la primitive **PALM_Verblevel_set**(int categorie, int *niveau). Ces primitives qui servent à la mise au point du couplage ont pour vocation à disparaitre lorsque les problèmes sont réglés car l'écriture de ces messages sur disque peut ralentir significativement l'application.

21.4 Contrôle du contenu des objets : palm_debug

Le fichier palm_debug.f90 ou palm_debug.c selon de choix du langage par l'utilisateur, permet de « contrôler » les objets PALM à l'envoi ou à la réception de ceux-ci. Pour activer cette fonctionnalité il suffit de choisir PL_DEBUG_ON_SEND, PL_DEBUG_ON_RECV ou PL_DEBUG_ON_BOTH pour l'attribut « Palm debug status » de la boite de dialogue de la communication. Si ce champ est activé, le sous programme palm_debug est appelé par PALM respectivement à l'envoi, à la réception ou au deux de l'objet.

Le même sous programme étant appelé pour tous les objets, il est nécessaire de caractériser les traitements selon le nom de l'objet et/ou de l'espace que l'on veut contrôler. Les commentaires dans le fichier palm_debug.f90/c sont suffisamment explicites pour trouver facilement la manière de mettre en œuvre cette fonctionnalité. Notons qu'il est plus facile et plus judicieux d'intervenir dans ce fichier plutôt que directement dans le code de calcul. Cette fonctionnalité n'est pas réservée au débogage « informatique » de l'application, il peut être mis en œuvre comme un verrou pour contrôler le couplage et arrêter l'application (par un appel à PALM_Abort) si quelque chose d'anormal au sens physique se passe. Par exemple pour un couplage Océan/atmosphère dans lequel on envoie des températures de surface de l'océan, il est aisé de contrôler dans ce fichier que le

tableau contenant ces températures ne contient pas des valeurs erronées dues à un couplage qui divergerait.

21.5 Affichage du contenu des objets : Primitive PALM_Dump

La primitive PALM_Dump, généralement utilisée dans le sous programme palm_debug (également accessible dans les unités PALM) permet d'afficher aisément certaines informations comme le minimum, le maximum ou la somme du tableau contenant l'objet PALM. Cette primitive est décrite dans l'annexe consacrée aux primitives PALM de ce manuel.

21.6 Rappel des points de ce chapitre

Dans ce chapitre nous avons vu quelques fonctions avancées de PALM, pas forcement nécessaires pour vos couplages mais qui peuvent rendre certains services. En particulier nous avons parlé de la façon de décrire des valeurs par défaut pour les objets ou comment en limiter le choix pour l'utilisateur, comment associer une date à un entier et réciproquement et finalement comment jouer sur la verbosité ou vérifier le contenu de objets pour rendre vos applications plus robustes.

22 Mode fichier de commande pour PrePALM

Il est possible de générer des fichiers PrePALM sans passer par l'interface graphique. Pour cela l'utilisateur doit construire un fichier avec l'extension .pml (PrePALM Meta Language). A titre de documentation des exemples sont fournis dans les répertoires corrigés des différentes session du répertoire training. On donne ci après le fichier .pml permettant de réaliser la session 8 du tutorial.

Pour tester positionnez vous dans le bon répertoire du tutorial:
> cd training/session_8

Lancez prepalm depuis cet endroit sur le fichier .pml contenant les commandes PrePALM:
> prepalm corrige/session_8.pml

Le fichier session_8.ppl est généré (attention de ne pas écraser le vôtre le cas échéant) ainsi que les fichiers de service. vous n'avez plus qu'à compiler l'application et la lancer :
> make
> mpirun -np 1 ./palm_main

```
# OpenPALM version > 4.1.7
# Exemple de fichier de commande PrePALM
# résolution de la session 2 du tutorial

# choix du mode MPI (1 ou 2)
MPI_MODE 2

# definition des constantes
CONSTANT IP_SIZE     PL_INTEGER 100000
CONSTANT debut_prod  PL_INTEGER 0
CONSTANT fin_prod    PL_INTEGER 1000
CONSTANT step_prod   PL_INTEGER 10
CONSTANT debut_print PL_INTEGER 1
CONSTANT fin_print   PL_INTEGER 1000
CONSTANT step_print  PL_INTEGER 7

# chargement des cartes d'identité
LOAD producteur.f90 vecteur_print.f90

##############################
# définition de la branche b1
##############################
BRANCH b1 IP_START_ON
# lancement d'une instance de producteur
LAUNCH producteur producteur 1 100

##############################
# définition de la branche b2
##############################
BRANCH b2 IP_START_ON
# déclaration des variables pour cette branche
VAR ib_do PL_INTEGER
VAR nouv_put PL_INTEGER
VAR dernier_put PL_INTEGER -1
```

```
# définition d'un block
BLOCK
  # définition d'une boucle do
  DO ib_do debut_print fin_print step_print
     # région fortran
     F90 nouv_put = (ib_do/10+1)*10
     # définition d'une condition
     IF nouv_put.ne.dernier_put
        PALM_PUT one_integer b2_put_1 PL_NO_TIME PL_NO_TAG nouv_put
        F90 dernier_put = nouv_put
     ENDIF
     # lancement d'une instance de vecteur_print
     LAUNCH vecteur_print vecteur_print 1 100
  ENDDO
ENDBLOCK

# communications en dur (plot rabatu)
SET_GET min_time.producteur debut_prod
SET_GET max_time.producteur fin_prod
SET_GET freq_time.producteur step_prod
SET_GET ref_time.vecteur_print ib_do

# définition des communications
COMM b2_put_1.b2 synchro.producteur PL_NO_TIME PL_NO_TAG
COMM vecteur.producteur vecteur.BUFFER debut_prod:fin_prod:step_prod PL_NO_TAG
COMM   vecteur.BUFFER   vecteur.vecteur_print   debut_print:fin_print:step_print
PL_NO_TAG  PL_NO_DEBUG  PL_NO_TRACK   {PL_INS  0  0}   PL_GET_LINEAR   IDENTITY
IDENTITY AUTOMATIC MEMORY PL_NO_OPTIM

#définition des instructions Steplang
STEPLANG
/* destruction des instances temporelles du vecteur qui ne servent plus */
for $time in [15:1000:7] {
     on {
      com("BUFFER",   0,   "vecteur",   $time,   PL_NO_TAG,"vecteur_print",   0,
"vecteur", $time, PL_NO_TAG);
        }
     do {
        $time1 = ($time / 10 - 1 ) * 10 ;
      delete("vecteur", $time1, PL_NO_TAG);
        }
}
ENDSTEPLANG

# trace de l'execution pour le rejeu graphique
TRACE_EXECUTION
TRACE_COMMUNICATION
TRACE_BUFFER

# nom du fichier .ppl à générer
SAVE session_8.ppl

# génération des fichiers de service
MAKE_PALM_FILES
```

23 Glossaire Palm

Action : instruction du langage steplang à exécuter sur un événement ; comme détruire un objet ou lui donner un attribut prêt.

