OpenPALM coupler version 4.3.0, User guide and training manual

T. Morel

▶ **To cite this version:**

T. Morel. OpenPALM coupler version 4.3.0, User guide and training manual. [Technical Report] CECI, Université de Toulouse, CNRS, CERFACS, Toulouse, France - TR-CMGC-19-70. 2019. hal-04730867

HAL Id: hal-04730867
https://cnrs.hal.science/hal-04730867v1

Submitted on 10 Oct 2024

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

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

OpenPALM coupler
version 4.3.0

User guide and training manual

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

April 2019

TR-CMGC-19-70

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

Table of Contents

1 Session 1: Getting acquainted with the Graphical User Interface....8
1.1 Introduction....8
1.2 Launching PrePALM....8
1.3 Inserting a branch....9
1.4 Editing the branch code....10
1.5 Compilation options setup....10
1.6 Generating the PALM service files....11
1.7 Compiling the application and executing it....12
1.8 Options of the PrePALM command....13
1.9 Summary of the main concepts....14
2 Session 2: Launching units....16
2.1 Introduction....16
2.2 From a stand alone code to a PALM unit....16
2.3 An example of a PALM unit....16
2.4 ID cards....17
2.5 Loading the units ID cards....17
2.6 Launching the units....18
2.7 Parallel computing....20
2.8 The performance analyser....21
2.9 Summary of the main concepts:....22
3 Session 3: The blocks....23
3.1 General comments on blocks....23
3.2 Launching the units inside the driver executable....27
3.3 Passing arguments to the executables started by PALM....28
3.4 Summary of the main concepts:....28
4 Session 4: More about branches and units....29
4.1 Launching by another branch....29
4.2 The steps....29
4.3 The scripts....30
4.4 Launching a MPI parallel unit....30
4.5 Launching an OpenMP parallel unit....31
4.6 Summary of the main concepts....32
5 Session 5: Communications....33
5.1 Introduction....33
5.2 Preparation of the units, the PALM primitives....34
5.3 The communications in PrePALM....37
5.4 Time lists....38
5.5 Hardwired values....40
5.6 The NULL space and space inheritance....40
5.7 Communications attributes....42
5.8 Summary of the main concepts....43
6 Session 6: Predefined units....45
6.1 Introduction....45
6.2 Summary of the main concepts....47
7 Session 7: Derived data type objects....48
7.1 Introduction....48
7.2 Memory contiguous objects....48
7.3 Non contiguous objects....49
7.4 Summary of the main concepts....53

8 Session 8: Time interpolation......54
8.1 Introduction......54
8.2 Units Preparation......54
8.3 Monitoring the application in real time......55
8.4 Steps, events and actions......57
8.5 The memory slaves......59
8.6 Summary of the main concepts......60
9 Session 9: Space inheritance and dynamic objects......61
9.1 Summary of the main concepts......65
10 Session 10: Assembling objects in the BUFFER......66
10.1 Summary of the main concepts......67
11 Session 11: Parallel communications......68
11.1 Introduction......68
11.2 The distributors......69
11.3 Block cyclic distributors......70
11.4 'CUSTOM' distributors......71
11.5 Examples of distributed objects......72
11.6 Localisations and process associations......74
11.7 Summary of the main concepts......78
12 Session 12: Sub-objects......79
12.1 Summary of the main concepts......82
13 Session 13: Read and write in files, geophysical fields interpolation......83
13.1 Summary of the main concepts......86
14 Session 14: Using a minimiser......87
14.1 Summary of the main concepts......90
15 MPI-1 Mode......91
15.1 Introduction......91
15.2 Restrictions at the level of the PALM coupler......91
15.3 Executing an application in MPI-1 mode......92
15.4 An application example in MPI-1 mode......92
15.5 Summary of the main concepts......95
16 Grid-based Interpolation with CWIPI library......96
16.1 General information......96
16.2 The bases of unstructured meshes in CWIPI......96
16.3 First steps with CWIPI under OpenPALM......96
16.4 A more complete exercise......104
16.5 Definition of the coupling in PrePALM......106
16.6 Exercise 1: initial instrumentation......111
16.6.1 Initialisation of the coupling......111
16.6.2 Creation of the coupling environment......111
16.6.3 Definition of the mesh support......112
16.6.4 Data exchange......113
16.6.5 Processing of the received data......113
16.6.6 Deletion of the coupling environment......113
16.6.7 Running the application and analysing the results......113
16.7 Exercise 2: detection of non located points......116
16.8 Exercise 3: time-varying coupling......117
16.9 Exercise 4: time-varying coupling with moving coupling surface......117
16.10 Advanced topics with CWIPI......118
16.10.1 Definition of the interpolation points......118
16.10.2 Asynchronous communication......118

16.10.3 User defined interpolation....119
16.10.4 Python interface....120
17 Connection of an external code to a PALM application....121
17.1 Introduction....121
17.2 How it works....122
17.3 Connecting a single processor code to PALM....122
17.4 Connecting a parallel code to PALM....126
17.5 To go further: IP connection of an external code....128
17.6 Summary of the main concepts....136
18 Writing PALM units in Python....137
18.1 Python unit....137
18.2 Object oriented Python interface....138
18.3 Dynamic communication via OpenPALM....139
18.4 Parallel codes: Get MPI communicator....140
18.5 Python help function....140
19 Writing PALM units in interpreted languages such as Perl or Tcl/Tk....141
19.1 Introduction....141
19.2 PALM unit in perl....143
19.3 PALM unit in Tcl/Tk....145
19.4 Summary of the main concepts....150
20 PALM Installation....151
20.1 Introduction....151
20.2 Installation of the PrePALM graphical user interface....151
20.2.1 Pre-requirements....151
20.2.2 PrePALM command definition....151
20.2.3 STEPLANG interpreter installation....152
20.2.4 Installation of the OASIS library, if needed....152
20.3 Installation of the PALM library....152
20.3.1 Pre-requirements....152
20.3.2 Installation....153
20.3.3 Example of installation on a Linux workstation....155
20.4 Summary of the main concepts....156
21 Some more or less specific utilities....157
21.1 Default value and choice from a list of pre-defined values for the units input plugs....157
21.2 Some subtleties on the time stamp: conversion to/from dates....157
21.2.1 Introduction....157
21.2.2 Two-ways coupling a.k.a. strong coupling....158
21.2.3 One-way coupling a.k.a. forcing....158
21.2.4 Conversion of integer time stamps from/to dates....159
21.3 Dynamic verbosity settings....159
21.4 Checking the object contents: palm_debug.f90/c....160
21.5 Print out the object contents: the PALM_Dump primitive....160
21.6 Summary of the main concepts....160
22 Batch file for PrePALM....161
23 Palm Glossary....163
24 Reference guide of the PALM primitives....168
24.1 C and Fortran formulation....168
24.2 Python formulation....172
25 List of PCW primitives for the CWIPI library....177
25.1 C and Fortran Formulation....177
25.2 Python Formulation....182

26 Identity Cards syntax..185

Introduction

The OpenPALM coupling tool is based on the PALM coupler under development at CERFACS since 1998 and on the CWIPI interpolation library developed at ONERA DSNA/ELCI. OpenPALM is being co-developed as free software distributed according to LGPL license by CERFACS and ONERA since January 2011.

The software's specificity are its ability to define complex algorithms around the computation codes to be coupled, as well as its efficiency and flexibility to transfer data between the codes. MPI technology which is in charge of process launching makes OpenPALM a portable and optimised tool on any unix/linux machine.

OpenPALM has various coupling functionalities such as geometric interpolation in any kind of mesh.

This manual is split into several training sessions and shows step-by-step all functionalities of OpenPALM. First use of the software is facilitated by the graphical interface PrePALM.

1 Session 1: Getting acquainted with the Graphical User Interface

1.1 Introduction

The use of the PrePALM graphical interface is a mandatory step in the development of a PALM application. It is very important to become acquainted with all the subtleties of this interface for the best usage of the coupler. In general you will spend more time in manipulating the PrePALM interface than in modifying the source code of the programs to be coupled. The principal reason is that the PALM interfaces (API) are very generic and not intrusive and the coupling algorithm is entirely described in the graphical user interface. Moreover, most of the coupler functions are defined via the graphical user interface which also controls the coherence of the input data.

1.2 Launching PrePALM

Once the PrePALM software is installed, it is recommended to define an alias exporting the path of the PrePALM installation directory and creating a shortcut for the interface invocation (*cf.* § 20.2.2)

In tcsh it would look like

```
alias prepalm ’setenv PREPALMMPDIR install-path ; $PREPALMMPDIR/PrePALM_MP.tcl \!* &’
```

while in bash

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

Usually, this command finds its place in the user’s shell configuration file (`.cshrc`, `.bashrc`, ... according to the Unix shell in use) or in a script. In the same file, one can also define the environment variable `PREPALMEDITOR` that selects the text editor invoked by the graphical user interface. If this variable is not initialised, PrePALM will use the `vi` editor. If for example you are more familiar with emacs, you may declare:

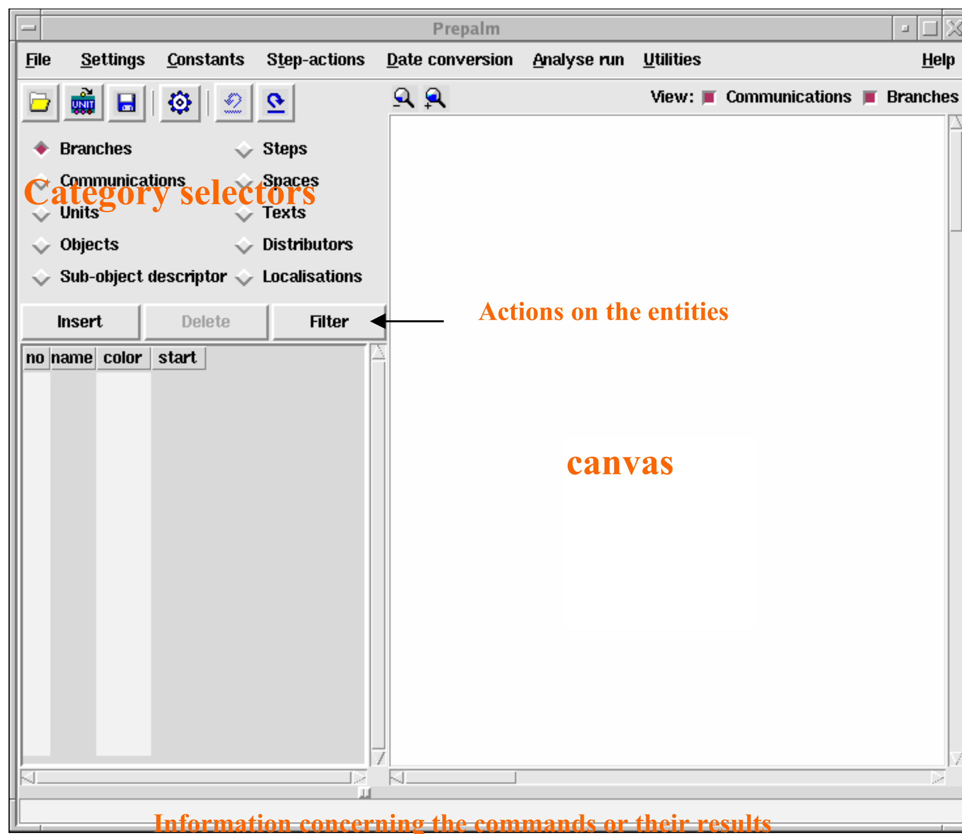
`setenv PREPALMEDITOR emacs` or `export PREPALMEDITOR=emacs`

When the alias is defined, you simply enter the following command to start the graphical user interface:

```
> prepalm
```

Let's try!

- Move to the directory `session_1`
- Launch the **PrePALM** graphical user interface



PrePALM graphical user interface.

1.3 Inserting a branch

The first operation we will carry out with PrePALM will consist in inserting an algorithm *branch*. Branches are used to schedule the launching of the PALM *units*. We will talk later about PALM units.

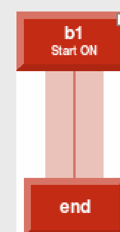
PALM is a dynamic coupler. The sequence of elementary actions follows the logic of a programming language with variables declarations, instructions and control structures (loops and conditional switches): all this is defined in the branches.

It's time to work!

- Select the **Branches** category
- Click on the Insert button

A window appears

- Give your branch a name, for example `b1`, and confirm
- In the canvas, you should see



The upper rectangle represents the beginning of the branch, the lower one represents the end of the branch, and the large line connecting them will be used to symbolize the progression between the units.

Do it!

- Double click on one of the two rectangles
- Modify the branch color: you have to guess how to do it!
- Right click and drag one of the two rectangles to move them individually
- Right click and drag the large line to move the whole branch
- Click on the little white rectangle to open/close the branch

Important: in case you did not notice it, a contextual help on the actions in the canvas appears at the bottom of the graphical user interface.

- Move the mouse on the various PrePALM zones to read the help messages.

1.4 Editing the branch code

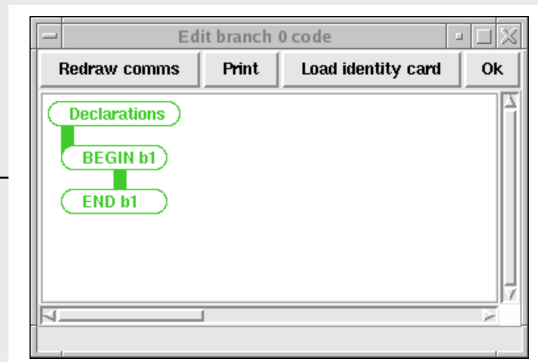
We will now create our first PALM application, the traditional "Hello World ". To do so, we will insert a Fortran 90 region in the branch code.

GO!

- Double click on the branch large line

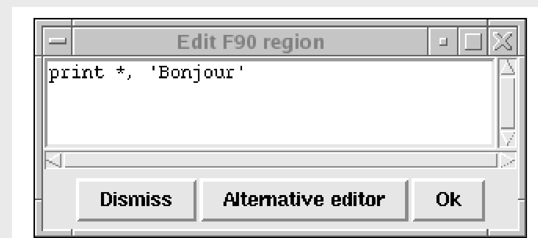
The `Edit branch code` window appears

- Click here



- Select

- Write the F90 instruction



1.5 Compilation options setup

PrePALM creates the PALM application by itself, including the application `Makefile`. To do so, it needs to know a number of things like the place where the PALM library is installed, the name of the F90, F77, C and C++ compilers, the compilation options.

For a better portability, the platform independent `Makefile` includes a file (`Make.include`) which is machine dependent. The `Make.include` file can be edited in the graphical user interface or a set of options can be loaded from a file with the `.mak` extension.

Let's do it!

- Menu `Settings` => `Palm Makefile options edit`
- Fill the fields or click on the `Load_options` button
- Click on the `Save Options` button to create a `.mak` file
- Check the box `Save as default` if you want PrePALM to start with the current set

Now, it's time to save our PrePALM file. PrePALM files carry the `.ppl` extension (for PrePalm Language)

Save!

- Menu `File` => `Save PrePALM file as (ppl format)` or icon
- Move to the directory training/`session_1`
- Give the file a name, like `session_1` for example

1.6 Generating the PALM service files

We can now generate the PALM service files, needed for the application

Ask PrePALM to work!

Menu `File` => `Make Palm files` or icon

Check the boxes as follow:

Click on `Ok`

Let's take a look on the files generated in the directory `session_1`.
A `ls` command gives:

```
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
```

After having generated the file `session_1.ppl`, PrePALM created the PALM service files:

- `Makefile` will allow us to compile the application with the `make` command.
- ***Make.include*** is a file included by `Makefile` containing all information specific to our machine. When you port your work on another machine, you'll just have to modify this file.
- `palm_init.c`, `palm_driver_servicesc.c`, `palm_driver_services.f90`, `palm_entities_services.f90` and `palm_debug.f90` are the PALM service files used to compile the application.
- `palm_user_param.h`, `palm_user_paramc.h` can contain constants. Their usage will be detailed later on.
- `session_1.pil` is a coupler input file: it contains part of the information entered in the ppl file compiled to be readable run-time by the coupler.

1.7 Compiling the application and executing it

The graphical user interface does not launch the PALM application. It has to be compiled and executed on the computer of your choice (parallel or not) . For this tutorial you are most probably going to use the same workstation (Linux) to run the graphical user interface and to execute the coupled applications but in another context you could likely use a PC for the graphical user interface and a mainframe or supercomputer for the execution.

Parallel computing and message passing in PALM are based on the MPI2 standard, some implementations meet this standard like Mpich Openmpi or Intelmpi on Linux. According to the library used (and version) it may be necessary before launching a parallel application to launch a daemon process

Let's try!