Algèbre : Unité prédéfinie.

Application : Une application PALM est un assemblage d'unités élémentaires dans le coupleur qui permet la définition de l'algorithme, le lancement des différentes tâches et l'échange de données entre les composants.

Association de processus : reliées aux communications parallèles et aux distributeurs, les associations de processus permettent de déterminer (dans le cas ou ce n'est pas trivial pour le coupleur) de quels processus à quels autres les données doivent transiter.

Barrière : associées aux steps, les barrières permettent de synchroniser les applications parallèles.

Block : assemblage de plusieurs unités Palm et éventuellement de structures de contrôles dans un exécutable unique.

Branche : les branches de calcul de PALM permettent de définir l'algorithme de couplage. Les branches tournent (ou non) en parallèle, peuvent démarrer ou non au début de la simulation. Elles contiennent des déclarations de variables et des structures de contrôle.

Buffer : zone mémoire localisée sur le processus du driver (palm_main), allouée dynamiquement au fur et à mesure des besoins, permettant à l'utilisateur de gérer explicitement les objets à échanger, comme par exemple pour les interpoler ou les composer.

Carte d'identité : fichier de définition des unités pour l'interface graphique Prepalm.

Catégorie : dans Prepalm, une des rubriques à sélectionner pour afficher les attributs de chaque élément.

Code de calcul : programme écrit dans un langage de programmation évolué comme Fortran, C ou C++

Communication : mise en relation dans l'interface graphique Prepalm des plots correspondant au primitives PALM_Put/Get du code utilisateur.

Communication parallèle : communication faisant intervenir au moins un objet distribué.

Constante : il est possible de définir des constantes dans Prepalm, ces constantes peuvent servir pour la définition de l'algorithme dans les codes de branche et dans les différents menus. Ces constantes peuvent également servir dans les unités en incluant l'un des fichiers (langage dépendant) dans le code source.

Coupler : action pour faire fonctionner deux ou plusieurs programmes dans une même application. PALM permet le lancement simultané (ou non) de ces programmes et permet l'échange de données entre ces composants.

Démon : processus s'exécutant en tache de fond permettant de réaliser un service. LAM fonctionne avec le démon lamboot, certaines version de Mpich avec mpd.

Distributeur : liste d'entiers ou sous-programme permettant de retourner, dans un format compréhensible par PALM, la manière dont un tableau est décomposé processus par processus.

Driver : Le driver de PALM (palm_main) est le programme principal de l'application. Son rôle est de séquencer l'exécution des branches en lançant dynamiquement les différentes unités ou blocks et de répondre aux requêtes des exécutables pour les communications.

Esclave mémoire : les esclaves mémoires sont des processus permettant d'étendre la mémoire du driver de PALM si celle-ci n'est pas suffisante pour l'application.

Espace : un espace PALM, caractérisé par son nom, permet de définir une quantité d'information, c'est ce qui permet de décrire informatiquement les objets, comme leur type et les dimensions du tableau.

Espace dynamique : un espace est dynamique si sa taille est déterminée à l'exécution de l'application palm. Cette taille peut également varier au cours de la simulation.

Evénement : associés aux steps, les événements sont les points de passage où une action dans le Buffer de palm peut être réalisée. Les communications sont des évènements au même titre que les steps définis par l'utilisateur.

Fichiers de service : les fichiers de service sont des sous-programmes (FORTRAN et C) généré par l'interface graphique PrePALM.

Fonction de distribution : voir distributeur

Granularité : qualificatif utilisé en programmation parallèle pour rendre compte de la taille (mémoire et/ou temps de calcul) des opérations élémentaires.

Héritage : dans l'interface graphique PrePALM, l'héritage d'espace permet de définir des objets de type indéfinis (NULL), qui prennent la valeur de ceux avec lesquels ont les met en relation.

Librairie : ensemble de sous-programmes compilés, regroupés dans un fichier.

Localisation : les localisations permettent de spécifier quels sont les processus impacté par un distributeur.

Mailbuff : Zone mémoire gérée par le driver de palm et permettant de stocker temporairement les objets non prêts à être reçu.

Makefile : fichier au format make permettant de compiler une application PALM. Ce fichier est automatiquement généré par l'interface graphique.

Modularité : qualificatif qui rend compte des possibilités qu'offre une application à être facilement réorganisée, ou partiellement réutilisé.

MPI : bibliothèque de communication permettant de paralléliser les codes de calcul selon une norme standard.

MPMD : acronyme (Multiple Program Multiple Data) utilisé en programmation parallèle pour dire que l'application parallèle fait tourner en même temps, et dans un même contexte de communication, plusieurs exécutables différents.

Objet : les objets PALM sont les quantités de données gérées par le coupleur pour échanger des informations entre les unités, les objets sont caractérisés par leur nom.

Objet dynamique : objet qui a un espace dynamique

PALM : Projet d'**A**ssimilation par **L**ogiciel multi **M**éthodes, le coupleur dont il est question ici.

Palmer : Action de rendre un programme utilisateur compatible avec le logiciel PALM. Certains préfèrent « palmériser ». Les palmipèdes utilisent palmer.

Paramètres : voir constantes.

Plot : dans PrePALM, les plots sont les petits disques matérialisés en haut (entrée Get) et en bas (sortie Put) représentant les objets que peuvent se transmettre les unités.

Prepalm : interface graphique de PALM

Primitive : sous-programme de la bibliothèque PALM appelable dans les unités utilisateur.

Priorité : attribut des unités (ou des blocks) déterminant leur ordre de lancement. En cas de misère de ressource, l'unité de priorité la plus forte passe avant les autres, en cas d'égalité de priorité l'ordre est indéterminé.

Processeur : composant électronique permettant le traitement de l'information.

Processus : à ne pas confondre avec processeur. Un processus est une instance de programme en cours d'exécution, on peut avoir plusieurs processus sur le même processeur.

Région fortran : dans l'interface graphique Prepalm, les régions fortran permettent d'entrer du code source dans les branches de calcul, la syntaxe à respecter est celle du Fortran 90.

Rejouer : fonctionnalité de l'interface graphique qui permet à l'utilisateur, après l'exécution de l'application, de visualiser séquentiellement l'ordre dans lequel les différents composants ont tourné.

Ressources : nombre de processus ou taille mémoire disponible pour une application. PALM gère en partie ces ressources.

Script : fichier comprenant des commandes systèmes.

Shape : taille de chaque dimension d'un tableau multidimensionnel.

SPMD : acronyme (Single Program Multiple Data) utilisé en programmation parallèle pour dire que l'application parallèle consiste à faire tourner en même temps, et dans un même contexte de communication, un seul exécutable dont les données (tableaux, variables) ont leur propre instance.

Step : évènement explicitement décrit par l'utilisateur dans Prepalm.

Steplang : Langage de programmation, interprété par PALM pour déclencher des actions sur les objets du BUFFER.

Structure de contrôle : les structures de contrôles (boucles et conditions) sont utilisables dans les branches de Prepalm.

Suivi en temps réel : fonctionnalité de l'interface graphique permettant de savoir quelles sont les unités, branches ou block de l'application en cours d'exécution.

Tag : attribut des primitives PALM_Put/Get permettant, s'il est différent de PALM_NO_TAG, de différencier deux objets de même nature.

Time : attribut des primitives PALM_Put/Get permettant, s'il est différent de PALM_NO_TIME, de différencier deux instances temporelles d'un objet de même nature.

Type dérivé : type informatique composé à partir de types élémentaires.

Unité : composant informatique, écrit en langage évolué, adapté pour tourner avec le coupleur PALM.

Unité parallèle : composant informatique tournant sur plusieurs processus.

Unité prédéfinie : Unité PALM, proposée par l'interface graphique, directement utilisable dans une application.

Verbosité : niveau plus ou moins élevé d'écriture de messages PALM dans les fichiers de sortie.

Sous-objet : fonctionnalité de PALM permettant de manipuler une partie seulement d'un objet.

Valeur en dur : fonctionnalité de l'interface graphique permettant d'imposer une expression fortran valide au retour d'une primitive PALM_Get. (click droit de la souris sur le plot de l'objets)

24 Liste des primitives PALM

24.1 Formulation C et FORTRAN

Selon le langage de programmation :

Forme d'appel C/C++ :

```
int il_err ;
il_err = PALM_Exemple(arg1,arg2,…) ;
```

Forme d'appel FORTRAN :

```
integer il_err
CALL PALM_Exemple (arg1, arg2, …, il_err)
```

Remarque : pour le C tous les arguments (IN et OUT) sont passés par adresse (comme en Fortran).

Dans la suite, les primitives PALM sont données selon leur forme C, pour le FORTRAN il convient d'ajouter le code d'erreur comme dernier argument.

Primitive pour terminer une application PALM

int PALM_Abort()

Primitives pour les communications

int PALM_Put(char *space, char *obj, int *time, int *tag, void *data)

int PALM_Get(char *space, char *obj, int *time, int *tag, void *data)

int PALM_Query_get(char *space, char *obj, int *time, int *tag)

int PALM_Query_put(char *space, char *obj, int *time, int *tag)

Avec :

space, obj : chaines de caractères de longueur PL_LNAME
time : entier ou PL_NO_TIME
tag : entier ou PL_NO_TAG
data : tableau contenant les données

Primitive pour convertir des entiers en date ou réciproquement

int PALM_Time_convert(int * dir,int *jour,int *mois,int *an,int *heure,int *min,int *sec,int *time)

Avec :

dir : PL_TIME_INT2DATE ou PL_TIME_DATE2INT

Primitives pour de gestion de la verbosité

int PALM_Verblevel_get(int categorie, int *niveau)

int PALM_Verblevel_set(int categorie, int *niveau)

Avec :

categorie : PL_VERB_BRANCH, PL_VERB_UNIT, PL_VERB_COMM,
PL_VERB_STEP ou PL_VERB_GENERIC
niveau : 0, 10, 20, 30, 40 ou 50

Primitives d'écriture dans les fichiers PALM

Ecriture de messages :
void PALM_Write
Uniquement pour C et C++ : fonction PALM_Write de même format que fprintf pour émuler le write(PL_OUT, … du fortran
Exemple : PALM_Write(PL_OUT, ''message %i'',23) ;
Taille maximum des messages : 1024 caractères.
Primitive pour forcer l'écriture du buffer d'écriture sur disque :
PALM_Flush(PL_OUT) ;

Vérification du contenu des objets :
int PALM_Dump(int op, char *space, char *obj, int time, int tag, void *data)

Avec :

space, obj : chaines de caractères de longueur PL_LNAME
time : entier ou PL_NO_TIME
tag : entier ou PL_NO_TAG
data : tableau contenant les données
op : PL_DUMP_MIN, PL_DUMP_MAX, PL_DUMP_SUM, PL_DUMP_ALL

Primitives pour de gestion des types dérivés

int PALM_Space_get_size(char *space)

int PALM_Pack(void *buffer, char *space, char *item, int *position, void *data)

int PALM_Unpack(void *buffer, char *space, char *item, int *position, void *data)

Avec :

space, item : chaines de caractères de longueur PL_LNAME
buffer : tableau contenant les données regroupée ou à regrouper
data : tableau contenant les données par item

Primitives pour les espaces dynamiques

int PALM_ Space_set_shape(char* space, int rank, int *shape)
int PALM_Space_get_rank (char *space, int* rank)
int PALM_ Space_get_shape(char *space, int rank, int *shape)
int PALM_Object_get_spacename(char *obj, char *space)

Avec :

space, obj : chaines de caractères de longueur PL_LNAME
rank : nombre de dimension
shape : tableau d'entier de taille rank

Primitives pour la connexion des codes de calcul indépendants

int PALM_ Connect()
int PALM_Disconnect()

Autres primitives

int PALM_Get_myname(char *name)

Retourne le nom de l'unité donné dans PrePALM, avec des noms différents si plusieurs instances de la même unité.

La variable name doit être déclarée comme suit :

C: char name[PL_LNAME]:
Fortran : character (len=PL_LNAME) :: name

int PALM_Barrier(MPI_Comm comm)

Permet une synchronization sur le communicateur MPI comm. Basé sur MPI_Ibarrier puis MPI_Test suivi d'une mise en veille, l'attente ne consomme pas de CPU. Les variables d'environnement PALMSPINWAITS et PALMNANOSLEEP ont une influence sur le comportement de cette primitive.

PALMSPINWAITS : nombre de tentatives MPI_Test avant la mise en veille.
PALMNANOSLEEP : temps d'attente passive en microsecondes.

int PALM_Unit_set_progress(float *progress)

Avec:

progress : valeur dans l'intervale [0:1] permettant d'informer PALM du degrès de progression de l'unité. Par exemple (numéro_de_l_itération)/(nombre_d_itération). Cette primitive permet d'afficher graphiquement dans PrePALM la progression de l'unité lors du suivi en temps réel de l'éxécution l'application PALM (rider).