- `make`
- `mpd& (if necessary with some mpich versions)`
- `mpirun -np 1 ./palm_main`

You should read on the screen something like "`Bonjour`"

Exercise 1:
Insert a loop (from 1 to 5) around the printing of "Bonjour" and add a condition inside the loop in order to have:

```
Bonjour 1
Bonjour 2
Third bonjour
Bonjour 4
Bonjour 5
```

Remark: insert loops and conditions in the branch code by clicking on the vertical line. Move the end of the loop by selecting `Select to move` then `Move ENDDO line here`.

1.8 Options of the PrePALM command

Hitherto we have started the PrePALM command with no arguments. Nevertheless there are different options to start the PrePALM as a compiler creating the service files. To know the full syntax of the `prepalm` command simply type

```
> 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)
```

To start the interface and directly load an existing file or to create a new file with a given name, issue :
> prepalm *file*.ppl

For some large applications or in some particular situations it is quite useful to use PrePALM from the command line with the `--c option` to generate the `.pil` and, optionally, the service files starting from the `.ppl` file with no need of starting the whole graphical interface. Some typical situations are:

- keep some versions of the `.ppl` input file for different configurations and rename and compile only the one needed for the current application
- when you'll become an expert user you'll see that it is some time quicker to manually edit the `.ppl` file without passing by the user interface and compile it on the command line. It is often the case if you only change the values of some PrePALM *constants* (*cf.* Chapter 3)
- when you work on a machine without X11 capabilities, as, for instance, if you want to submit the generation of the `.pil` and the service files as a batch job on a supercomputer.

The *compiler* mode , generating the `.pil` file and all the service files, is activated by:
> prepalm file.ppl --c

The `--p` option asks the compiler to generate only the `.pil` file but no service files.

The `--t` option activates the compiler mode and activate the tracing options for the performance analysis (cf. §2.8).

The `--l` option replace the values of the PrePALM constants by what is read in the F90 file indicated as an argument (cf. Chapter 3).

In all case the `prepalm` command requires a machine with X11 capabilities (where you could launch the graphical interface, to be clear). For the environments with no X11 capabilities, the alternative command `prepalm_tclsh.tcl` is available. Before invoking this command you have to explicitly set the `PREPALMMPDIR` environment variable (cf. § 1.2). The syntax and the options of this command are the same as for the `prepalm` command.

1.9 Summary of the main concepts

In this first session you've learnt how to start the Graphical User Interface PrePALM and how to create and edit or manipulate the basic entity used to describe a PALM algorithm, i.e. a *branch*. Then you have seen how to generate, compile and run the PALM application and where to look for its output.
To conclude this session you have seen the different syntaxes of the `prepalm` command and how to use it as a command line compiler to generate the files needed by the PALM application.

You should have realised that the graphical user interface PrePALM does not start or drive the execution of the coupled application. It is just used to enter the control structures, some F90-like regions and to prepare the ingredients to compile the application on the target machine.
Notice that a number of C and F90 service files are created by PrePALM when you click on "*Make PALM files*". For instance, any possible syntax error in the F90 regions of a branch will only be detected when compiling the applications. Once compiled, this services files are linked against the PALM driver library (*libdrv.a* in the PALM install directories) to obtain the palm_main executable. This is an MPI application, master of all the PALM coupled applications. From now on, we'll mention this leading process as the *driver*. This is the executable we start on the target machine and that, in most of the cases, takes care of organising the rest of the application.

In this example, under LAM, we can start it simply as `./palm_main`, but it would be syntactically more correct to use the full form `mpirun -np 1 ./palm_main`.

Notice that the syntax used in PrePALM to describe the control structures and the code regions is close to F90: in reality you are not writing a Fortran code but an instruction tree which is interpreted run-time by the PALM driver. The first role of the PALM driver is to interpret the branch code. If there are several concurrent branches (as we'll see in the following) the driver act as a *scheduler* to decide what branch will receive the control for the next move and so on: at every scheduler iterate, the driver examines all the branches pending instructions, choose the next one on the base of a simple but not trivial priority criterion and make the algorithm advance.

There is a useful feature which is inherited from F90: all the F90 regions of a branch are translated as subroutines contained in a single F90 module where all the variables declared in the branch are defined. In this way the different variables are known for the whole life of the branch, not only of a single F90 region. One can use a complete F90 syntax inside the regions, but it is mandatory to close all the control structures in the same region where they have been opened (e.g. `DO` and `END DO` in the same region) because every region translates into a different routine. It is therefore very important to understand the difference between a DO loop inside a F90 region and a branch DO control structure entered via the graphical branch editor.

This flexible syntax is very useful to define complex and dynamic coupling algorithms. One has nevertheless to remember that the branch code should not be used to carry on real computations but just to drive the sequential or concurrent execution of coupled elementary computational tasks which are carried on by the PALM units that we are going to get to know in the next session.

2 Session 2: Launching units

2.1 Introduction

A PALM *unit* is a chunk of user code that can be invoked by calling a C or C++ function or a FORTRAN subroutine without arguments (or, as we'll see later, in some cases it can also be a precompiled black-box code or an interpreted language procedure). A unit must be seen as an elementary task that can be scheduled inside a coupling algorithm. The units' granularity must be chosen according to the applications and to the expected degree of modularity. For models coupling a very coarse grain is generally sufficient; each unit can correspond to a complete computer code. For data assimilation applications, where the goal is to organize operators (direct and adjoint model, observation operator,…) in order to generate one or more assimilation algorithms (3DVAR, 4DVAR...), a finer grain is often essential.

PALM makes it possible to assemble units written in different languages. For the moment our units will be simple independent subroutines without communications. The goal is to show how to interface an existing computer code to make a PALM unit out of it.

2.2 From a stand alone code to a PALM unit

In general, to couple computer codes, we start from existing codes. The entry point of these codes is the `main` (C or C++) or `PROGRAM` (F90 and F77) instruction. The first thing to be done is thus to replace `PROGRAM` by `SUBROUTINE` (or `main` by another function name). The only constraint is to have the source code of the program to be coupled or at least the of the main routine. If we cannot access the main program source code, the situation is not desperate (although less comfortable) if it is possible to call some user-defined routines from within the black box code. This opportunity is in general available for the industrial codes which are distributed without the source listings (*cf.* Chapter 17)

2.3 An example of a PALM unit

You will find in the directory `session_2` four examples of basic PALM units written in C, C++, F77 and F90. Let's take a look to the F77 one:

```
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
```

The first four lines are comments. There is nothing original in the following apart from the PALM header include palmlib.h and the writing to the logical unit `PL_OUT`. `PL_OUT` is simply associated to a PALM output file which can be used to follow the execution of the different units (it can be referenced as unit `PL_OUT`, because it is defined in `palmlib.h`)

2.4 ID cards

Let us go back to the first four lines. The aim of these lines is to declare the `unit77` subroutine as a PALM unit recognized by the PrePALM graphical user interface. These lines are comments, so they can be inserted in the unit source code, which is practical for the maintenance of the code. But the user is not forced to write these lines at this place, they could be in a separated file. A complete help for the writing of the ID cards is accessible in the PrePALM `Help` menu and in Chapter 26.

Description of the various fields in our example:

`-name unit77`: gives the unit a generic name. Names given inside PALM and PrePALM should not contain spaces nor dots.

`-functions {f77 unit77}`: allows to specify which fortran77 subroutine must be called. Notice the f77 in front of unit77: it specifies the programming language used for the unit77 unit. Notice also the braces, they are used to describe lists; indeed it is possible to declare a unit which calls several functions in sequence.

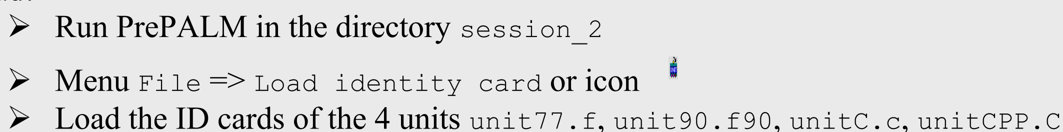
`-object_files {unit77.o}`: allows to specify, for the application compilation and link, the name of the `.o` files (objects) the unit needs. If, for example, unit77 calls another subroutine which is not in the same file, it is then necessary to add the `.o` file containing the other subroutine. During the PALM application compilation, if the `.o` file does not exist or is not up to date following the modification of the source program, the PALM `Makefile` will try to generate it by using default compilation rules. In our example it will thus create `unit77.o` starting from `unit77.f`

Important remark:
If your unit is for example a computer code calling many subroutines written in many files, it is not wise to describe in the field object_files all of the `.o` files.
In this case it is better to proceed like this:
You should compile in advance (without the link phase) the source files, and, rather than creating the executable file, you may create a library (`.a`). Then, the library name has to be listed in the `object_files` field.

See the differences!
- In directory `session_2`, open the files: `unit77.f`, `unit90.f90`, `unitC.c`, `unitCPP.C`
- Notice the difference depending on the programming languages

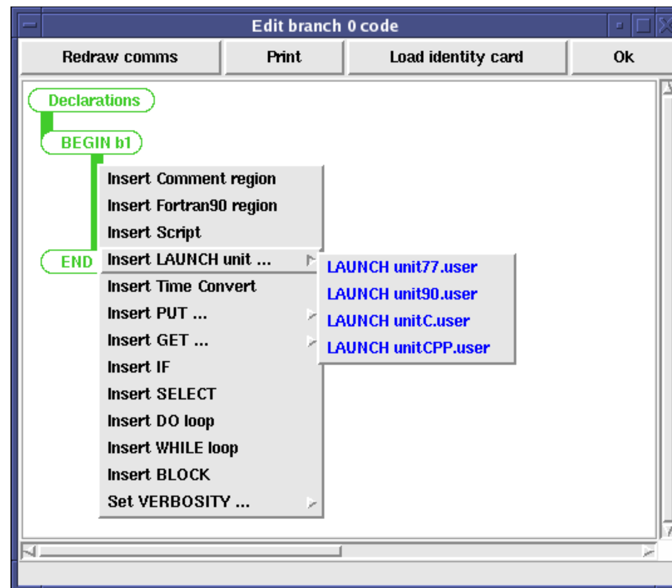
2.5 Loading the units ID cards

Before launching the units inside PrePALM, it is necessary to load their ID cards.

Load!
- Run PrePALM in the directory `session_2`
- Menu `File` => `Load identity card` or icon 
- Load the ID cards of the 4 units `unit77.f`, `unit90.f90`, `unitC.c`, `unitCPP.C`

2.6 *Launching the units*

The launching of the units is made at the branch code level. By clicking on the continuous line after the instruction `BEGIN`, this menu appears:



Try it!

- Create a b1 branch
- Edit the branch code
- Launch the 4 units, one after the other
- Save your application
- Build the services files
- Set the maximum number of concurrent processes resource in `Settings` => `Palm execution setting` => `Max number of procs`: **1**
- Compile and execute **`mpirun -np 1 ./palm_main`**

The result of your "simulation", that is to say the writings in the `PL_OUT` file, is in the file 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
```

If you look closer in the directory `session_2`, you can see that there are now 5 executable files:

- `main_unit77`
- `main_unit90`
- `main_unitC`
- `main_unitCPP`
- `palm_main`

You will also find the source codes (`main_unit*.c`) which have been generated by PrePALM (service files). If you open the file main_unit90.c, you have:

```
#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);
}
```

The `unit90` FORTRAN subroutine call was encapsulated in this main program. Initialization and finalization of PALM is done here so that the user does not have to add these calls in the units' program source. The unit name ("main_unit90") has been generated by the graphical interface PrePALM. Care must be taken when the original code shall be instrumented. The call to PALM_Init must use the same name chosen by PrePALM. If several instances of the same code are executed, these names change.

`f2c_name()` is a precompilation macro (defined in PALM for the preprocessor) intended to convert automatically the function names in order to ensure the compatibility between the FORTRAN and C languages.

A PALM application is an MPMD application (Multiple Program, Multiple Data): several distinct executable files can communicate between them. The launching of the units is made by the PALM driver (`palm_main`). As the units are on the same branch (computational sequence), PALM launches the executable files sequentially, one after another.

Important: commands to remember for making a little cleaning with the Makefile generated by PrePALM:

↳**`make tidy`**: deletes the output files of PALM only. This is necessary to start again an execution because the output files are opened in "append" mode (writing at the end of the file). If `make tidy` is not issued, the output of the former executions is kept, and the new outputs will be appended after the previous ones.

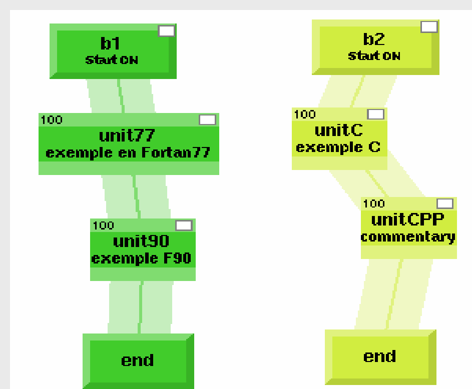
↳**`make clean`**: "`make tidy`" AND deletes the executables and the `.o` files

↳**`make allclean`**: "`make clean`" AND deletes the service files generated by PrePALM. It keeps only what is essential.

2.7 Parallel computing

We now will create an application where the units will be executed in parallel. PALM handles two levels of parallelism. The first is a task level parallelism which can be simply defined in the graphical user interface by creating several branches. The second is an internal parallelism, i.e. within the units. We will come back later on to this second level of parallelism.

Parallelize it!
- Create a second branch
- Edit both branch codes at the same time
- In the first branch: click on `Unit_C` and `Select to move`.
- In the second one: click on `Move - LAUNCH line here`.
- Repeat for `Unit_CPP`, then close the branch editors
- You can make the canvas look nicer by moving the units: left click to select a unit (it becomes red), then right click to move it. You should have something like that:



- Change the max. nr. of processes resource: `Settings` => `Palm execution setting` => `Max number of procs`: **2**
- Save, generate the service files, `make clean`, `make`, **`mpirun -n 1 ./palm_main`**

Congratulations!
You have executed your first (?) parallel computing almost without realizing it, without anything to know about MPI, only by drawing! It is one of the PALM features: one can make parallel computing without any further specific knowledge. The results are now in two distinct files: the file b1`_000.log` contains the branch 1 units output and the file b2`_000.log` the branch 2 units output. Open these files to see the results.

Remark:
The command mpirun -n 1 ./palm_main has launched a single program (palm_main, also called the PALM driver). This program will schedule the launch of the other executable files (main_unitC,

main_unit77, main_unit90, main_unitCPP) in a parallel run or sequentially depending on the definitions done in the PrePALM branches.

2.8 The performance analyser

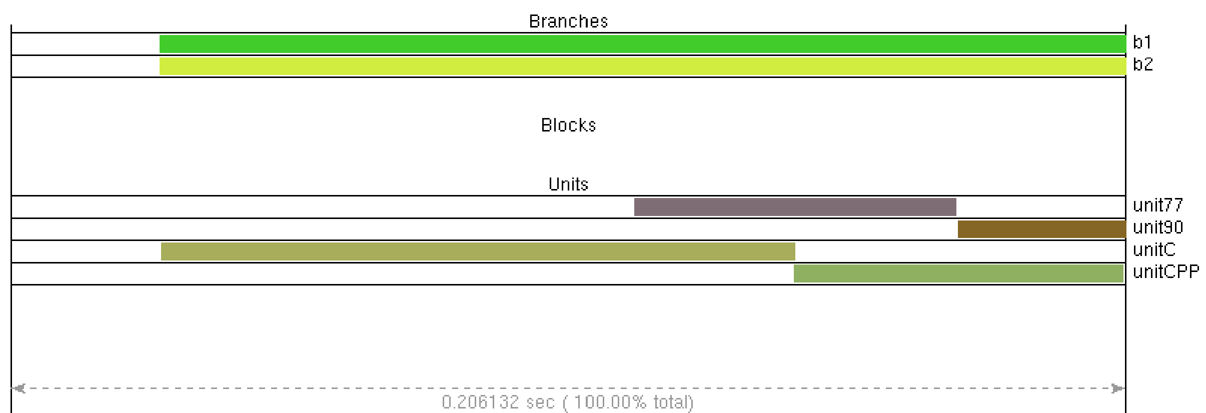
By choosing the right option in PrePALM, PALM is capable of generating trace files, making it possible to analyse the application performances or to replay the execution.

Analyze!

- In menu `Settings` => `Palm execution settings`, check this box:

  Trace execution for animation (file palmperf.log)

- Service files; `make clean`; `make` ; `./palm_main`
- Menu `Analyse run` => `Load file`: choose `palmperf.log`
- A summary report gives you the execution time per unit
- Menu `Analyse run` => `Play`: it allows you to see the launching of the units graphically (find by yourself how to do it!)
- Menu `Analyse run` => `Performances analyser`: produces the following picture



On this example, we can see that `unit77` runs at the same time as `unitC` and `unitCPP`.

Management of the processes (processors)

As we have seen, PALM manages the processes according to the units to be launched: sequentially inside the branches, or in parallel between the various branches. In all cases PALM will not exceed the maximum number of concurrent processes specified in PrePALM. The management is dynamic inside a static envelope. This is necessary in order to use the proper number of processors in a batch job and to avoid any overloading of a parallel machine.

If PALM does not have enough resources, for example if there are two branches and only one processor, PALM is able to alternate the units executions. When two units are concurrent, PALM looks at the priority level, and determines which one must be launched first. By default this priority level is set at 100 (it is located in the top left corner of the box representing the unit), but it can be modified by the user: a left click on the priority decrements it, a right click increments it.

Exercise 2:
Launch the PALM application with 2 branches but with only one process, play with the priorities in order to run these units in this order:

- unit77
- unitC
- unit90
- unitCPP

Check the effects with the *Play* animation and with the the *Performances analyser* timeline

2.9 Summary of the main concepts:

In this session you have seen how to define and launch a PALM *unit*.
In particular you have got to know what an identity card is and its basic syntax.
You have also known some more features of the PrePALM generated makefile targets.
In the following you have learnt how to run several concurrent parallel branches.
Then you have seen how to analyse after the execution, the performances of a PALM application.
You are now able to manage the computational resources and to play with the static processors envelope and with the execution priorities of the units to optimise the execution of your program.

3 Session 3: The blocks

3.1 General comments on blocks

So far we have seen that each PALM unit is transformed into an executable file and started (spawned) independently. For two main reasons (memory sharing and launching time optimization) it is interesting to gather several units in a single executable program, called a *block*.

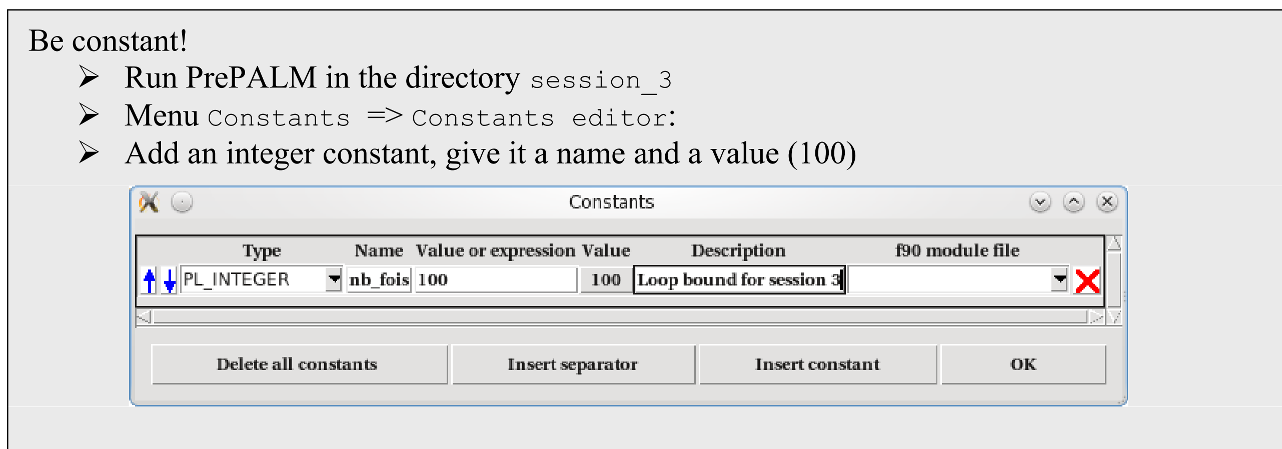
We will build a new application in which we launch a unit in a DO-loop. To be able to easily parametrise the number of loops of this unit, we will use the PrePALM *constants*.
PrePALM constants are implemented as FORTRAN `PARAMETER` declarations, or C `#define`. These values are known at compile time and cannot be modified during the execution.

PrePALM provides a menu to declare the constants (type, name, value or expression). These constants may depend on each other through an arithmetic expression.

It can be useful to have access to the values of the constants defined in the graphical user interface itself from inside the units. For this purpose, PrePALM can generate user files to be included in the unit's source programs according to their programming language:

- palm_user_param.f90 F90 `use palm_user_param`
- 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`

Each of these files contains the same declaration of the PrePALM constants. They can be used for example for dimensioning the static arrays.
The user can optionally choose additional file names (fortran90 module) to create subsets among the constants.
In addition, for the sake of readability it is possible to introduce separators in the constants list.
In our application, we will define only one constant which will be used only in the graphical user interface for the control of the upper bound of the DO-loop.



For this application, we will use 2 units that you may find in `unit_1.f90` and `unit_2.f90`.
Here is the code for unit_1:

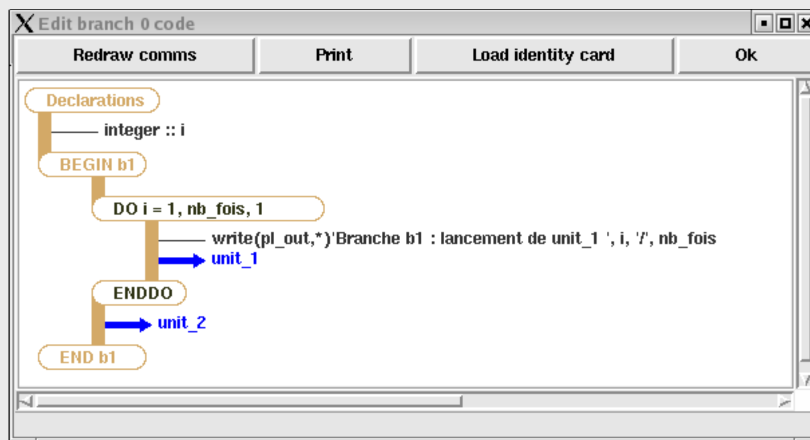
```
!PALM_UNIT -name unit_1\
```

```
!           -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
```

This unit prints out the value of a variable (`iglobal`) which is incremented at each unit_1 subroutine call. This variable is defined as a global variable through the COMMON declaration. The unit unit_2.f90 is identical to unit_1 except that it does not increment the variable `iglobal`.

Do not block!

- With the branch code below you are able to build the first branch of the application.

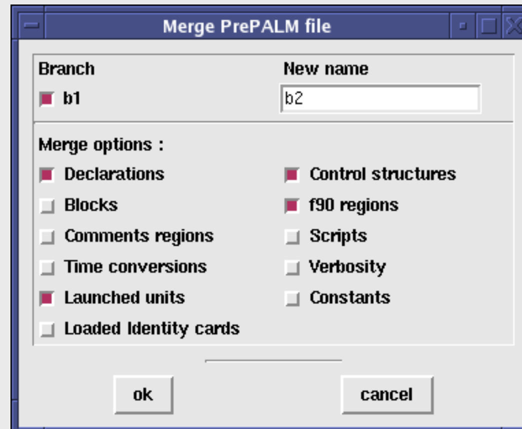


- Save your file.

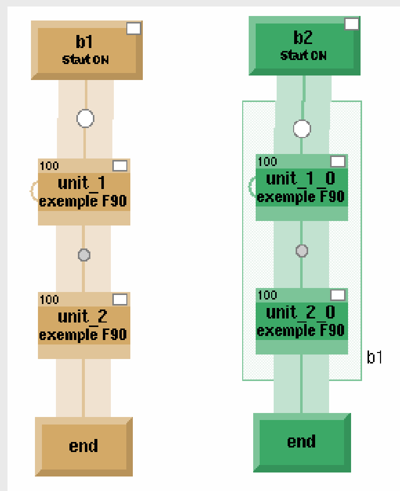
We now will build a second branch. As it will be very similar to the first one, we will use the PrePALM **Merge** function which makes it possible to transfer a part of an existing application to the current one. In our case, we will use the file which we have just saved.

"Copy" yourself!

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



- In the canvas you find the two duplicated branches (right click on the branch to move it), for a better look, change the color of one of the branches.
- Open the second branch and insert a *block* before the DO-loop
- Move the end of the block just before the end of the branch
- Change b1 by b2 in the Fortran region
- Your application should look like this:



- Save and execute the application with 2 processes.

The block appears in the PrePALM canvas like a greyed rectangle containing the units. Notice that in addition to the PALM driver, the application consists of just 3 executables, the units `unit_1_0` and `unit_2_0` having been gathered into a single executable named main_block_1.

Consequences of the block construct:

1) Sharing the memory

If we look at the file b1_000.log, output file of the branch b1, we notice that the value of the global variable iglobal is equal to 1 each time unit_1 runs. This is normal since, in the loop, we launch an executable each time and it terminates after each execution. For the same reason, the value of iglobal in unit_2 is 0.

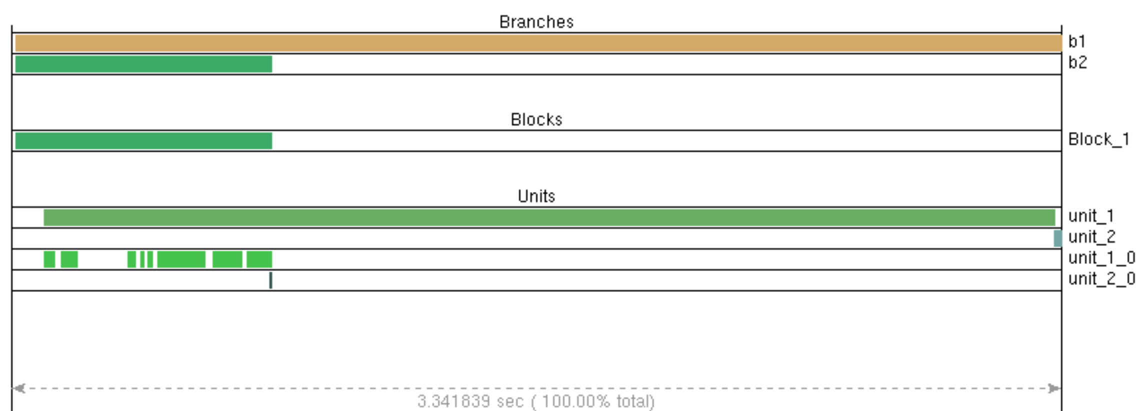
On the other hand in the file b2_000.log, output file of the branch b2, the value of iglobal is preserved between each instance of unit_1_0. This is normal because now, the launch of a unit corresponds to the call of a subroutine in the main program executing the loop. For the same reason, now the unit unit_2_0 knows the value of iglobal, calculated by unit_1_0.

Within the blocks, one can exchange information between units through global variables. This data sharing mechanism is not recommended in the most usual PALM approach because units sharing global data are not fully independent (later on, we will see a better way to proceed), but it can be very useful to split a single legacy code into "functionally" independent units or to optimise a coupled application.

Sharing global memory presents some advantages but also can have some drawbacks. For example, if you have two independent units in which we declare some large static arrays (or dynamic arrays without deallocation), the memory size of the executable assembled in a block will be the sum of the memory sizes of the two units which can pose serious problem for hardware memory.

2) Optimising the CPU time

If you open the PrePALM performance analyser for this application, you will see (depending on your computer) that the branch b2 runs faster than b1:



This result can be expected since in the branch b1, the executable unit_1 is loaded in memory and launched 100 times, whereas in the branch b2, a single program containing the DO loop is loaded and run only once.

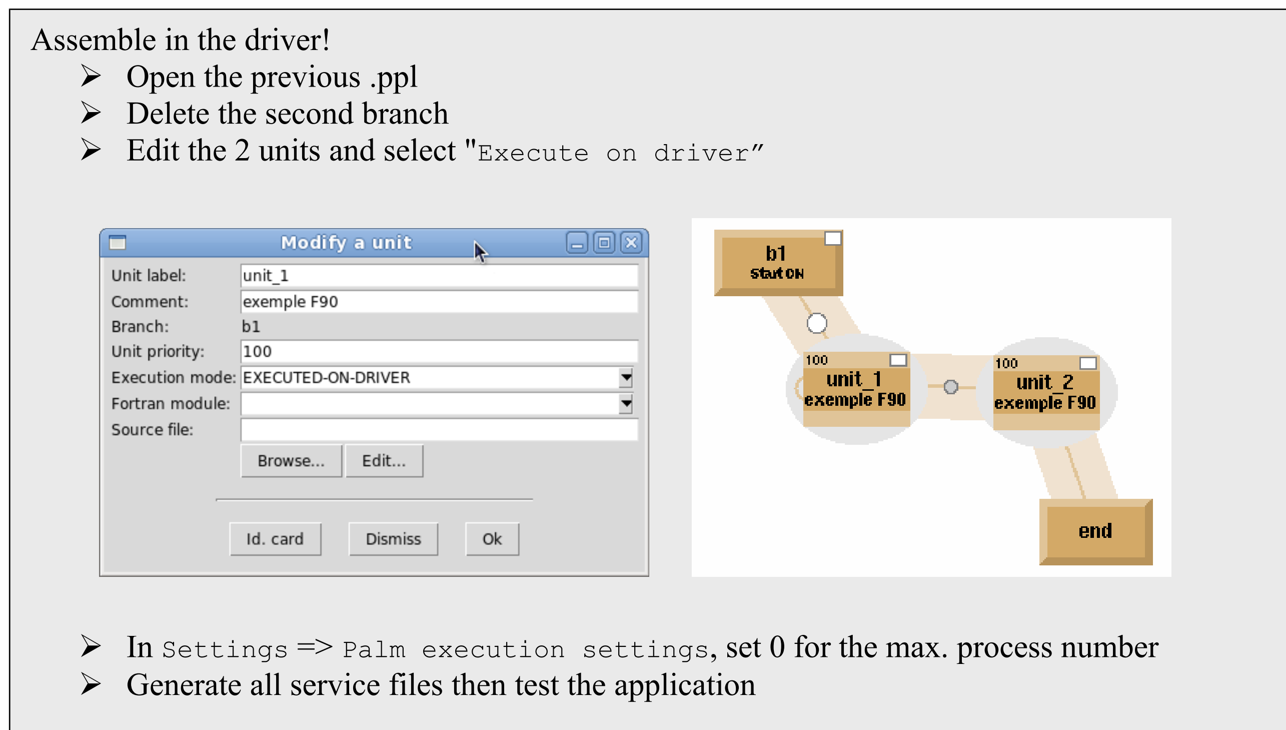
The choice of using or not the blocks results from a trade off between two optimisations: computing time and memory size. Luckily, the graphical user interface is flexible enough to test easily several configurations. One can see the advantage of having defined the units as subroutines and not as programs. This leaves all the flexibility to encapsulate the code in blocks at the the graphical user interface level.

Remark:
Some computer codes (and they are many, accordingly to our past experience) have real difficulties to run in a loop inside a block, simply because they were not designed for this usage: files are not

closed, variables are not deallocated, initialisations are carried out only once... In this case, it is simpler to avoid using the blocks, and this does not prevent us from executing such a code in a loop.

3.2 Launching the units inside the driver executable

To allow even more flexibility and to possibly avoid any waste of resources, it is possible to execute the units not as independent processes but directly as subroutines of the PALM driver program (`palm_main`). The PALM driver being mono-processor, only non-parallel units can be executed in such a way.



Assemble in the driver!

- Open the previous .ppl
- Delete the second branch
- Edit the 2 units and select "`Execute on driver`"
- In `Settings` => `Palm execution settings`, set 0 for the max. process number
- Generate all service files then test the application

Notice that the units which are executed in the PALM driver appear surrounded by a gray oval in the PrePALM canvas. The execution results (writings in file `PL_OUT`) are now available in the file `palmdriver.log`. Notice that the application behaves as if the units had been assembled in a block because in our case they all fit in the same executable (`palm_main`).

The possibility of running the units in the PALM driver is very interesting if you do not have parallel units (or a parallel machine). You can for example build a mono-processor PALM application, without making parallel computing. You keep all the PALM flexibility and modularity with the possibility to describe several computing branches, or assembling units in different programming languages, etc. As long as your units do not communicate (we will see further how to exchange data between units) it is not possible to deadlock the application.

This functionality is also very interesting for parallel applications. It can prevent the need of an additional process. But it can sometimes lead to deadlocks if the application is not correctly synchronized. Indeed, the PALM driver provides two essential functions: launching the units according to the coupling algorithm described in the graphical user interface, and answering the requests issued by the units during the run (mainly for the communications). If the PALM driver is

busy running a unit, it is temporarily unable to provide these two services. Therefore launching a unit in the PALM driver may have heavy consequences on its behaviour.

If you do not have a parallel computer, or if your application does not need a parallel computing, it is also possible to install a single processor PALM version in which all units are executed in the PALM driver. In this case you have to specify `--without-mpi` during the configuration phase of the PALM installation.

3.3 Passing arguments to the executables started by PALM

The independent executables launched by PALM have been built and compiled starting from functions (or subroutines) defined by the user and with no arguments. Their name is `main_XXX` or `main_block_NN` accordingly to their belonging to a block. By default PALM uses the MPI-2 Spawn function to launch the programs.
If your PALM unit comes from a code which requires input arguments and if for some reason you want to preserve this feature, PALM allows to pass arguments to the unit or block. For the sake of flexibility, the arguments are not entered in the graphic user interface but are passed via an input file.
If PALM finds the file **`main_XXX.args`** in the working directory when launching the `main_XXX` executable, it will use the arguments listed in this file.
In the `session3/arguments` directory, you'll find examples for this mechanism in C, C++, F77 and F90. The case of a block is also treated: in this case the arguments are passed to all the units of the block.
In the examples the arguments files are created via Fortran or command line (sh) functions inside the PALM branches, but you can imagine any other mechanism.
In the units written in C one should define the input arguments with the standard `(int argc, char **argv)` syntax. In Fortran, the use of the arguments is not standard. The common workarounds use extensions like `iargc()` that returns the number of arguments and `getarg(n, arg)` that returns the string `arg` containing the nth argument.

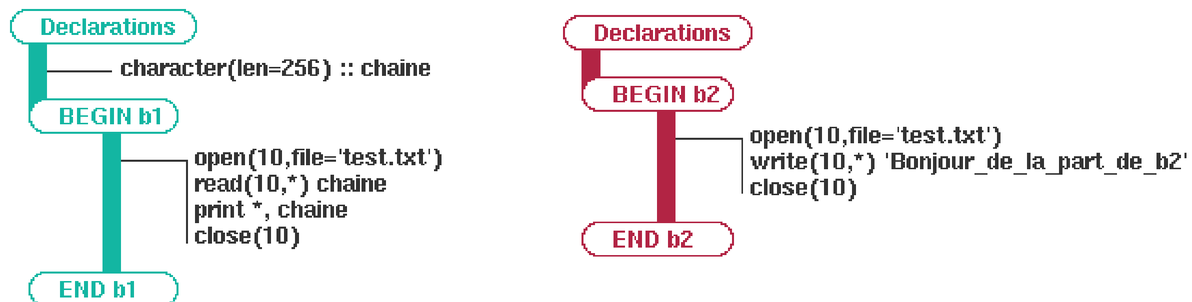
3.4 Summary of the main concepts:

In this session you have got to know the third PALM entity, the *block*. To illustrate this feature that can be seen as an optimisation issue, you had the opportunity to learn how to define PrePALM *constants* and how to access their values from inside a used defined unit.
Another practical issue you have gone through is the *Merge* ppl file command that lets you import part of another coupling algorithm in the current one.
After that you have seen how to use the PALM *driver* process to execute some mono-processor units and in which cases it could be useful.

4 Session 4: More about branches and units

4.1 Launching by another branch

Until now, all the branches we have defined, started automatically at the beginning of the application because they had the "*start on*" attribute. It is possible to delay their execution by making them to be started by other branches. In the example below, the application crashes because when the branch b1 has to read some data in the file `test.txt`, this file has not been created yet by the branch b2, although this is its task. There is a problem of synchronization in this application.



> **Exercise 3:**
> Open the file `ne_marche_pas.ppl` ("does_not_work.ppl") in the directory `session_4`
> Check that it does not work!
> Set the branch b1 to "`start off`"
> Launch b1 from b2 to make the application run correctly
> Save with another name and test it.

4.2 The steps

Another way of synchronizing an application consists in using barriers on the branches. In parallel computing, the barriers are synchronization of the processes: one can see them as a meeting point where every program waits until the other programs have reached the barrier. These barriers are associated to the PrePALM *steps*. To use a step it is necessary to create it in the graphical user interface. A step may or may not have the *BARRIER* attribute. If it has this attribute, the application will be synchronized on every call to this step (all processes will wait until the last one reaches this point).

> **Exercise 4:**
> Open the file `ne_marche_pas.ppl` in the directory session_4
> Create a step: select the step category, then the button "insert"
> Set the step value to: `PL_BARRIER_ON`
> In both branch codes, call the step (`Insert step`) in order to synchronize the end of the writing of the file `test.txt`, with the beginning of the reading of this file.
> Save with another name and test.

4.3 The scripts

Let us imagine an application which launches a computer code in a loop with several data files, for example to make a parametric study. Quite often, computer codes read their data in a file having a given name. In our case this file will be called `don.in`. To run the computer code on different cases, we need to copy, before launching, the appropriate data file to `don.in`. Here our data files will be called `fic1.in`, `fic2.in`, `fic3.in`, ... The copy of one of these files can be done by a simple UNIX command like `cp -f fic3.in don.in`. In the graphical user interface, it is possible to launch UNIX commands or to launch scripts to make this kind of operations. It is even possible, in these commands, to refer to the branch code variables.

Write!

- Still in the session_4, start from scratch (`File` => `New file`)
- Read the ID card in the file `code.f90`
- Insert a DO-loop: the index `ib_do` goes from1 to 5
- Launch a unit code inside the DO-loop
- Before the launch, insert this script:

- Test your application

The result should be like that:

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

Exercise 5:
Try to make the DO-loop run in a block
What is happening?
Change the code in `code.f90` to make it work correctly

4.4 Launching a MPI parallel unit

For those who are not familiar with parallel computing, and to make it simple, this type of parallelism (MPI), consists in dividing a problem into a number of processes. Each process is an instance of the same executable program which is duplicated at launch time. By a call to a MPI library function, each process knows its rank in the pool of the processes as well as the size of the pool, and thus, can differentiate itself to perform a part of the calculation. Once launched, each process runs independently from others, and manages its own data (variables, arrays). MPI, which is a standard, provides both the execution environment of the processes and the way of making them exchange data by relying on primitives specifically designed for the communications. This

type of parallelism is particularly effective (when it is well coded), and it allows to take the best advantage of the distributed memory machines where each process is attached to a processor.

PALM handles two levels of parallelism. The first one, we have already seen, is a task parallelism managed by the branches. The second one is an internal parallelism within the units. Every parallel program can become a parallel PALM unit. To illustrate the launching of parallel units in PALM we will start from an example of parallel code and transform it into a PALM unit.

This example is one of those given in the LAM MPI installation. It calculates π with several processes.

Palm it!

- Copy the file `fpi.f` to `pi_mpi.f` and edit it
- Add the following ID card:

```
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 {pi calculation}
```

- Change `program main` to `subroutine pi_mpi()`
- Add the inclusion of the file `palmlib.h`
- Comment out the following lines:
  - the `mpi_init` call
  - the `mpi_finalize` call
  - the `stop` instruction
- Everywhere in the program change `MPI_COMM_WORLD` by `PL_COMM_EXEC`
- Save your file
- Start from a scratch PrePALM file and insert the unit pi_mpi
- Set the unit's number of processes to a number between 2 and 64
- In “Palm execution settings” set an appropriate max. number of processes
- Test your application

Every new PALM user can take this small example as a starting point to adapt any computer code to PALM. The writing of the ID card does not pose a particular problem. If the program is split in several files and has its own Makefile, it may be better to keep this Makefile. But instead of creating an executable, we may rather create a library (`.a` file), which will be referred to in the object_files field of the ID card.

The search of the main program (`main` in C or `PROGRAM` in FORTRAN) and its replacement by a function or subroutine name is straightforward. The `mpi_init` and `mpi_finalize` calls, should be eliminated. They are usually invoked only once by the main program, respectively at the beginning and at the end of it. These calls are thus very easy to locate. The replacement of the MPI communicator `MPI_COMM_WORLD` by `PL_COMM_EXEC` can be more problematic since it may be present in many files, and a single omission can be catastrophic. It is then much safer, and recommended, to create a script in order to do it automatically.

4.5 Launching an OpenMP parallel unit

For those who are not familiar with, in very simple words, OpenMP allows to parallelize codes with an approach very different from MPI. Here, only one executable is launched. Using compiler level directives (and by activating the right option for it), the user informs the compiler on the source code areas where a parallel computing is possible: for example at the beginning of a loop involving some array calculations. During the execution, the program launches "light" tasks (called threads) to make several processors work together on the same code, but on different portions of the global shared data. This type of parallelism is often easier to implement than MPI, and is less intrusive in the code, but it does not allow to increase the size of the problem beyond the memory size of a single cluster node nor to manage efficiently a large number of processors. However, it is especially adapted for shared memory parallel machines.

With PALM, it is possible to launch programs parallelized with OpenMP. For this, we just have to define the type of parallelism in the unit ID card, and to compile the unit with the right option (`-mp` in our case). We have also to indicate in PrePALM the number of processes dedicated to the unit.

Exercise 6:
Run the unit `prodmv omp.f90` on 4 processes.

Remark:
Depending on the machine we are using, the program may deadlock. In this case it is necessary to modify the stack size with the command:

```
> limit stacksize unlimited
```

The number of required processors can be greater than the number of physical processors on the machine. In this case OpenMP issues a warning, which does not prevent the application from running. You should obtain a result like that:

```
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
```

If you need to run a parallel OpenMP computation you can take this example as a starting point. The method is always the same: look for the entry point of the computer code (`main` in C or `PROGRAM` in FORTRAN), replace it by a subroutine name, define the ID card, and create a library rather than an executable.

Before we try to establish communications between PALM units (with PALM specific primitives which we will see in the following chapter), we have first to be sure that the application runs correctly in the PALM environment.

4.6 Summary of the main concepts

In this session you have learnt how a branch can launch other branches and how you can synchronise the branches with a barrier-type event (step).
Then you have seen how to adjust a parallel code, either MPI or Open-MP based to build a PALM unit.

5 Session 5: Communications

5.1 Introduction

Until now, we have seen how PALM manages the processes in many different ways. In order to create a genuine coupled application, it is time to let the units talk together.

Either for a full code or for a very simple routine, you first have to define what needs to be exchanged, what are the useful coupling data to be identified in a coupled application? Only the user or user group are able to answer this question. The coupling of an ocean model with an atmosphere model, for example, will imply the exchange of the temperature fields through the interface between the two models (e.g. the Sea Surface Temperature). On the other hand, in the coupling between a fluid dynamics model and a meshing tool the entities to be exchanged will be the constraints on a structure and the meshes. In the real computer code, these physical quantities or these fields are stored in variables, which have a type (integer, real, structure...) and a size (1d, 2d... arrays). In order to be a generic tool, PALM does not give any constraint concerning the nature, the type or the format of the data to be exchanged.

In PALM, we call "*object*" the data to be exchanged, and "*space*" the computer representation of the objects. Several objects can share the same space. In the graphical user interface and in the units, the objects and the spaces will be characterized by a name.

In order to have a maximum of flexibility in building PALM applications, the exchange of information will be made in two steps, thus allowing a total independence between the units.

A unit (the source code) will never tell explicitly to which destination an object will be sent to. However the unit has to "publish", at a specific place in the program, the fact that some data has been computed and is ready to be sent. This action will be done by inserting in the source code a call to the **`PALM_Put`** primitive.
A unit will not tell either from which specific source it must receive its data, but simply it will pass the information that it needs some data and in which variable(s) the data must be received. For that it will be necessary to insert in the unit source code a call to the **`PALM_Get`** primitive.

The "true" link between the Puts and the Gets is done quite simply in the graphical user interface by connecting two plugs representing graphically the PALM_Put and PALM_Get invoked in the code. These calls must be first listed in the unit ID card.

In most of code couplings, we deal with time evolving processes that translate into iterative time stepping procedures. The objects exchanged in this context reflect for example the temporal instances of a physical evolving quantity. In PALM, it is possible to differentiate in time two instances of the same object. When calling the Put/Get primitives it is necessary to specify a field "*time*" containing the time value the object is associated to. For PALM, this time field is simply an integer variable. The user is free to associate this integer to a physical date (*cf.* Chapter 21.2 for the *date to/from integer* conversion utility) or to neglect this attribute by using a predefined constant `PL_NO_TIME` if the object is not time dependent.

5.2 Preparation of the units, the PALM primitives

This will be easier to understand if we consider an example. In the directory `session_5`, open the file `producteur.f90` (meaning producer.f90). This unit produces a square matrix of size IP_SIZE*IP_SIZE and a vector of size IP_SIZE. Let's look in detail at its ID card:

```
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: In addition to the key word `PALM_UNIT` you already know, you can see new keywords: `PALM_SPACE` (6,11) and `PALM_OBJECT` (16,21,26).

6: the first `PALM_SPACE` allows us to define a 2-dimensional array by the field shape (7). These two dimensions are described with a parameter (`IP_SIZE`). N.B. It is perfectly possible, to hardwire values, but using the `IP_SIZE` constant which has to be defined in the graphical user interface PrePALM constant editor make it easier to describe different configurations without having to edit and reload the identity cards. One then defines the size of each array element. As this size may depend on the PALM library compilation options, it is given with specific PALM keywords. This size will have the right value in accordance with the linked PALM library: for example the automatic promotion, or not, of single precision reals to double precision depends on the compilation options (usually `-r4` or `-r8`).

11: the second `PALM_SPACE` allows us to define a 1-dimensional vector. Note the parenthesis in the "-shape" definition, you should never forget them.

16: the first `PALM_OBJECT` allows us to define an input (`intent IN`) object. It refers to a space which has not been explicitly defined (`one_integer`). Several spaces are pre-defined, like `one_integer`, `one_real`, `one_double`, `one_complex`, `one_string` (256 characters string) and

one_logical. You should keep in mind that these pre-defined objects exist in order to simplify the ID cards description.

The other objects (21, 26) are output objects (intent OUT). They correspond respectively to a vector and a matrix which will be produced by the unit. You may notice that for the vector, it has been specified that the time field (time ON) will be used. The PALM_Put primitive call will thus be done with a time that is not equal to PL_NO_TIME, and PALM will be able to manage independently the different instances of this object.

Let's take a look on the source code of this unit. In our example, all PALM primitives calls are made in the unit's subroutine but nothing prevents us from using these primitives in lower level subroutines. Notice that the order in which the Put/Get primitives are called is of no importance for a correct coupling operation. However, it has an impact on the performances of the 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: The first thing to notice is the use of the module palm_user_param. This module is created by PrePALM. It allows the units to access to the parameters set in the graphical user interface. In our case it is then possible to use the constant IP_SIZE (the same constant as the one used in the space definition) in the dimensions of the arrays (42). This will enable us to easily change the vector and the matrix sizes without modifying the units source code, just by re-compiling the application.

40: The strings cl_object and cl_space will contain the objects and spaces names. They are declared with length PL_LNAME (length pre-defined for this kind of strings in PALM). This constant, like many others, is declared in the palmlib module (PALM library): 35. In the Fortran source code we have to issue a USE of this module

54: A call to the PALM_Put primitive is needed to send the matrix. It may be tempting to write directly `CALL PALM_Put(' mat2d', ' matrice', PL_NO_TIME, PL_NO_TAG, dla_mat, il_err)` with the space and object names as arguments. But FORTRAN compilers have no standard way to manage the strings in subroutine arguments. It is then preferable to use the intermediate variables cl_space and cl_object which were defined with a specific length for PALM names (PL_LNAME).

61: A call to PALM_Get is made in order to know at which the time the vector object needs to be sent. This allows us to illustrate how to manage objects using the time attribute.

63 - 66: All PALM primitives return an error code. If the error code is different from 0 it means that a problem occurred and that the variable returned by PALM_Get is not correct. Our unit is made for producing a vector object at the time returned by the PALM_Get. If it does not have this time, it cannot work. Therefore the application needs to be stopped by a call to PALM_Abort. Let's notice that this is the only way to properly stop a PALM application: the commands STOP, EXIT and CALL MPI_Abort must be banished because the PALM driver is not informed.

76: The vector is sent at the time il_vect_time.

Exercise 7:
Open the file vecteur_print.f90 in the directory session_5. The ID card is not yet complete. Before completing it, answer the following questions looking at the FORTRAN code.
Questions:

- What is the program instruction which allows us to dimension the array dla_vect to IP_SIZE?
- How many IN objects are there?
- How many OUT objects?
- How many spaces are used?
- How many spaces have to be declared in the ID card?
- What happens if the time is never received?

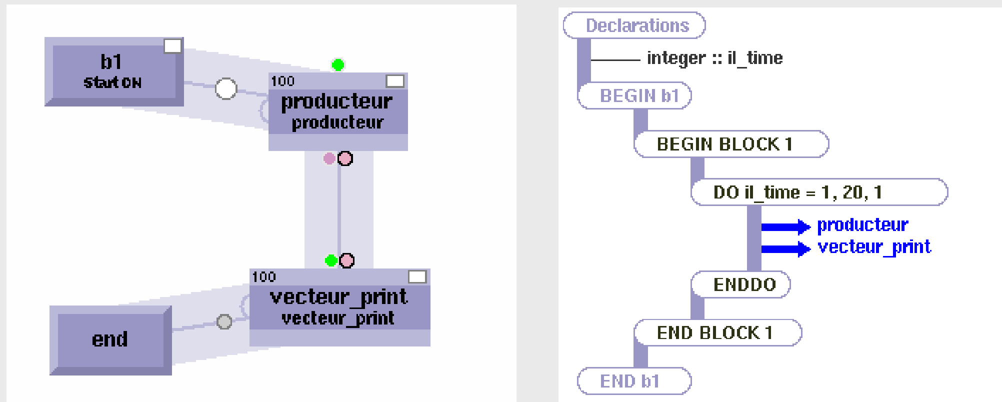
Complete the ID card and save the file!