24.2 Formulation Python

Python ne permet pas le passage par référence d'arguments de type fondamental (int, float), ce qui empêche de formuler les fonctions de la même façon que dans C et FORTRAN.
Dans python, une classe est utilisée pour regrouper les attributs d'entrée-sortie. Les méthodes Palm sont appelées sur cet objet. Pour bien comprendre l'utilisation, voir l'exemple dans le chapitre dédié à Python.
Il y a une classe PalmObject pour les communications, une classe TimeConvert pour les conversions de dates et quelques primitives sans classe appartenant au module palm

Classe PalmObject:

Attributs publics:

object
: chaine de longueur PL_LNAME contenant le nom de l'objet

space
: chaine de longueur PL_LNAME contenant le nom de l'espace pour les opérations d'envoi/réception
pour les espaces dynamiques, il doit avoir la valeur "NULL"
utilisé uniquement par les fonctions space_set_shape, put et get

dynspace
: chaine de longueur PL_LNAME contenant le nom de l'espace dans le cas des espaces dynamiques
utilisé uniquement par les fonctions object_get_spacename, space_get_rank et space_get_shape

rank
: entier décrivant la dimension du tableau data

shape
: numpy array d'entier de dimension rank qui décrit la forme des données.
La conversion vers un objet numpy est automatique, si un entier, un tuple, une liste ou une array est fourni.
Le type retourné est toujours un numpy array

time
: entier décrivant l'identification temporelle de l'échange

tag
: entier décrivant l'identification supplémentaire tag de l'échange

Méthodes publiques:

Créateur:
PalmObject(object='', space='', rank=0, shape=None, time=PL_NO_TIME, tag=PL_NO_TAG)
Crée un PalmObject, dont les attributs peuvent être initialisés explicitement en argument. Tous les attributs non fournis prennent leurs valeurs par défaut.
arguments:
object, space: chaines de longueur inférieure à PL_LNAME
rank: entier
shape: entier ou tuple, liste ou tableau d'entiers
time, tag: entiers, par défaut PL_NO_TIME/TAG

PalmObject.space_set_shape()
Communique la valeur de l'attribut shape au moteur OpenPALM pour définir la taille d'un espace dynamique.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
space: doit contenir le nom de l'espace dynamique
rank: doit contenir la dimension des données
shape: doit contenir la forme des données

PalmObject.put(ndarray io_object)
Envoie des données via l'objet OpenPALM en cours.
arguments: objet numpy contenant les données dans son buffer
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
object: doit contenir le nom de l'objet OpenPALM
space: doit contenir le nom de l'espace ('NULL' pour un espace dynamique)
time, tag: identification de l'échange

PalmObject.get(ndarray io_object)
Reçoit des données via l'objet OpenPALM en cours.
arguments: objet numpy de taille suffisamment grande dans le buffer duquel sont écrites les données. Pour déterminer la taille requise dans le cas des espace dynamiques, voir les méthodes space_get_rank et space_get_shape
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
object: doit contenir le nom de l'objet OpenPALM
space: doit contenir le nom de l'espace ('NULL' pour un espace dynamique)
time, tag: identification de l'échange

Les méthodes suivantes s'appliquent aux espaces dynamiques, elles utilisent un attribut dédié *dynspace* pour le nom de l'espace, alors que l'attribut *space* garde la valuer 'NULL'

PalmObject.object_get_spacename()
Détermine le nom de l'espace dynamique utilisé par un objet.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
object: doit contenir le nom de l'objet
attributs de sortie:
dynspace: reçoit le nom de l'espace dynamique

PalmObject.space_get_rank()
Détermine le rang d'un espace dynamique.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
dynspace: doit contenir le nom de l'espace dynamique (différent de 'NULL')
attributs de sortie:
rank: reçoit le rang de l'espace dynamique

PalmObject.space_get_shape()
Détermine la forme d'un espace dynamique.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
dynspace: doit contenir le nom de l'espace dynamique (différent de 'NULL')
rank: doit contenir le rang de l'espace dynamique
attributs de sortie:
shape: reçoit la forme de l'espace dynamique

PalmObject.dump(int operation, ndarray data)
Ecrit le contenu d'un objet dans PL_OUT
arguments:
operation: operation à effectuer sur le data, parmi:
PL_DUMP_MIN, PL_DUMP_MAX, PL_DUMP_SUM, PL_DUMP_ALL
les operations peuvent etre sommées pour les combiner
(PL_DUMP_MAX+PL_DUMP_SUM)
data: données à inverstiguer
valeur de retour: status d'erreur *il_err*
attributs d'entrée:
space: doit contenir le nom de l'espace
object: doit contenir le nom de l'espace
time, tag: identification de l'échange
attributs de sortie: pas d'attributs de sortie

Classe TimeConvert:

Attributs publics:
jour, mois, an, heure, min, sec: entiers décrivant la date en forme courante
time: entier qui code la date dans la convention Palm avec les paramètres de PrePALM

Les conversions entre date Palm et date de calendrier se font automatiquement lors de l'accès en lecture aux attributs.

Méthodes publiques:
TimeConvert.convert_time()
Appel manuel de la conversion de date (OpenPALM -> calendrier).
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
TimeConvert.convert_to_time()
Appel manuel de la conversion de date (calendrier -> OpenPALM).
arguments: aucun argument
valeur de retour: status d'erreur *il_err*

Primitives sans classe:

init(char *unit_name)
Initialisation d'une session palm; appelé par les fichiers de service générés par PrePALM. En général, il n'est pas nécessaire d'appeler cette primitive à la main.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
finalize()
Finalisation d'une session palm; appelé par les fichiers de service générés par PrePALM. En général, il n'est pas nécessaire d'appeler cette primitive à la main.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
get_mycomm(Comm application_comm)
Obtention du communicateur MPI utilisé dans Palm pour des unités parallèles.
arguments: MPI Comm objet dans lequel est stocké le communicateur
valeur de retour: status d'erreur *il_err*
abort()
Terminaison d'une application parallèle palm en cours. Toutes les unités parallèles se terminent disgracieusement.
arguments: aucun argument
valeur de retour: status d'erreur *il_err*
freeproc_nb()
Permet d'obtenir le nombre de processeurs disponibles.
Variante plus facile à intégrer dans python que get_freeproc_nb
arguments: aucun argument
valeur de retour: nombre de processeurs

get_freeproc_nb(ndarray nbproc)
arguments: nbproc: tableau numpy que la fontion remplit avec le nombre de processeurs
valeur de retour: status d'erreur *il_err*
write(char *string)
Ecrit un message dans le fichier output de la branche OpenPALM active (PL_OUT)
arguments: chaine à écrire
valeur de retour: status d'erreur *il_err*

space_get_size(char *spacename)
Obtenir la taille d'un espace palm ou d'un espace dérivé pour pouvoir dimensionner des tableaux.
arguments: spacename: nom de l'espace (chaine de longueur PL_LNAME)
valeur de retour: status d'erreur *il_err*
pack(ndarray buffer, char *spacename, char *item, int position, ndarray data)
Stocker une donnée au sein d'une structure de transfert OpanPALM. La structure doit être définie dans les ID_cards de PrePALM
arguments:
buffer: numpy array dans lequel les données seront stockées
spacename: nom de l'espace (chaine de longueur PL_LNAME)
item: nom de l'élément de la structure (chaine de longueur PL_LNAME)
position: Indice de l'élément à stocker dans le cas d'un tableau (basé à 0)
data: numpy array contenant la donnée à stocker dans buffer
valeur de retour: status d'erreur *il_err*
unpack(ndarray buffer, char *spacename, char *item, int position, ndarray data)
Déstocker une donnée d'une structure de transfert OpanPALM. La structure doit être définie dans les ID_cards de PrePALM
arguments:
buffer: numpy array duquel les données seront extraites
spacename: nom de l'espace (chaine de longueur PL_LNAME)
item: nom de l'élément de la structure (chaine de longueur PL_LNAME)
position: Indice de l'élément à extraire dans le cas d'un tableau (basé à 0)
data: numpy array dans lequel la donnée est écrite
valeur de retour: status d'erreur *il_err*

not yet interfaced with Python:
connect, disconnect, query_put, query_get, verblevel_set, verblevel_get, flush

Les fonctions ci-dessous existent pour conserver une interface similaire à C et FORTRAN. Par contre, dans python il est conseilllé d'utiliser les méthodes de la classe PalmObject.
space_set_shape(char *spacename, int rank, ndarray shape)
object_get_spacename(char *objectname, char *spacename)
space_get_rank(char *spacename, ndarray rank)
space_get_shape(char *spacename, int rank, ndarray shape)

put(char *space_name, char *object_name, int time, int tag, ndarray object)
get(char *space_name, char *object_name, int time, int tag, ndarray object)

25 Liste des primitives PCW interface pour la bibliothèque CWIPI

25.1 Formulation C et FORTRAN

Selon le langage de programmation :

Forme d'appel C/C++ :

```
int il_err ;
il_err = PCW_Exemple(arg1,arg2,…) ;
```

Forme d'appel FORTRAN :

```
integer il_err
CALL PCW_Exemple (arg1, arg2, …, il_err)
```

Dans la suite, les primitives PCW sont données selon leur forme C, pour le FORTRAN il convient d'ajouter le code d'erreur comme dernier argument.

Contrôle du couplage CWIPI

int PCW_Init()
Initialisation de la bibliothèque CWIPI et redirection des sorties CWIPI dans les fichiers de log d'OpenPALM. Point de synchronisation de toutes les unités OpenPALM utilisant CWIPI.

int PCW_Init_tuned(int id_flag, <= PL_CWIPI_ON ou PL_CWIPI_OFF
MPI_Comm *id_outcomm) <= communicateur MPI en retour

Idem PCW_Init avec deux arguments supplémentaires, id_flag peut prendre les valeurs PL_CWIPI_ON ou PL_CWIPI_OFF, cet argument permet de préciser les processus du code parallèle participant ou non au couplage. Pour les processus avec id_flag = PL_CWIPI_OFF le reste du programme ne doit pas appeler d'autres primitives PCW. Le communicateur id_outcomm retourné ne comprend que les processus qui appellent avec PL_CWIPI_ON. En fortran id_outcomm est un entier.

int PCW_Finalize()
Termine l'environnement CWIPI. Point de synchronisation.

int PCW_Create_coupling(char *coupling_id, <= identifiant du couplage
int coupling_type, <= type de couplage

int entitiesDim, <= dimension du maillage (1, 2 ou 3)
double tolerance, <= tolérance géométrique pour la localisation
cwipi_mesh_type_t mesh_type, <= type de maillage
cwipi_solver_type_t solver_type, <= type de solveur
int output_frequency, <= fréquence des sorties des champs
char *output_format, <= format des fichiers de visu
char *output_format_option, <= option pour les fichiers de visu
int nb_locations) <= parametre optionnel, seulement si mesh_type = CWIPI_CYCLIC_MESH, donne le nombre de localisations à conserver en mémoire

Création d'un environnement de couplage (objet de couplage) Avec:

- coupling_type

en C/C++:

CWIPI_COUPLING_SEQUENTIAL
CWIPI_COUPLING_PARALLEL_WITH_PARTITIONING
CWIPI_COUPLING_PARALLEL_WITHOUT_PARTITIONNING

en FORTRAN:

CWIPI_CPL_SEQUENTIAL
CWIPI_CPL_PARALLEL_WITH_PART
CWIPI_CPL_PARALLEL_WITHOUT_PART

- mesh_type

CWIPI_STATIC_MESH
CWIPI_CYCLIC_MESH
CWIPI_MOBILE_MESH (inopérant)

- solver_type

CWIPI_SOLVER_CELL_CENTER
CWIPI_SOLVER_CELL_VERTEX

- output format

"EnSight Gold"
"MED_fichier"
"CGNS"

- output option

"text" : ASCII file
"binary" : output binary files, default
"big_endian" : force binary files to big endian
"discard_polygons" : do not output polygons or related value
"discard_polyhedra" : do not output polyhedra or related value
"divide_polygons" : tesselate polygons with triangle
"divide_polyhedra" : tesselate polyhedra with tetrahedra and pyramids, adding a vertex near each polyhedron's center

int PCW_Delete_coupling(char *coupling_id)
Destruction de l'objet de couplage

int PCW_Define_mesh(char *coupling_id, <= identifiant du couplage (in)
int n_vertex, <= nombre de nœuds du maillage

int n_element <= nombre d'éléments
double *coordinates <= coordonnées des nœuds (x1, y1, z1, x2, y2, z2 ...)
int *connectivity_index <= définition de la connectivité (taille n_elements+1)
int *connectivity <= définition des éléments à partir des indices des noeuds (taille connectivity_index[n_element+1])

Définit le maillage du couplage, la connectivité doit être triée de telle sorte que les éléments apparaissent dans cet ordre :
pour des éléments 1d : segments
pour des éléments 2d : triangles, quadrangles, polygones
pour les éléments 3d : tétraèdres, pyramides, prismes, hexaèdres

int PCW_Ho_define_mesh(char *coupling_id, <= identifiant du couplage (in)
int n_vertex, <= nombre de nœuds du maillage
int n_element <= nombre d'éléments
int order <= ordre des éléments
double *coordinates <= coordonnées des nœuds (x1, y1, z1, x2, y2, z2 ...)
int *connectivity_index <= définition de la connectivité (taille n_elements+1)
int *connectivity <= définition des éléments à partir des indices des noeuds (taille connectivity_index[n_element+1])

int PCW_Ho_Ordering_from_ijk_set(char *coupling_id, <= identifiant du couplage (in)
int t_elt, <= type d'élément
int n_element <= nombre d'éléments
int n_nodes <= nombre de noeuds
int *ijk_grid <= user ordering to (u, v, w) grid (size = elt_dim*n_nodes)

int PCW_Set_points_to_locate(char *coupling_id, <= identifiant du couplage (in)
int *n_vertex <= nombre de nœuds ou barycentre de maille à localiser
doubles *coordinates) <= coordonnées des point à localiser

Localisation des points à interpoler à un autre endroit que ceux du maillage source (aux nœuds pour du CELL_VERTEX où aux centres des éléments pour du CELL_CENTER)

int PCW_Add_polyhedra(char *coupling_id, <= identifiant du couplage (in)
int id_nb_elem, <= numéro du polyèdre à ajouter (in)
int *ida_face_index, <= Face index (0 to n-1) size : n elements + 1
int *ida_cell_to_face_connectivity, <= Polyhedra => face (1 to n), size : face_index[n_elements]
int *ida_face_connectivity_index, <= Face connectivity index (0 to n-1), size : n_faces+1
int *ida_face_connectivity) <= Face connectivity (1 to n), size : face_connectivity_index[n_faces]

Ajout d'un polyèdre au maillage.