5.3 The communications in PrePALM

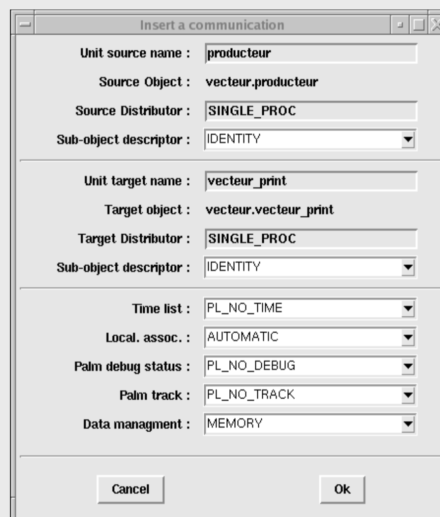
Now, we will make our units work together

Communicate!

- Start from scratch in the directory `session_5` (new PrePALM file)
- Add a constant `IP_SIZE` with value 1000 (vector and matrix size)
- Load the ID card of `producteur.f90` and `vecteur_print.f90`
- Insert a branch `b1` (`IP_START_ON`)
- Edit the branch code. Launch first `producteur` and then `vecteur_print` in this order
- Add a DO-loop around the 2 units (**`il_time`** as index varying from 1 to 20) and a block around the DO-loop. Close the branch.



- Without clicking, move the mouse cursor on the plugs (small colored circles on the units). Examine the pop up window and also the help message at the bottom of the PrePalm main window.
- Click on the plug corresponding to the production of the `producteur` vector. It becomes red. Do the same with the plug of `vecteur_print` corresponding to the PALM_Get.
- You should have the following dialog box:



Insert a communication

Unit source name : producteur
Source Object : vecteur.producteur
Source Distributor : SINGLE_PROC
Sub-object descriptor : IDENTITY

Unit target name : vecteur_print
Target object : vecteur.vecteur_print
Target Distributor : SINGLE_PROC
Sub-object descriptor : IDENTITY

Time list : PL_NO_TIME
Local. assoc. : AUTOMATIC
Palm debug status : PL_NO_DEBUG
Palm track : PL_NO_TRACK
Data managment : MEMORY

Cancel Ok

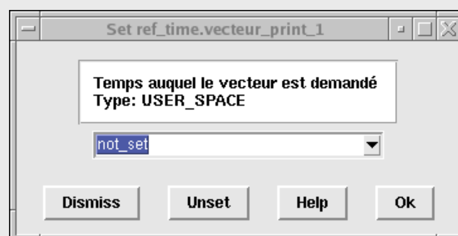
- In the field `Time list` instead of `PL_NO_TIME`, enter `1:20`

You have just defined your first communication between two PALM units. By initialising the field `Time list` with 1:20, you authorize the vector (if it is created) to be sent and received at these times. We now have to tell the unit "producteur" to produce the vector at these 20 times, and the "vecteur_print" unit to request the vector at all of these times.

We don't have a unit producing objects for a set of different times at once, and it would not be a good idea to create one just for this purpose. Telling the unit to produce an object at a specific time more than once is typically an instruction of a **coupling algorithm**. It is then the graphical user interface which will provide a method for doing that. The two units are launched in a time loop, with a loop index varying from 1 to 20. It is of course this variable which can be used to force the two units to work at these times. There are two solutions to do that, either issue a `PALM_Put` in the branch code, or directly hardwire the value in the consumer unit.

Tell what to do!

- Open the branch code
- In the DO-loop, before "producteur" insert a call to a `PALM_Put` with an integer (insert `PUT … => one_integer`)
- Name of the variable: `il_time`
- Close the branch code. A plug appears on the branch line in the canvas
- Create a communication between this object and the "producteur" input (click on both plugs)
- You could do the same for "vecteur_print" but we will do it differently. Right click on the "vecteur_print" `ref_time` input plug.
- The following dialog box should appear:



- Instead of `not_set` select the branch variable `il_time`, and then validate
- Your application is ready and you can test it

5.4 Time lists

In our example, we set 1:20 in the **time list**. The syntax of the **time list** field is as follows:

start1[**:end1**[:**step1**]] [| **start2** [**:end2**[:**step2**]]] [; …]

The expressions between [ ] are optional.

The character pipe **|** makes it possible to describe different times for the source and for the target.

The character ";" is used to separate two expressions.

The character ":" is used for the loop ranges.

Examples:

Expression	Source	Target
18 ; 33 : 34	18 33 34	18 33 34
20 : 30 : 5	20	20

	25 30	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 because the loops on the two sides of | have a different number of elements	

It is possible, for each elementary field, to use PrePALM constants and arithmetic expressions. The branch **variables** are **not allowed**, because the definition must remain static, but the constants are authorized. If source and target times are different (use of |), one can define the source time as a function of the source or target time (noted **i),** the time instance sequence number (noted **o**), and the total number of times (noted **nb**), and conversely.

Examples:

Expression	Source	Target
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

Combined together, these notations allow us to describe any type of association between target times and source times. This is a provision, in real applications, different times between source and target are used rarely.

Description of the field `tag`

If one of the two plugs has its **tag** field activated (`-tag ON`), the dialog window requires to fill the field **`tag list`**. You have to describe here all the tags for which the communication has to be performed. The syntax of the field **tag list** is strictly identical to the syntax used for the field **time list**. There is no example on the use of tags in this user guide for the sake of simplicity, but you might appreciate using them in your own applications.

Combination of fields time and tag

The times and tags fields described here can be combined between them by a Cartesian product. For example if one enters 10:12 for the time field and 7 | 107; 4 | 44 for the tag field, the authorized communications will be:

Source		Target	
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 Hardwired values

In the previous example, the branch loop variable (`il_time`) was used to set the time at which `vecteur_print` must work. In this field, PrePALM accepts any type of valid Fortran90 expression. This ability to use directly an expression for an input plug is not restricted to scalar variables. Any `PALM_Get` can be initialised this way. This functionality can be very interesting to perform unitary tests on the units. The only constraint is that the expression used to initialise the variable must be written with a single Fortran90 instruction.

Exercise 8: vecteur_print unit test
Start from scratch
Load the `vecteur_print` unit
Define the vector size to be 50 in the PrePALM constants
Launch the unit in a DO-loop with `ib_do` going from 10 to 100 with a stride of 10
Hardwire the plug `ref_time` with the DO-loop variable
Declare an integer variable `i` in the branch
Initialize (hardwire with a right click) the input vector with the following Fortran90 expression
`(/(i,i=1,IP_SIZE)/)*ib_do`

Remark: do not insert blanks in your expression; PrePALM does not accept them.

5.6 The NULL space and space inheritance

In this session, the two units `producteur` and `vecteur_print` exchange a 1d vector: this is possible because the computer representation of the object (type and size of the spaces) are the same. The compatibility is ensured in the identity cards, where both spaces are declared being of double precision real type and of the same size `IP_SIZE`, the latter being a user defined constant whose value is set in the graphic user interface. The compatibility check is performed by the graphic user interface when the user draw a communication between the two corresponding plugs. The check acts on the type and the the size of the spaces, not on their names that can differ in the two units.

Sharing a PrePALM constant name in two identity cards is a safe way to enforce compatibility, but it breaks the full independence of the units.
Another solution would be to use two different constant names in the two units, let's say `IP_SIZE_P` and `IP_SIZE_V` and then to assign `IP_SIZE_P =IP_SIZE` and `IP_SIZE_V=IP_SIZE` in the PrePALM `Constant` menu. Nevertheless this solution also has a drawback: it makes impossible to change the size of the handled objects between different (or successive) instances of the units.

To make a generic purpose unit as `vecteur_print`, really generic, the solution is to assign the input object to the `NULL` space. The `NULL` keyword just indicates to PALM that the space features of the object will be inherited from the other communication end-point when drawing the communication. Notice that when inserting the unit on the canvas, the plugs of objects belonging to the `NULL` space are yellow, then they inherit the space colour as well when the communication is drawn.
This approach has some consequences on the way `vecteur_print` has to be programmed. The array used to receive the input object has to be dynamically allocated to match the shape and the size of the received object. To do that PALM provides two primitives:

- **`PALM_Object_get_spacename`** to ask PALM the name of the inherited space
- **`PALM_Space_get_shape`** to ask PALM the shape of the ibnherited space

Once we have obtained this information we can allocate (or reallocate) the local array before the corresponding call to `PALM_Get` or `PALM_Put`. Notice that, to be consistent with the identity card, the space name in the communication primitives has to be set to `'NULL'`.

In the directory of session 9 you'll find a version of `vecteur_print.f90` using this feature. In session 9 you'll learn as well how to manipulate dynamic size spaces, where the shape of some spaces is known or computed at run-time. For the sake of simplicity, for the moment we'll let one side of the communication statically define the space.

Finally notice that if you delete the communication, the inherited space keeps the colour of the last association. Before associating the plug to another space, you have to click once on the plug, then in the `Object` pane on the left you double click on the highlighted line and you access a pop-up menu with the object properties. In the `Space` field open the scroll down menu and choose `NULL`.

5.7 Communications attributes

Let's take the time to go through all the possible attributes you can set in a communication box. Not all the possible options will be used during the training sessions but it is mandatory to know all of them if debugging or optimising a coupled application.

The box is split in three sections. The first one recalls the features of the source object, namely:

- the source unit name
- the source object name, suffixed by the unit name, to grant uniqueness in case of multiple instances of the same unit or of objects with the same name in different units
- the way the source object is stored on the source unit, what we call a *distributor*. This information will become meaningful when dealing with parallel units and parallel communications in session 11. In this session the distributor is `SINGLE_PROC` because the unit is not parallel and therefore the array containing the whole object is entirely stored on the only processor of the unit
- the subset of the local object that is really used for the communication. Most of the time a communication exchanges a full object, coincident with the full local storage. Some particular applications (think of exchanging only the 2D surface layer of a 3D model) only need to access a subset of the local object. This field contains the name of the description of the mapping of the subobject in the local object. `IDENTITY` means that the whole object is exchanged. Session 12 is entirely devoted to the use of subobjects.

The second section of the box is identical to the first one but describes the target object.

In the third section we find:

- the `Time list` field that we have previously described in this session. This field is present only if at least one of the objects at the two sides of the communications has the `time ON` attribute in its unit identity card
- the Tag `list` field, if one of the two objects has the `tag ON` attribute
- the `Local. assoc.` field that we'll describe in session 11 dedicated to parallel communciations
- the `Palm debug` status field that triggers a user defined check function acting on the exchanged object. A template of this function is found in `palm_debug.f90` or `palm_debug.c`: comments in these files are quite exhaustive. The field can be set to `PL_NO_DEBUG`, `PL_DEBUG_ON_SEND`, `PL_DEBUG_ON_RECV` or `PL_DEBUG_ON_BOTH` accordingly to the choice of not executing the check function, executing it on the source side, inside the `PALM_Put` call, on the target side inside the `PALM_Get` call or on both sides. The debug function can be used to print the content of communications without modification to the units.
- the `Palm track` field that, if a high enough verbosity level is set for communications, tracks the execution of the different steps of the communication: notification of the Put/Get to the driver, routing, etc... These information is written in the PALM log files (`palmdriver.log`, branch`_XXX.log`)
- the `Data Management` field defaulting to `MEMORY`. If the user chooses `DISK` instead, the pending objects, waiting to be delivered are written on disk instead of being kept in the PALM own memory buffer. This option can sensibly slow down the application and it has to be used only in case of RAM memory size problems, after having tried every other solutions (like synchronisation).
- the `Optimisation` field is a specific field to optimse performances under particular conditions, because it imposes some extra constraints on the order of the communications and can easiliy lead to deadlocks. The `PL_OPTIM` option implies the use of blocking communications on the source side (`PALM_Put`). This option is useful for massively parallel codes. With the more flexible `PL_NO_OPTIM` option all the processors have to interact with the driver to know the routing table of the exchanged data and, with a large number of concurrent parallel tasks it can turn out to be a bottleneck. With `PL_OPTIM`, the routing table is static and computed only once, and the communications bypass the driver

`PL_NO_OPTIM`	`PL_OPTIM`
Non blocking PALM_Put, the source unit can continue even if the corresponding PALM_Get has not been posted yet. In such a case PALM bufferises the pending object.	Blocking PALM_Put: the source unit waits for the corresponding target PALM_Get to realise a direct communication.
Dynamic routing of the communications. Exchanges with the driver.	Static routing. No need to communicate with the driver.
Flexible and generic.	Better performances for massively parallel units

5.8 Summary of the main concepts

In this important session you have seen in practice how the PALM one-sided communication model works. You have got familiar with the concepts of *space* and of *object* and you have seen how to describe them in the identity card. By the way you have noticed that the spaces for scalar quantities are predefined in PALM (`one_integer`, `one_real`, etc).

You have then seen how to describe the effective communications in the PrePALM interface and you have therefore learnt how to describe the time lists of allowed communications and how to associate the on the source and target side if they do not coincide. This point, that usually requires some time to become familiar, is of extreme importance. The concept of *tag* is very similar to the *time stamp* one.
You have seen how to hardwire a value to fill a get request with a constant value, a PALM variable or a FORTRAN90 like construct.
You have also had the opportunity to pinpoint that the `PALM_Abort` primitive is the only really clean way to shut-down a coupled application in case of error.

Remark: At this point you have seen the main features of the PALM coupler. You should already be able to create a coupled application with independent units exchanging information. In the following sessions you will see more advanced functions which may not be necessary for your work, but which can prove useful in developing a complex application.

6 Session 6: Predefined units

6.1 Introduction

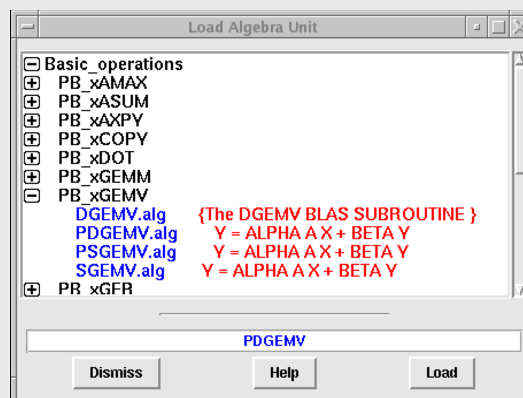
When you are building a coupled application, you may need to perform linear algebra operations on PALM objects before exchanging them between two units. For example a matrix vector product. These "generic" operations, from the simplest to the most complex, are directly available in the graphical user interface PrePALM. When they exist, you are even strongly encouraged to use these functions rather than to develop them by yourself, since they are calling the mathematical libraries tested and optimised on your computer.

The predefined units (or algebra units) have the same operating mode as the user-defined units. The only difference concerns the time and tag attributes management of the objects sent to, or received from these units. Indeed, these units must be able to receive the objects at the times defined by the user in his application. For each received and/or sent object, it will thus be necessary to specify for which time and which tag the `PALM_Get` and/or `PALM_Put` must be performed in the algebra unit.

To illustrate the use of algebra units, we will make the product between the matrix and the vector built by the "producteur" unit (the same unit as in session 5)

Load the predefined units!

- Open a new PrePALM in the directory `session_6`
- In a first "START_ON" branch launch "producteur" then "vecteur_print"
- Make both units work at the time10 by hardwiring the proper input plugs
- Load the algebra unit `DGEMV`: menu `File` => `Load Algebra unit`

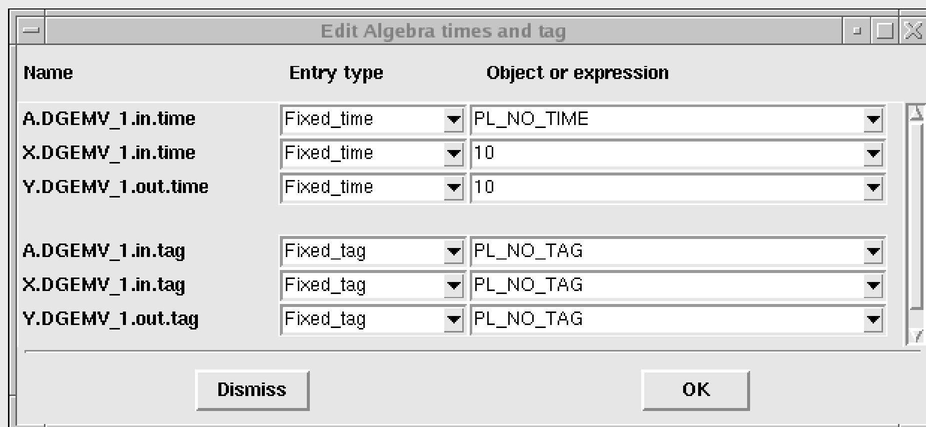


- Open `Basic_operations` => `PB_xGEMV`
- Select `DGEMV` then click on `Load`

After loading the unit, PrePALM pops up a help window concerning this routine. Check that it corresponds to what you want to do and close the window.