Primitives pour les communications

int PCW_Sendrecv(char *coupling_id, <= identifiant du couplage (in)
char *exchange_name, <= identifiant de l'échange (in)
int stride, <= nombre de champs entrelacés à échanger (in)
int time_step, <= instance temporelle uniquement pour la visu. (in)
double time_value, <= valeur du temps uniquement pour la visu. (in)
char *sending_field_name, <= nom du champ à envoyer (in)
double *sending_field, <= champ envoyé (in)
char *receiving_field_name, <= nom du champ à recevoir (in)
double *receiving_field, <= champ reçu (out)
int *not_located_points) <= nombre de points non localisés (out)

Echange croisé de champs sur les deux maillages, cette primitive synchronise l'application. En langage C/C++, l'échange peut être dans une seule direction, équivalent à PCW_Send ou PCW_Recv en utilisant un pointeur NULL pour le champ reçu ou envoyé (respectivement).

int PCW_Recv(char *coupling_id, <= identifiant du couplage (in)
char *exchange_name, <= identifiant de l'échange (in)
int stride, <= nombre de champs entrelacés à échanger (in)
int time_step, <= instance temporelle uniquement pour la visu. (in)
double time_value, <= valeur du temps uniquement pour la visu. (in)
char *receiving_field_name, <= nom du champ à recevoir (in)
double *receiving_field, <= champ reçu (out)
int *not_located_points) <= nombre de points non localisés (out)

int PCW_Send(char *coupling_id, <= identifiant du couplage (in)
char *exchange_name, <= identifiant de l'échange (in)
int stride, <= nombre de champs entrelacés à échanger (in)
int time_step, <= instance temporelle uniquement pour la visu. (in)
double time_value, <= valeur du temps uniquement pour la visu. (in)
char *sending_field_name, <= nom du champ à envoyer (in)
double *sendingfield, <= champ envoyé (in)
int *not_located_points) <= nombre de points non localisés (out)

int PCW_Irecv(char *coupling_id, <= identifiant du couplage (in)
char *exchange_name, <= identifiant de l'échange (in)
int tag, <= tag MPI de la communication (in)
int stride, <= nombre de champs entrelacés à échanger (in)
int time_step, <= instance temporelle uniquement pour la visu. (in)
double time_value, <= valeur du temps uniquement pour la visu. (in)
char *receiving_field_name, <= nom du champ à recevoir (in)
double *receiving_field, <= champ reçu (out)
int *request) <= identifiant de requête (out)

Initialise la réception des champs pour une communication non bloquante. La réception effective des champs se fait à l'appel de la primitive PCW_Wait_irecv

int PCW_Wait_irecv(char *coupling_id, <= identifiant du couplage (in)

int request) <= identifiant de requête (in)

int PCW_Issend(char *coupling_id, <= identifiant du couplage (in)
char *exchange_name, <= identifiant de l'échange (in)
int tag, <= tag MPI de la communication (in)
int stride, <= nombre de champs entrelacés à échanger (in)
int time_step, <= instance temporelle uniquement pour la visu. (in)
double time_value, <= valeur du temps uniquement pour la visu. (in)
char *sending_field_name, <= nom du champ à envoyer (in)
double *sending_field, <= champ envoyé (in)
int *request) <= identifiant de requête (out)

int PCW_Wait_issend(char *coupling_id, <= identifiant du couplage (in)
int request) <= identifiant de requête (in)

Autres primitives

int PCW_Set_output_listing(int iunit) <= numéro de l'unité logique fortran
Permet de rediriger les sorties CWIPI vers l'unité logique Fortran de son choix, par défaut les sorties sont écrites dans les fichiers de même nom que les branches suivi du numéro du proc si l'unité est parallèle.

int PCW_Dump_application_properties()
Ecrit les propriétés du couplage dans le fichier de sortie.

int PCW_Locate(char *coupling_id) <= identifiant du couplage (in)
Lancement explicite de la localisation (faite par défaut au premier échange sauf pour les communications non bloquantes).

int PCW_Update_location(char *coupling_id) <= identifiant du couplage (in)
Lancement explicite de la mise à jour de la localisation pour les maillages mobiles.

int PCW_Get_n_not_located_points(char *coupling_id, <= identifiant du couplage (in)
int *n_not_located_points) <= nombre de points non localisés (out)
Permet de connaître le nombre de points non localisés après un PCW_Locate

int PCW_Get_not_located_points(char *coupling_id <= identifiant du couplage (in)
int n_not_located_points, <= nb de pts non localisés (in)
int *not_located_points) <= indices des points non localisés

int PCW_Get_n_located_points(char *coupling_id, <= identifiant du couplage (in)
int *n_located_points) <= nombre de points localisés (out)
Permet de connaître le nombre de points localisés après un PCW_Locate

int PCW_Get_located_points(char *coupling_id <= identifiant du couplage (in)
int n_located_points, <= nb de pts localisés (in)

int *located_points) <= indices des points localisés

int PCW_Get_distance_located_points(char *coupling_id <= identifiant du couplage (in)
float *distance) <= distance aux élement localisé (out)
Permet de retourner les distances des points aux élements localisés du maillage distant les plus proches, le tableau distance doit être préalablement alloué par l'utilisateur aux nombre de point localisés.

int PCW_Dump_notlocatedpoints(char *coupling_id <= identifiant du couplage (in)
int n_not_located_points) <= nb de pts non localisés (in)

int PCW_Reorder(double *field_to_reorder <= champ à réordonner (inout)
int field_size, <= nb de points (in)
int stride, <= nombre de champs entrelacés (in)
double default_value, <= valeurs pour les points non localisés (in)
int *not_located_points, <= indices des points non localisés (in)
int n_not_located_points) <= nb de pts non localisés (in)

Permet de réorganiser les champs reçus en affectant une valeur par défaut aux points non localisés.

int PCW_Nb_coupling_names(int *nb_coupling) <= nombre de couplages (out)
Retourne ne nombre de couplages cwipi déclarés dans la carte d'identité de l'unité.

int PCW_Coupling_names(char **coupling_names) <= nom des couplages (out)
Retourne la liste des couplages déclarés dans la carte d'identité de l'unité.

Déclaration de coupling_names
C :

```
char **coupling_names;
coupling_names = (char **) malloc(nb_coupling*sizeof(char *));
 for (int i = 0; i <nb_coupling ; i++) {
    coupling_names[i] = (char *) malloc(PL_LNAME*sizeof(char));
 }
```

int PCW_Ho_options_set(char *coupling_id <= identifiant du couplage (in)
char *option <= « opt_bbox_step » (in)
char *value) <= valeur de l'option

La fonction « `PCW_Ho_options_set` » permet de modifier la valeur de paramètres concernant l'algorithme géométrique de localisation des points cibles dans le maillage source. Pour l'instant un seul paramètre est modifiable, il joue sur la finesse de l'échantillonnage des éléments lors du calcul approché de leurs boites englobantes. Dans les versions ultérieures de CWIPI, des paramètres supplémentaires pourront être modifiables par l'utilisateur comme le degré de précision souhaité dans les algorithmes de localisation dans les interfaces volumiques ou de calculs de distances aux interfaces surfaciques.

Fortran :
character(len=PL_LNAME), allocatable :: coupling_names(:)
allocate(coupling_names(nb_coupling))

int PCW_Coupling_connected(char *coupling_id, <= identifiant du couplage (in)
int *il_coupling_connected) <= 1 : relié dans PrePALM, 0 :non
Drapeau indiquant si le couplage est relié ou non relié à un autre couplage dans PrePALM.

int PCW_Coupling_target(char *coupling_id, <= identifiant du couplage (in)
char *unit_target_name) <= nom de l'unité relié par le couplage (out)
Retourne le nom de l'unité cible pour un couplage donné si celui ci est relié dans PrePALM.

int PCW_Nb_objects(char *coupling_id, <= identifiant du couplage (in)
int *nb_exch) <= nombre d'objets de couplage (out)
Retourne le nombre d'objets de couplage d'un couplage donné.

int PCW_Object_names(char *coupling_id, <= identifiant du couplage (in)
char **object_names, <= noms des objets de couplage (out)
int *object_modes) <= mode d'echange (PL_SEND, PL_RECV, PL_SENRECV)
Pour un couplage donné, retourne la liste des objets de couplage définis dans PrePALM.
Déclaration de object_names
C :

```
char **object_names;
object_names = (char **) malloc(nb_exch*sizeof(char *));
 for (int i = 0; i <nb_exch ; i++) {
    object_names[i] = (char *) malloc(PL_LNAME*sizeof(char));
 }
```

Fortran :
character(len=PL_LNAME), allocatable :: object_names(:)
allocate(object_names(nb_exch))

int PCW_Object_connected(char *coupling_id, <= identifiant du couplage (in)
char *object, <= nom de l'echange (in)
int *connect) <= 1 : relié dans PrePALM, 0 :non (out)
Drapeau indiquant si la communication est relié ou non relié à un autre objet dans PrePALM.

int PCW_Object_target(char * coupling_id, <= identifiant du couplage (in)
char *object, <= nom de l'echange (in)
char *object_target) <=nom du l'echange cible (out)
Pour un objet de couplage, retourne le nom de l'objet cible dans l'unité cible.

Primitives de stockage des localisations, à partir de la version 0.8.0 de CWIPI

Stockage en mémoire:
int PCW_Set_location_index(char * coupling_id, <= identifiant du couplage (in)
int index) <= numéro de la localisation (in)

Stockage sur disque:
int PCW_Open_location_file(char * coupling_id, <= identifiant du couplage (in)
char *filename, <= nom du fichier (in)
char *mode) <= 'r' pour lecture, 'w' pour écriture
Ouvre un fichier permettant de stocker les localisations.

int PCW_Close_location_file(char * coupling_id) <= identifiant du couplage (in)
Ferme le fichier.

int PCW_Save_location(char * coupling_id) <= identifiant du couplage (in)
Stocke sur disque la localisation courante, à appeler après un échange.

int PCW_Load_location(char * coupling_id) <= identifiant du couplage (in)
Lit la localisation sur disque à l'emplacement de la localisation courante, à appeler avant un échange.

25.2 Formulation Python

De façon similaire à l'interface Python de Palm (24.2), le module PCW comporte une classe Coupling qui regroupe les attributs et méthodes pour les communications, ainsi que des fonctions indépendantes en dehors de la classe.

Classe Coupling:

L'interface est très similaire au C, mais l'id du couplage est conservé tout le long dans l'objet, dans tous les appels de méthodes, on ne trouve donc pas l'argument char *coupling_id comme en C.

Attributs publics:

n_not_located_points
entier égal au nombre de points non localisés lors d'un mapping de maillage CWIPI

not_located_points
tableau numpy d'entiers de longueur *n_not_located_points* qui contient les indices des points non localisés

irecv_request
entier, référence unique à l'échange asynchrone en cours (obtenu par la méthode irecv)
issend_request
entier, référence unique à l'échange asynchrone en cours (obtenu par la méthode issend)
Méthodes publiques:

Créateur:
Coupling(char *coupling_id, int coupling_type, int entitiesDim, double tolerance, int mesh_type, int solver_type, int output_frequency, char *output_
format, char *output_format_option)
Crée un objet Coupling, pour l'utilisation voir la référence C/FORTRAN
arguments:
object, space: chaines de longueur inférieure à PL_LNAME
rank: entier
shape: entier ou tuple, liste ou tableau d'entiers
time, tag: entiers, par défaut PL_NO_TIME/TAG

Destructeur:
~Coupling()
Détruit l'objet de couplage CWIPI.

Coupling.define_mesh(int n_vertex, int n_element, ndarray coordinates, ndarray connectivity_index, ndarray connectivity)

Coupling.add_polyhedra(int id_nbelem, ndarray ida_face_index, ndarray ida_cell_to_face_connectivity, int n_faces, ndarray ida_face_connectivity_index, ndarray ida_face_connectivity)

Coupling.set_points_to_locate(int elem, ndarray coordinates)

Coupling.update_location()

Coupling.locate()

Coupling.get_n_not_located_points()

Coupling.get_not_located_points()

Coupling.get_n_located_points()

Coupling.get_located_points()

Coupling.dump_notlocatedpoints()

Coupling.reorder(ndarray reorder_field, int field_size, int stride, double default_value)

Insère les points non localisés manquants dans le tableau reorder_field avec leur valeur par défaut. Il faut auparavant appeler les méthodes *get_n_not_located_points* et *get_not_located_points* pour que les attributs *n_not_located_points* et *not_located_points* soient renseignés

arguments:

reorder_field: champ de valeurs obtenu lors du couplage CWIPI
field_size: dimension du champ ci-dessus
stride: le pas de parcours du champ
default_value: valeur par défaut utilisée pour tous les points non localisés

attributs d'entrée:

n_not_located_points: nombre de points non localisés obtenu par appel à *get_n_not_located_points*
not_located_points: tableau des indices des points non localisés. La méthode reorder met tous les éléments de ces indices à la valeur par défaut et décale les autres valeurs.