Use the predefined unit!

- Create a second “START_ON” branch
- Launch the algebra unit in this branch, close the branch code
- Note the 3 yellow plugs on the input side and the yellow one on the output. They indicate that the space associated to these objects is not yet defined (NULL space)
- Create a `PL_NO_TIME` communication between the producer matrix and the object `A` of DGEMV. The dotted line indicates that time is `PL_NO_TIME`
- Send the producer vector to the object `X` of DGEMV. Set 10 for the time list.
- Send the output `Y` of DGEMV to vecteur_print at the time 10.
- Hardwire the value 0 (right click) for the input `Y` of DGEMV.
- Now click on the DGEMV unit left rectangle (time & tag receiver)



- This dialog box allows to manage the times of the algebra unit objects. The matrix is produced with `PL_NO_TIME` by the producer (unit “producteur”). Thus, it must be received with `PL_NO_TIME` in DGEMV. On the other hand the object `X` is produced at time 10. Then, it must be required at this same time in DGEMV.
- Set 10 for `X.DGEMV_1.in.time` and `Y.DGEMV_1.out.time`
- Test your application.

Remark:
For our example, we have imposed the time of the object `X` and `Y` to be `Fixed_time`.

This time may also be received: in field `Entry type`, select `Received_time`. In this case, a plug will appear in the rectangle time & tag receiver and a communication for this plug has to be defined.

Moreover, the time can be calculated from other received times: in field `Entry type`, select `Calculated_time`. In this case it is necessary to enter an expression which can use other times (received or calculated): to use other times, we have to give the full name (left-hand column) preceded by the $ sign. If you run this application in a DO-loop, you may for example receive the loop index as time for `X.DGEMV_1.in.time` and then calculate the time for `Y.DGEMV_1.out.time` with the expression: `$X.DGEMV_1.in.time`

Exercise 9:
Start from the previous application
Change the vector and matrix sizes in the PrePALM constants: reduce it from 1000 to 10 (for a better look of the results)
Execute the units “producteur” and “vecteur_print” in a block, inside a DO-loop (from 1 to 10) and produce the vector at the different times

The goal is to multiply the vector by the matrix only for the even times (2, 4, 6, 8 and 10). For other times the vector produced by producer must be directly printed by vecteur_print (without using DGEMV)

Hint:
You can keep the second branch at START_ON and launch DGEMV inside a loop.
The switch between the even and odd time values for the two units will be done by selecting the objects in the communications field: Time list.
Use the received and calculated times together with the Put of branch variables, for the management of the times of the objects sent to DGEMV.

Another solution is to position the branch b2 to START_OFF and start it in the branch b1

6.2 Summary of the main concepts

PALM provides a toolbox of algebraic operations. They are made available as a set of predefined units. The only difference they have with user-defined units is the time stamp and tag specification mechanism.
You also had the opportunity to practice a little more the usage of the time stamps lists association.

7 Session 7: Derived data type objects

7.1 Introduction

Up to now, the exchanged objects type was always a canonical data type: integer, real or double precision. In PALM, it is possible to manage objects having a derived type which corresponds to data structures defined by the user. This possibility is interesting for example, when sending in a single message several arrays having different characteristics, or just arrays of data structures defined by the user in a unit. Whether the derived types are contiguous or not in memory, the use of the PALM primitives is more or less practical. We will examine both cases.

7.2 Memory contiguous objects

For contiguous objects, which were declared as such in the unit source program, the procedure is the same as for traditional objects. The only difference concerns the description of their space size. PALM must know (or be able to deduce) the size of the array to be handled. Two solutions are proposed to describe the size of the space of derived type objects.

The first one consists in describing the field `-element_size` in the form of an arithmetic expression using the basic types of the structure in terms of the PALM keywords identifying the basic types (`PL_INTEGER_SIZE`, `PL_REAL_SIZE`,...). Let us suppose that our derived type is as follows:

```
TYPE personne ! = person
  SEQUENCE
  CHARACTER*20 :: nom ! = surname
  CHARACTER*20 :: prenom  ! = first name
  INTEGER :: age
  REAL :: taille ! = size
END TYPE personne
```

Notice the attribute `SEQUENCE` in the Fortran90 definition of this derived type. It forces the compiler to keep the four fields contiguous in memory. In the ID card, in the space definition, the elements size can be described in the following way:

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

The second solution consists in referring to a list of spaces whose size was already defined. For the same example, we must first define a space of size 20 characters (`chaine20`). Then we must define a list of items (tuples) containing : i) the item name and ii) the space associated with this item. For the same example of derived type, we may describe its space in the following way:

```
!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}
```

The second solution seems more complicated but it is compatible with objects which are not contiguous in memory. Thus, we recommend to use this second manner for the description of the derived type objects.

Type!
- Open a new PrePALM in the directory `session_7`
- Look at the unit source codes `personnes.f90` and `pers_print.f90`
- Launch these 2 units sequentially within only one branch
- Connect the two plugs
- Test the application

After execution, you should read:

```
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
```

Exercise 10:
Start from the previous application.
Create a third unit named `vieillir.f90` (= getting_older.f90)
This unit will call a `PALM_Get` for an integer corresponding to a number of years `n`, and will make older every person of a group object (to be declared in input and output) by `n` year(s).
If there is no communication described for `n` in the application, `n` will take the default value 1. For this, initialize the variable before the `PALM_Get` and test the error code or use the `PALM_Query_get` primitive (*cf.* Chapter 24).
Test your unit by inserting it between `personne` and `pers_print`.

7.3 Non contiguous objects

It is also possible to describe objects whose elements are not necessarily contiguous in memory. This possibility is very useful to handle data structures (C language) of pointers (thus dynamically allocatable arrays for which the alignment in memory cannot be guaranteed) or more simply to send in a single message several arrays having different characteristics.
To illustrate this function, we will send two arrays with different shapes and sizes in a single `PALM_Put`. The unit which produces the data is written in C. The unit which recovers them is written in Fortran90. This will also illustrate the differences between the languages when calling the PALM primitives.

Open the source code of `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: With the keyword "`-items`" both arrays will be assembled in a single object made up of two different spaces (7 and 12). The size of these arrays is parametrised in PrePALM by the use of the constants NB_ENTIERS and 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: In C, this "include" file allows us to use the constants defined in PrePALM for dimensioning the arrays.
56: In order to send both arrays with a single `PALM_Put`, they must be first copied in an intermediate variable. In C we define a void pointer which can be allocated with the size of the object declared in the ID card. The `PALM_Space_get_size` primitive returns this size.
59: The variable `ila_pos` is used to define the position of each element in the structure, here we have just one element (cf. 18: `-shape (1)`). If the shape had been of rank 2 (for example (10,25)), `ila_pos` would have been an array of 2 elements. In our example `ila_pos` is set to 0, since in C arrays indexes always start at 0.
62 and 63: The vectors are packed and copied in the variable `buffer` with the `PALM_Pack` primitive.
70: The assembled object is sent with a single `PALM_Put` call.

Let's take a closer look to the unit `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: The call to the `PALM_Space_get_size` primitive differs in C from the FORTRAN call.
46-48: The size is expressed in bytes. Since we are manipulating an integer array, with element size of 4 bytes, we have to divide the size by 4.
55: Here the position of the element is 1 because the index of FORTRAN arrays always starts at 1.

Exercise 11:
Test both units

Remark:
Derived types are quite practical, often they enhance the readability and the flexibility of the codes. However, we should not go too far in encapsulating any data in derived types which can be heavy to handle. The more your PALM units will handle derived data type objects, the less portable they will be and their interface with other units will be less general. Think for example to the linear algebra units. To be portable, they handle only simple pre-defined types. Sending all your data as derived data types, will prevent you to use the predefined algebra units, except if you insert some interface PALM units.

7.4 Summary of the main concepts

This session deals entirely with derived data types and structures. The main difference is between objects which are contiguous in memory, for which you have simply seen how to describe them in the identity cards, and generic structures for which you have to create a contiguous communication buffer. For this reason you have got to know the new *PALM_Pack* and *PALM_Unpack* primitives.

8 Session 8: Time interpolation

8.1 Introduction

When two computer codes are coupled, these programs do not necessarily use the same time step. In order to sidestep this problem we may interpolate in time the physical fields: to be able to accomplish the interpolation we have to store in memory several temporal instances of the same object. This kind of time interpolation is one of the functions of the PALM coupler: a unit can for example produce its objects every 10 seconds whereas another unit requires them every 3 seconds. In association with this time interpolation function we will see how to manage data in a permanent storage memory space called the *PALM BUFFER*. Let's recall here the difference between how PALM handles the communication amongst units and the usage of this intermediate space. In direct communications, PALM tries to optimise the memory consumption and the number of data transfers. If a `PALM_Get` is posted before the corresponding `PALM_Put`, the concerned object will be directly routed from the source to the target unit as soon as the `PALM_Put` is issued. On the contrary, the non blocking policy imposes to store in a temporary space a produced object if the corresponding `PALM_Get`'s have not all been issued yet. As soon as the last `PALM_Get` is completed, the object copy is removed from the temporary storage space. If for some reason the user needs to keep a copy of a posted object regardless of if, when and how many times the object has to be received, he can address the communication to an explicit storage space that has to be thought of as a sort of shelf from where the object can be repeatedly recovered until it is not explicitly discarded. In order to avoid any overflow of this BUFFER with data that are no more required by the application, a flexible mechanism has been designed for a detailed management of the objects stored in the BUFFER: this is the *steplang* language.

8.2 Units Preparation

We will start again from the “producteur” (= “producer”) unit which we improved so that it behaves more closely like a real model. In its inner loop it will produce a vector for different time steps, which is often the case in the computer codes. To be more flexible, this unit will ask for the indices of start, end, and stride of the inner loop.

Send your objects to the BUFFER!

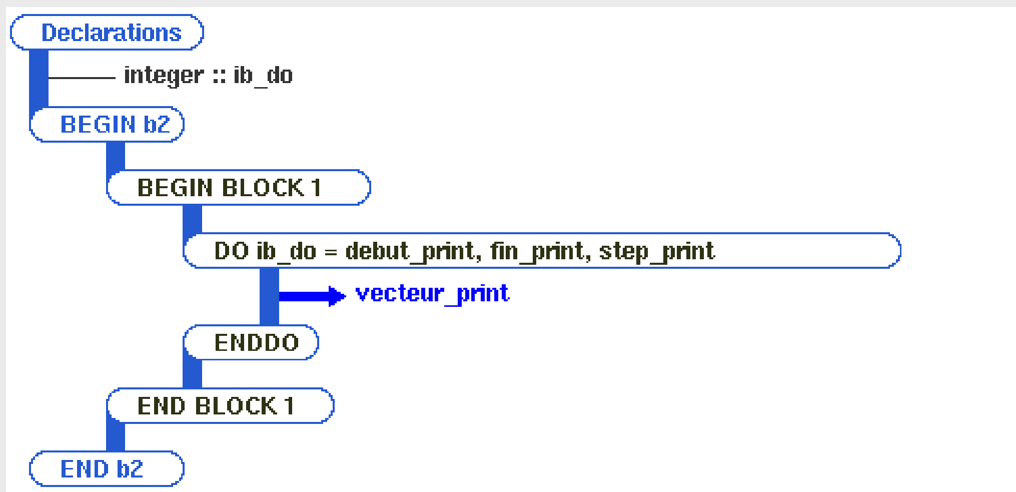
- Open a new PrePALM canvas in the directory `session_8`
- Define four constants: `IP_SIZE` = 100000 (vector size), `debut_prod` = 0, `fin_prod` = 1000 and `step_prod` = 10 (= start, end and frequency stride of the inner loop)
- Insert a branch `b1` which launches the producer. For the producer input, in order to control the produced times, hardwire the values by selecting the previously defined constants.
- Send the vector to the PALM BUFFER. To do so, double-click on the vector output plug. The dialogue box asks you an object name for the copy of vector stored in the BUFFER. For the field “time list” put: **`debut_prod:fin_prod:step_prod`**. then validate.
- On the canvas, you should see a communication which is plugged in a small square: the PALM BUFFER symbol.

You have just authorized all vector time instances to be stored in the permanent memory of PALM. Physically, these data are managed by the PALM driver process (`palm_main`). Warning: as long as you do not specify that they should be deleted, these objects will stay in memory.

Let's perform a time interpolation on the vector. In this example we'll choose a linear interpolation between the two closest time values. The nearest neighbour choice is also provided by default. The third possible choice is to go through a user defined interpolation between the two closest time values that has to be coded following the template provided in `palm_time_int.f90` (cf. Make PALM files)

Interpolate!

- Define three new constants: `debut_print` = 1, `fin_print` = 1000 and `step_print` = 7 (i.e. print_beginning, print_end and print_step)
- Create a second branch as:



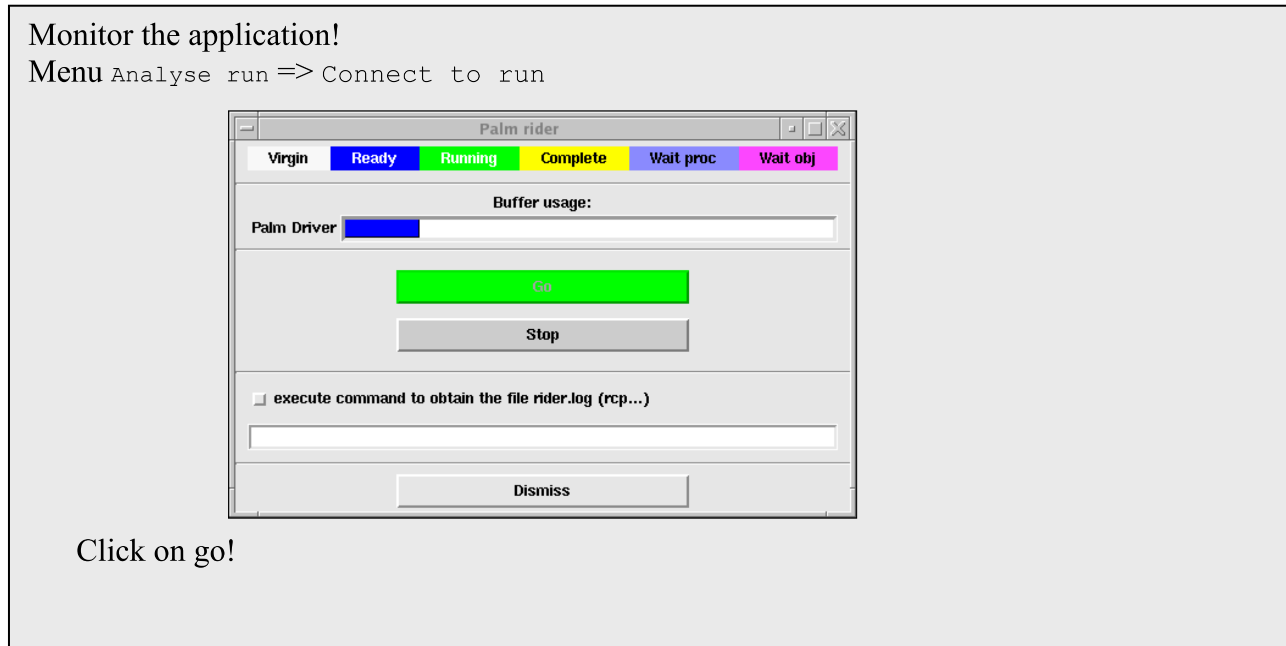
- Hardwire the loop index in the `ref_time` input plug of the vecteur_print unit
- Double-click on the vector input plug
- "time list" field: `debut_print:fin_print:step_print`
- Choose **`PL_GET_LINEAR`** for the interpolation field
- In the menu `Settings` => `Palm execution settings` check all boxes except the last one
- Still in the menu `Settings` => `Palm memory settings` set 100 for the **`memory for the driver mailbuff`** field, this is equivalent to the maximum size of the PALM BUFFER.
- Test the application

8.3 Monitoring the application in real time

It is normal that the execution takes some time to run. In the loop producing the vector (unit producer) a `sleep(1)` call artificially slows down the execution. We need it to learn how to monitor in real time the execution of a coupled simulation.