Coupling.set_interpolation_function(function f)

Coupling.set_interpolation_function_f(function f)

Coupling.sendrecv(char *exchange_name, int stride, int time_step, double time_value, char *sending_field_name, ndarray sending_field, char *receiving_field_name, ndarray receiving_field)

Coupling.recv(char *exchange_name, int stride, int time_step, double time_value, char *receiving_field_name, ndarray receiving_field)

Coupling.send(char *exchange_name, int stride, int time_step, double time_value, char *sending_field_name, ndarray sending_field)

Coupling.irecv(char *exchange_name, int tag, int stride, int time_step, double time_value, char *receiving_field_name, ndarray receiving_field)

Coupling.issend(char *exchange_name, int tag, int stride, int time_step, double time_value, char *sending_field_name, ndarray sending_field)

Coupling.wait_irecv()

Coupling.wait_issend()

Autres primitives

init()

finalize()

set_output_listing(file output_listing)

dump_application_properties()

dump_status(cwipi_exchange_status_t status)

26 Cartes d'identités

!PALM_UNIT			
Attribut	Description	type	
-name	Nom de l'unité	chaîne	
-functions	Liste des fonctions à appeler avec leur langage de programmation (F77 pour Fortran77, F90 pour Fortran90, C pour C ou C++ pour C++ python pour Python) (*)	Liste	
-object_files	Liste des fichiers objets (*.o) ou bibliothèques (*.a) nécessaires pour l'édition de lien de l'unité	Liste	
-parallel	Type de parallélisme : mpi pour MPI omp pour OpenMP no pour les unités monoproc	mpi omp no	option
-minproc	Nombre minimum de processeurs	entier	option
-maxproc	Nombre maximum de processeurs	entier	option
-comment	Commentaire	Liste	option
-class (**)	Permet de spécifier que l'unité est une unité d'algèbre	algebra ou rien	option
-library (**)	Library d'algèbre dans le cas d'une unité d'algèbre	chaîne	option
-mode (**)	Précision sur le contexte d'exécution dans le cas d'une unité d'algèbre	sticky ou no_sticky	option
-label (**)	aide sommaire définissant l'unité dans le cas d'une unité d'algèbre	texte	option
-help (**)	aide détaillée de l'unité	texte	option

(**) attribut spécifique aux unités prédéfinies d'algèbre.
(*) Si plusieurs fonctions sont spécifiées, l'unité consistera en un appel séquentiel à ces fonctions dans l'ordre dans lequel elles sont données.

!PALM_SPACE			
Attribut	Description	type	
-name	Nom de l'espace	chaîne	
-shape	Dimensions du tableau, les dimensions sont séparées par des virgules, le tout doit être entre ()	Expression d'entiers ou de constantes	
-element_size	Taille de chaque élément du tableau, possibilité de décrire des types dérivés à partir des éléments de base : PL_INTEGER, PL_REAL, PL_LOGICAL, PL_DOUBLE_PRECISION, PL_COMPLEX PL_AUTO_SIZE (*)	Expression d'entiers faisant intervenir des constantes génériques PALM	
-items	Liste contenant pour chaque élément une liste de deux éléments : le premier est le nom donné à cet item, le second est un espace déjà défini.	Liste	(*)
-comment	Commentaire	Liste	Option

(*) Dans le cas d'espace de type dérivé (cf. paragraphe suivant), si l'attribut **-items** est utilisé, alors l'attribut **-element_size** doit être initialisé à la valeur PL_AUTO_SIZE.

!PALM_OBJECT			
Attribut	Description	Type	
-name	Nom de l'objet	Chaîne	
-space	Nom de l'espace	Chaîne ou NULL (1)	
-distributor	Nom du distributeur	Chaîne ou NULL (2)	Option
-localisation	Nom de la localisation	Chaîne ou NULL (2)	Option
-time	ON si les communications utilisent un champ time différent de PL_NO_TIME, OFF sinon	ON/OFF	Option
-tag	ON si les communications utilisent un champ tag différent de PL_NO_TAG, OFF sinon	ON/OFF	Option
-intent	IN/OUT/INOUT selon que l'objet est utilisé dans un PALM_Get, un PALM_Put ou les deux.	IN/OUT/ INOUT	
-default	Valeur par défaut pour l'objet, uniquement pour les scalaires.	Valeur dépendante de l'espace	Option
-closedlist	Uniquement pour les scalaires, liste prédéfinie de valeurs que peut prendre l'objet. { {val1 comment1} {val2 comment2} … }	Liste de liste	Option
-comment	Commentaire	Liste	Option
-rank (3)	Rang de l'objet = nombre de dimensions du tableau	Entier	

(1) Les objets dont l'espace est défini à NULL dans la carte d'identité héritent des caractéristiques des objets lors de la définition des communications dans l'interface graphique
(2) NULL dans le cas des unités d'algèbre prédéfinies
(3) uniquement pour les unités d'algèbre prédéfinies

!PALM_ DISTRIBUTOR			
Attribut	Description	type	
-name	Nom du distributeur	Chaîne	
-type	Type du distributeur : `Regular` pour un distributeur de type regular `Custom` pour un distributeur de type custom `Regular_wh` pour un distributeur de type regular with halos	Chaîne : regular custom regular_wh	
-shape	Profil de l'objet distribué (étendues dans chaque dimension, séparée par des virgules, le tout entre parenthèses)	Expression d'entiers ou de constantes	
-nbproc	Nombre de processus de la distribution	Entier ou constante	
-function	Nom de la fonction de distribution	Chaîne	
-object_files	Nom du fichier objet contenant la fonction de distribution	Chaîne	
-comment	Commentaire	Liste	option

!PALM_ LOCALISATION			
Attribut	Description	type	
-name	Nom de la localisation	Chaîne	
-type	Type de la localisation : `Distributed` pour un objet distribué. `Replicated` pour un objet répliqué.	Chaîne : distributed replicated	
-description	Liste entre {} décrivant les processus incriminés, avec liste de la forme : **deb1**[**:fin1**[**:stp1**]] [; …]	Chaîne	

!PALM_ INTERN_COMM			
Attribut	Description	type	
-source	Nom de l'objet source	Chaîne	
-target	Nom de l'objet cible	Chaîne	
-time	Liste des temps valides	Liste	
-tag	Liste des tags valides	Liste	
-debug	Lancement ou non de la fonction debug de PALM	PL_DEBUG ou PL_NO_DEBUG	
-track	Ecriture ou non des informations sur la communication dans les fichiers de sortie de PALM	ON ou OFF	
-localassoc	Association des localisations	Chaîne	

-source_so_descriptor	Description du sous-objet source	Chaîne	
-target_so_descriptor	Description du sous-objet cible	Chaîne	
-dtm	Mode de gestion de la mémoire pour cet objet	MEMORY ou DISK	

Ce mot clé permet de définir des communications internes à une unité. Elle est utilisée pour composer des unités (regroupement de plusieurs unités en une seule) et permet de faire transiter les objets entre les différentes fonctions de base de l'unité sans devoir redéfinir les communications.