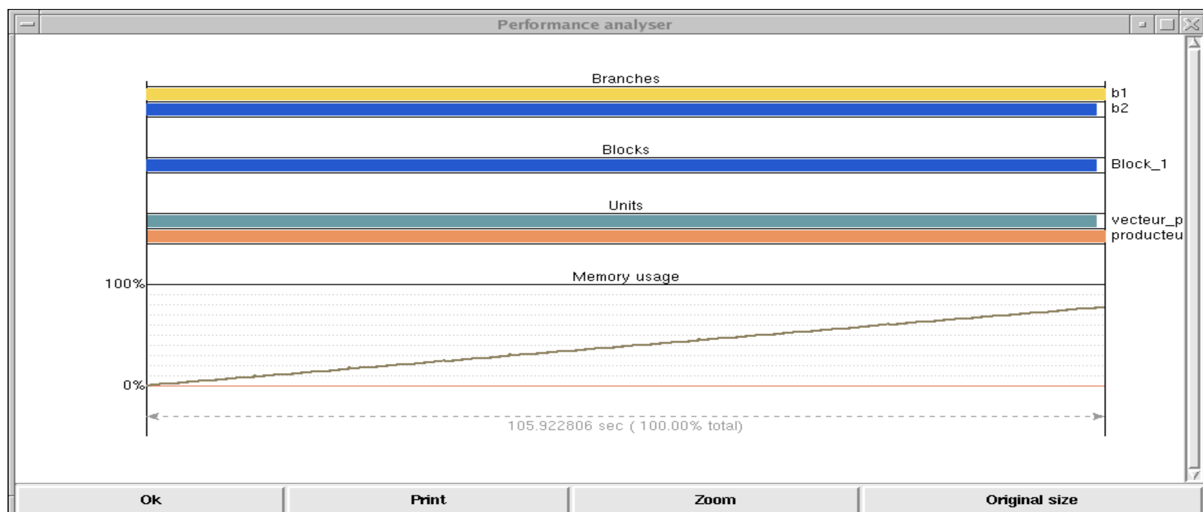
Check that, when clicking on the *Palm execution settings* item of the *Settings* menu, in the opening window the "`Trace execution for animation and performances`" option is ticked along with the "`BUFFER usage`" option.
If not, do that and restart your application.

Monitor the application!
Menu `Analyse run` => `Connect to run`



Click on go!

You'll see that in the PrePALM canvas the units colours are changing: they are green when they are running and yellow when they have finished. On the right side of the units you may notice a red number which shows how many times each unit has already completed a full run. The Buffer usage progress bar allows us to see the size of the PALM driver memory really used. You'll notice that this size is constantly increasing throughout the run. Thus, there is a risk of memory overflow. This is normal since we have specified that the objects produced at every time step had to be stored in the PALM BUFFER. We will see how to avoid this risky situation for our application. But before that, check that in the file b2`_000.log` you have obtained the good interpolated values of the vector. It is easy to check because our vector depends linearly on time. Another way of visualizing the % of the BUFFER in use consists in opening the performances analyser after having loaded the file `palmperf.log`:

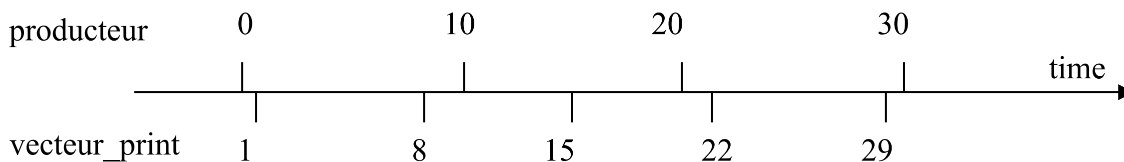


8.4 Steps, events and actions

In session 4, we have already seen how to define steps for synchronization purposes (`PL_BARRIER_ON`). The steps are events explicitly inserted at specific places in the branches. Associated with these steps, we may define some actions to be performed on the BUFFER objects. The communications managed by PALM are also events which can be used to trigger actions on the objects in the BUFFER. The association between the events and the actions is made by a programming language specific to PALM named *steplang*. In the PrePALM `Help` => `Help on steplang grammar` menu, you may find the syntax of the steplang language. You should read carefully this help .

Does it sound obscure? A concrete example will show you that it is not so difficult to use steplang.

Let us go back to our interpolation problem. The unit "producteur" produces a vector every 10 time steps starting at time 0. The unit "vecteur_print" needs the values of this vector every 7 time steps starting at time 1. In order to perform the interpolation at a required time step, it is necessary that the producteur unit produces the vector at two time steps surrounding the required time step.



For example when vecteur_print needs a vector at time step 15, it is necessary that producteur had already produced the vectors at times10 and 20. On the other hand, at this stage of the application, the vector produced at time 0 is no more useful. Same thing for the vectors printed at times 22 and 29: they do not need the vector produced at time 10 anymore (see figure).

In order to avoid any waste of memory storage, we should be able to ask PALM to perform the following action:
"At all times after time 15, when a communication from the BUFFER to the vecteur_print, is complete, delete from the BUFFER the objects "older" than the smaller time used for the interpolation".

This is easily translated into the steplang language by:

```
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);
  }
}
```

Nice trick with integers!!

The flexibility of the Steplang language together with some imagination is enough to easily translate relatively complex actions to manage the objects in the buffer.

Cleaning time!

- Menu `Step-actions` => `Edit Step-actions`
- Enter the steplang instructions described above
- Check the syntax of your script by the command `Check step-actions syntax`
- Start again your application and check that the useless objects are really removed from the BUFFER.

We have seen how to optimise the memory usage for this application. In this case everything goes well because the unit "producteur" produces its objects at a much slower pace than what is needed for the interpolations and the prints. In fact the unit producteur is slowed down artificially by the call to the function `sleep(1)` between each produced object. This call delays the execution of the unit by one second. If you comment out this call, you may observe that the memory BUFFER size extends first, and decreases only when the unit vecteur_print recovers the objects from the BUFFER. This occurs just because a **PALM_Put** is **never blocking**. For a better usage of the memory, we should consider to synchronize the two units. Synchronizations can be made quite simply by a call to the `PALM_Get` primitive which is blocking (if the Get is actually connected in PrePALM).

For this, in the producteur unit, we just need to add a call to a `PALM_Get` for an integer (synchronization) value. No matter which integer value is recovered, the only aim of this Get is the synchronization action. It must be made in the vector production inner loop, right after the Put.

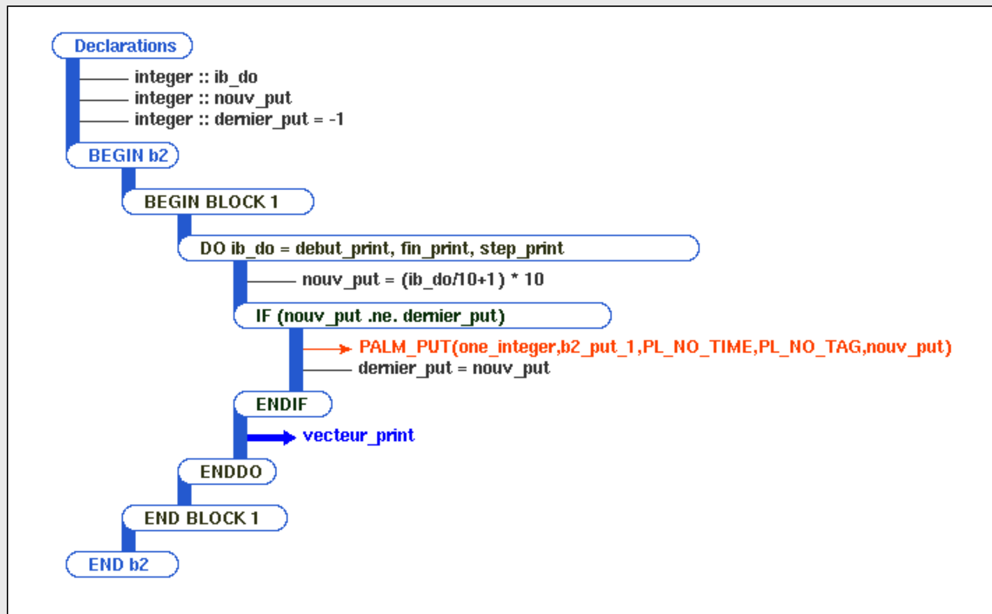
Slow down the producer!

- Edit the file `producteur.f90`
- Comment the call to `sleep(1)`
- Add a "synchro" object: one_integer IN, no time, no tag, in the ID card
- In the vector production loop, after the `PALM_Put`, make a `PALM_Get` of this new synchro object
- Reload the producer ID card in PrePALM
- The plug you have just added should appear in the PrePALM canvas.

It is necessary now to decide which source will send an object at each iteration to producteur in order to slow it down. The best way is to do this task in the vecteur_print branch where we have an external loop over time. The only problem is that the times that we manage in this branch are not the same as those of the producteur unit. Thus, some simple manipulations will be necessary to generate the good number of synchronizing Put in the branch. Fortunately PrePALM can make this kind of calculations in the FORTRAN regions.
Notice that this kind of optimization is very specific for the current coupling. The ideal working policy with PALM should be: start with generic units and implement the full coupling algorithm. Check the correctness of the results. Attentively analyse the performances, including the buffer memory usage. Apply specific changes to optimize the current coupling.

Synchronize!

- Change the `vecteur_print` branch as:



- Connect the branch `PALM_Put` to the producteur unit
- Test the application

Now, the `producteur` unit works exactly at the right pace to produce the objects when they are needed by the `vecteur_print` unit. This type of synchronization is very important in parallel computing and may serve to optimise the applications. Thus, it may be necessary to add calls to the PALM primitives, just for this purpose.

8.5 The memory slaves

In order to avoid memory overflows (e.g. in batch jobs or to prevent pagination), in the *Palm memory settings* item of the *Settings* menu you can limit the maximum size of the memory that PALM can use for internal storage, including the permanent BUFFER. If you are working on a distributed memory computer, this limitation has to reflect the maximum size you can reasonably allocate on one processor. Initially, the internal storage memory is progressively allocated on the processor running the driver. When there is no more place for the PALM BUFFER, new processes can be associated to the PALM driver. These additional processes are devoted only to the management of the BUFFER memory. When these processes are launched on other processors, they allow to use their memory as an extension of the BUFFER, and thus we can go beyond the limits of the driver alone. Notice again that this option is of interest only on distributed memory machines. On shared memory machines, each processor can access the full memory, thus making the “memory slaves” useless.
In the Palm memory settings menu you can therefore set the maximum number of additional processors that can be used to store the PALM work memory. If it set to 0, only the driver will handle this workspace. In order to avoid the overhead of starting new memory slave processes and create the communication context within them and the driver, you can already start some of them at start time. To do this, simply set the *Min memory slaves* field to more than 0.

The application that we may use to illustrate the interest of the memory slaves is still our interpolation problem. But now the objects are recovered in the BUFFER in the reverse order of their production. It is then necessary to store the whole trajectory in memory. People working on data assimilation will certainly find an interest to do that...

Ask for helpers!

- Open the file `slave_mem.ppl` with PrePALM
- Launch the application and follow it with the real time monitoring. Observe the behavior of the application

Exercise 12:
Answer the questions:
How many memory slaves are there?
How many processes (with the PALM driver)?
What is the size of the PALM mailbuff?
What do we do on step 1?
Is there an interest in making both units run in parallel?
What can we do to save a process?

Launch the application saving one process

8.6 Summary of the main concepts

In this session you have learnt how to use the PALM provided *time interpolation* to receive objects with time stamps different from their production time stamps.
This has lead us to introduce the PALM *buffer*, i.e. a permanent storage space from which the objects are not removed after reception. The way to explicitly remove objects from the buffer, goes through event-driven actions which are programmed in PrePALM with a specific language called *steplang*.
You have also learnt how to follow the state of an ongoing simulation with the *Real time monitoring* tool provided by PrePALM.
Finally you have seen some basic principle to optimise memory usage by synchronisation and how to use more processes as *Memory slaves* on distributed memory computers.

9 Session 9: Space inheritance and dynamic objects

Until now, in every unit we used, the memory size of the objects was known at compile time. We have seen that this size could be parametrised via the constants of PrePALM, however it was static, i.e. it could not be changed at run-time. The computer codes which could be coupled with PALM may involve objects whose size is known only at execution time. These objects are typically dynamically allocated arrays. Let's see now how we can deal with this type of objects.

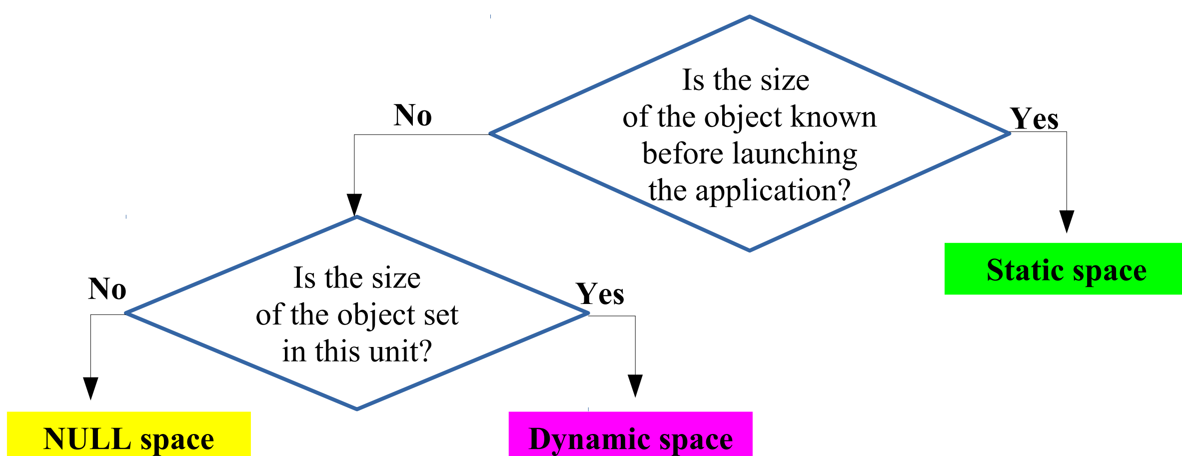
A first remark can be made: the size of a *dynamic* object must, of course, be known before the exchange, and this information can only be provided by the unit which produces the object. Only the sending unit may know its current size. Thus, the source unit must, before the sending, let PALM know the actual size of its object so that the coupler can transfer the right message size: the `PALM_Space_set_shape` primitive will be used for that.

The receiving unit should use a compliant space for the exchange: since the units are supposed to be independent (spaces are private to each unit) in this case a *NULL* space can be specified for the object in the receiving unit identity card. It means that the space definition will be derived from a communication described in the graphical user interface PrePALM: we say that the receiving unit *inherits* the properties from the source object.

In PrePALM the dynamic objects correspond to pink plugs. Objects with an indefinite space (NULL) correspond to yellow plugs. No static space will be given a pink or yellow colour code.

Moreover, you should notice that NULL spaces can indifferently inherit the properties from a dynamic space or from a static space. Algebra boxes, which have been presented in session 6, use NULL spaces in order to be generic. For example, in a single application, we may have several instances of the same pre-defined unit working on objects of different size. This would not be authorized with constants, only the NULL space and the inheritance mechanism can make it possible. For the same reason we recommend the use of NULL spaces when implementing units that can appear more than once in the same application if they can work on objects of different size/shape.

Important to note: You can easily find the type of space of an object, after answering both questions in the diagram below.



Exercise 13:
In this exercise, the unit `producteur` will now produce a matrix and a vector at the time `PL_NO_TIME`. The matrix and vector sizes are entered via the keyboard. A second unit, `produit_mv` (= product_mv) will compute the product of the matrix by the vector and will pass the result to the unit `vecteur_print`. The units `produit_mv` and `vecteur_print` must be able to accept matrices and vectors of any size.

Answer the questions:
How many objects for `producteur`, which spaces (static, dynamic, NULL)?
How many objects for `produit_mv`, which spaces (static, dynamic, NULL)?
How many objects for `vecteur_print`, which spaces (static, dynamic, NULL)?

Edit the unit `producteur.f90` in the `session_9` directory
By which primitive call does PALM know the matrix and vector sizes?
What are the second and the third argument of this primitive?

Edit the unit `produit_mv.f90`
Note the spaces and the objects NULL in the ID card.
How do we know the inherited space?
How do we know the space size?

Improve the unit `produit_mv` so that this one stops the PALM application if the matrix or the vector is not received.

Assemble these three units in a branch, in a block, in a loop which will enable you to request three times the vector size. Run the application and check the results.

Unit 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
```

Unit 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
```

Exercise 13 bis:
Create a second branch START_OFF, move the unit "vecteur_print" into this branch and launch the second branch from the first one, at the start of the loop, just before calling "producteur".

If you run the application, it will terminate with PALM_Abort, and the following message can be found in palmdriver.log:

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

Synchronise the application in order to define the space before it is used in "vecteur_print". You can synchronise the units either by a step barrier or via a communication.

9.1 Summary of the main concepts

In this session you have learnt how to describe in PALM spaces of which you don't know the size beforehand. On the sender side you can define run-time the space size with the `PALM_Space_set_shape` primitive. Such a space is said to be *dynamic*. On the receiver side you postpone the space declaration till run-time, giving to it the *NULL* label and the space size is said to be *inherited* at run time. Some additional primitives (`PALM_Object_get_spacename` and `PALM_Space_get_shape`) let you retrieve the space features in order to perform the necessary allocations before the actual PALM_Get.

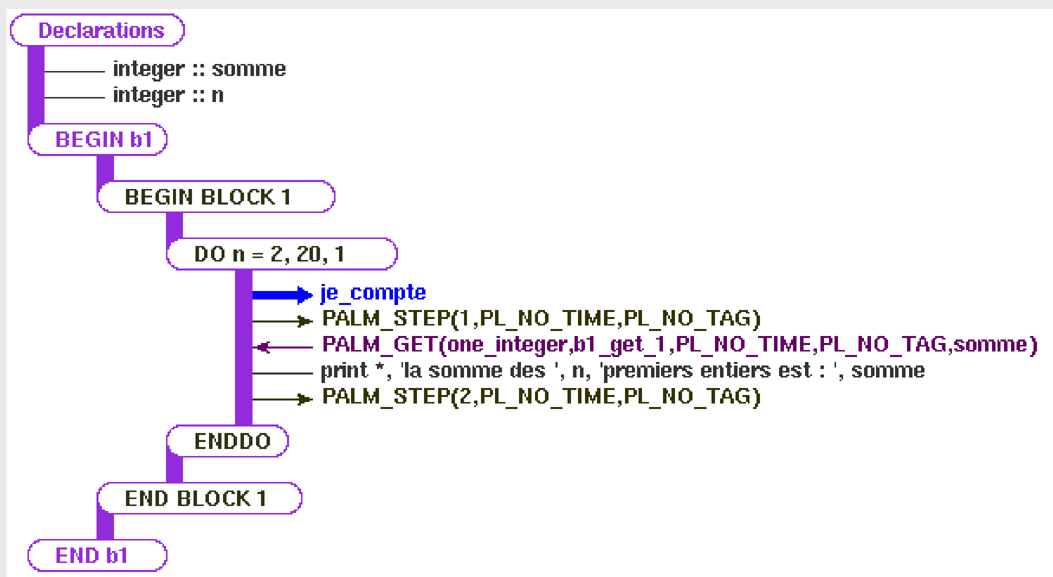
10 Session 10: Assembling objects in the BUFFER

We have introduced in session 8 the PALM BUFFER and we have seen how to use it for keeping objects in a permanent memory and thus to be able to recover them repeatedly or to interpolate them in time. Another role of the BUFFER is the object assembling, for example to calculate averages or sums of objects. The underlying idea is that several sources of communications can contribute to the same result. We allocate a place for the final result in the PALM BUFFER and we address all contributing communications toward this location. Since we do not know necessarily in advance how many contributions are going to be collected, the result is flagged as *not ready* and therefore cannot be received with a `PALM_Get`. We are once more going to use the *steplang* to describe the conditions under which the result can be considered *ready*.

For our training session, the unit `je_compte.f90` (= I_count.f90), that you may find in the directory `session_10`, produces in a loop the integers from 1 to N, N being an input of the unit. If a 4 is given in input to je_compte, it produces 1, 2, 3 and 4. We want to compute the sum of the integers produced by this unit and let the branch post the result. When repeating the procedure with N ranging from 2 to 20, we will start the unit `je_compte` by initialising each time the sum to 0. You will learn how to reset to 0 and to flag as *not ready* an object in the BUFFER, using the *steplang*.

Arrange!

- Create the following branch:



- In the PrePALM canvas, define the communications:
  - The input n in `je_compte` takes the values of the loop index (right click)
  - The output of the integers is stored in the BUFFER as the object `somme` (=sum). Check the field "`add`" of "Palm algebra" and set 1 and 1 for the assembling coefficients.
  - The branch issues a `PALM_Get` to recover this object from the BUFFER
- An object being assembled in the BUFFER is not supposed to be ready. With the steplang help (`Help` => `Help on steplang grammar`) write an instruction which sets the object state to "`ready`" on step1 and which reset its value to 0 on step2.
- Test it

Do not forget this PALM function. Often, it may avoid a call to a pre-defined algebra unit or another user unit. It can be useful for example for changing the physical unit of a field exchanged between two PALM units which are not using the same units system (you can easily guess how to play with assembling coefficients).

10.1 Summary of the main concepts

You have seen another important and often useful usage of the PALM BUFFER. When addressing a communication to the BUFFER you can ask instead of replacing any possible previous copy to *combine* the new with the old value (you indicate the coefficients of the combination).
To avoid any synchronization problem, you have seen how to act on objects stored in the buffer to change their status flag and/or to reset their value to 0. This is just another use of the event driven actions described with the *steplang*.

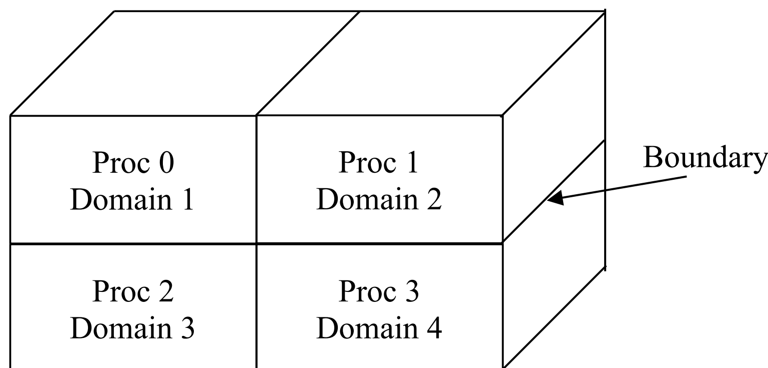
11 Session 11: Parallel communications

11.1 Introduction

PALM has been designed to manage parallel units. In session 2, with the calculation of π, we have already seen how to launch such units. When we talk about parallel programming, this also imply to say something on data distributions. Now will should see how PALM can manage distributed data in order to facilitate the exchanges between units.

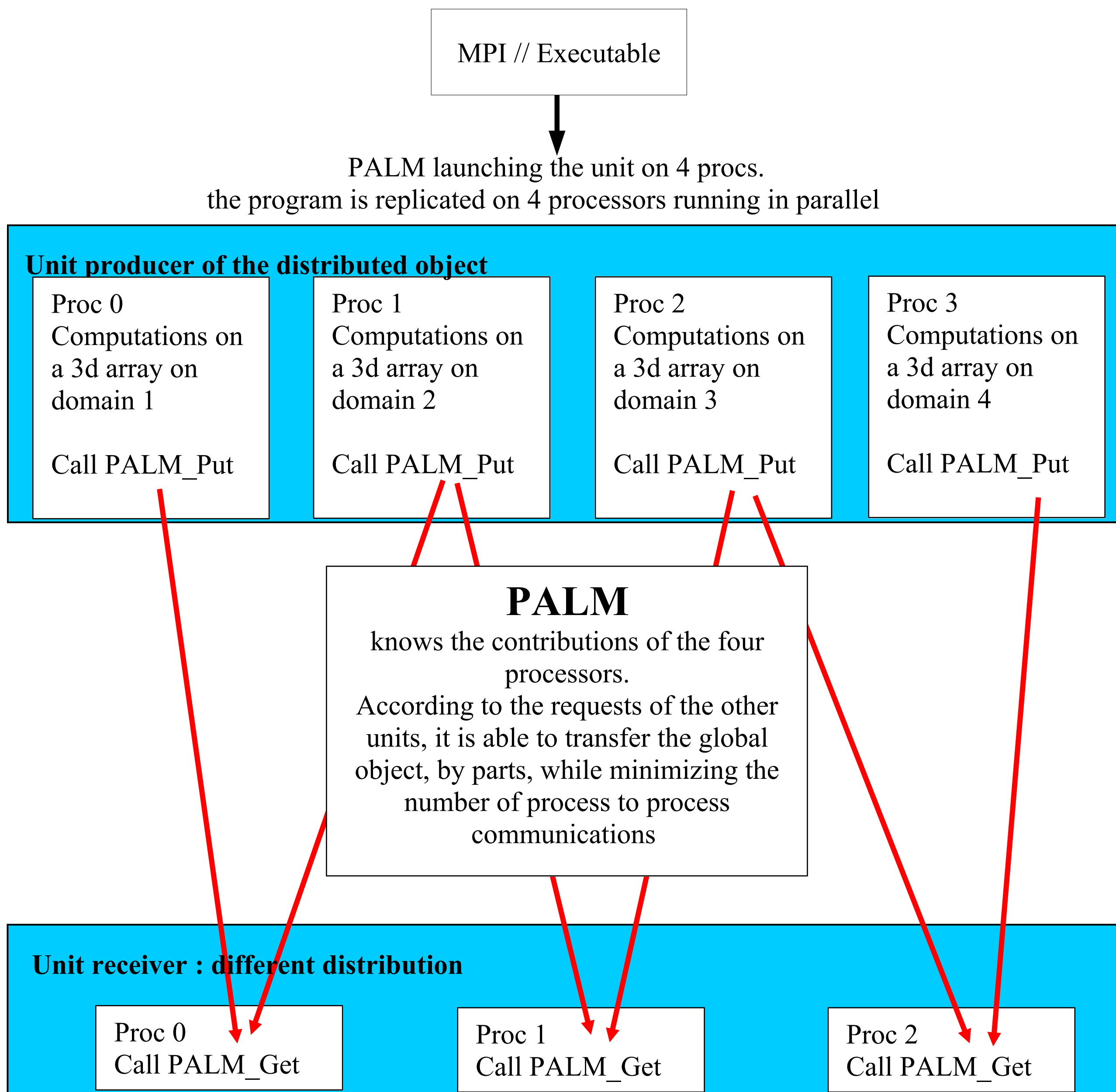
First of all, some concepts of parallel computing are essential for this session. We deal here with the so called domain decomposition parallelism, implemented with the SPMD (Single Program Multiple Data) paradigm. For the treatment of the whole domain, the same program is replicated on several processors (or processes) sharing the problem to be treated, according to a strategy which is defined by the programmer. This type of programming is not possible without the use of a parallel library like MPI. At execution time, each process specializes itself and treats only part of the problem. Note that the domain decomposition is not made by PALM, but the coupler will know how the data are distributed in order to manage the object as a whole.

To be concrete, let us imagine an ocean circulation model where the 3d global domain is discretized by finite differences in the form of elementary grid cells. This type of program works on 3d arrays (corresponding to a physical domain discretization) which cover the whole Earth. The code is parallelized in order either to reduce the computing time, or simply to be able to treat a large problem which does not hold in the memory of a single processor. Each process treats only a part of the virtual 3d array. In general, with this type of parallelism, you need to exchange at each time steps some information at the borders of each local domain to be able to continue the iteration process. This does not prevent us from being able to consider the global 3d arrays as a PALM object, even if it is physically distributed in the memory of several processors.



Example of distributing a 3d field on 4 processors

Each processor holds only one portion of the whole array. The local arrays stored in each processor does not have necessarily the same size in memory. To avoid gathering the global array in only one processor before sending it, PALM offers the possibility that each processor issues a PALM_Put of only the part of the field it knows.



If we want PALM to be able to handle such exchanges, it is necessary to describe how the objects are distributed, on the source side and on the target side. It is a necessary and sufficient condition. This is the role of the distributors.

11.2 The distributors

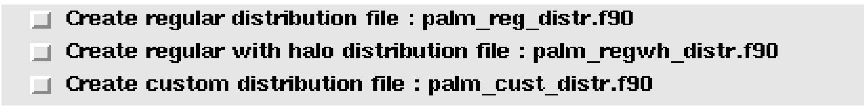
Distributors are nothing else but the way of letting PALM know how the objects are distributed, that is to say how the code has been parallelized. The relevant information is: “what is the part of the global object managed by every single process and how is is stored locally?”. In PALM we have introduced a syntax to describe this information. To grant enough flexibility, it is possible to describe the distributors in two different ways suitable for different approaches.

The distributors of **regular** type allow a description of a basic pattern which is repeated inside the global array. These distributions, directly inspired by the decompositions used by parallel scientific

libraries like ScaLAPACK, are very concise but are quite often not well adapted to the data distribution used in models.

Distributors of **custom** type are less concise but they allow to describe any kind of distribution.

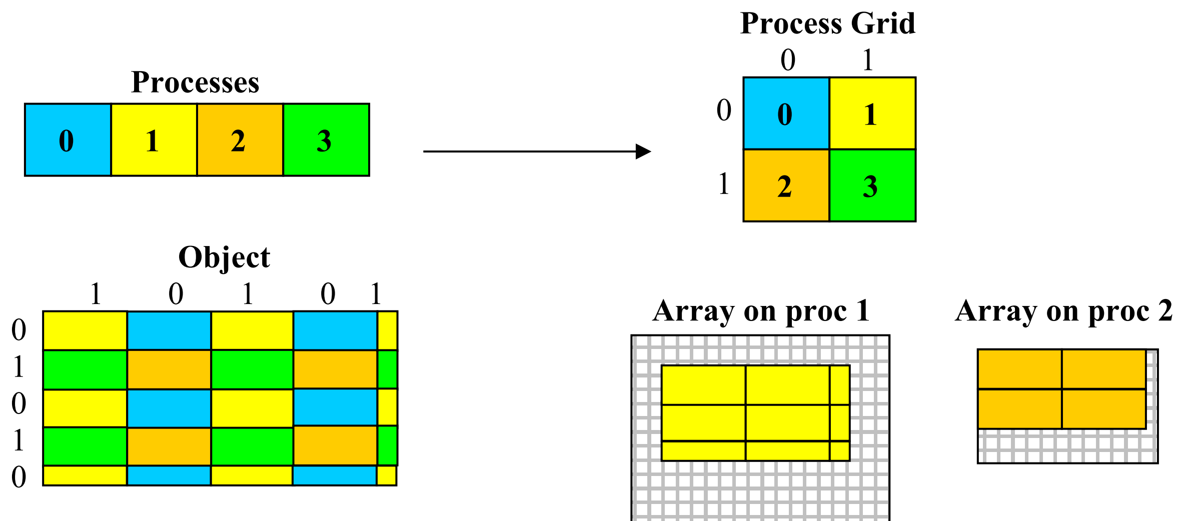
The objects are associated to distributors in the identity cards of the respective units.
Although it is possible to describe the distributors directly in the unit ID card in the form of a list of integers or constants, it is highly recommended to describe it in a distribution function. PrePALM will provide you with a distribution function template if you check one of the boxes in the dialog box appearing in the *Make PALM files* menu:



11.3 Block cyclic distributors

This distribution is directly inspired by the distributions used in the ScaLAPACK library. In PALM, they are called "**regular distributions**".
The processes involved in a distribution are organized according to a multidimensional grid, having the same number of dimensions as the global object. The global object is split into blocks in a regular way, n_i elements per block along each dimension i.
If the size of the global object along a dimension is not an exact multiple of the blocks size in this dimension, then the last block is smaller than the others.

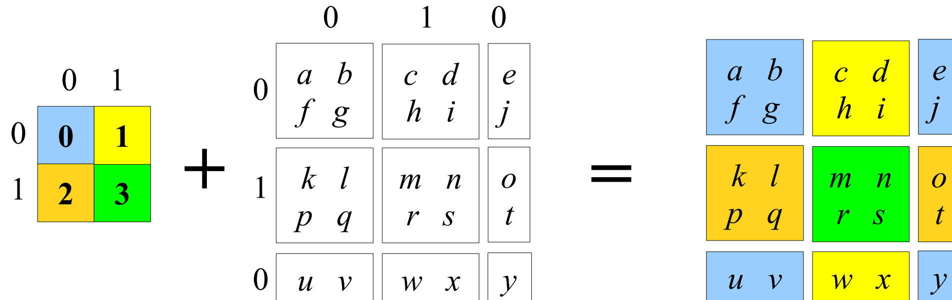
Example of a 2d object distributed on a 2x2 process grid:



To define the order in which the blocks are assigned to the processes, the blocks are cyclically numbered in each dimension, with a cycling length corresponding to the process grid size in this dimension. The choice of the number from which one starts to count in every dimension is arbitrary.
In this example, we choose a 2x2 grid, therefore we number every direction form 0 to 1. We dispose the 4 processors on the grid and we associate them to the two coordinates of the grid (top right image. We split the object in blocks of the given size (plus the remainder), then we number every dimension from 0 to 1, arbitrarily starting from 0 for the rows and 1 for the columns, cycling

back to 0 as many time as needed. Mapping the block coordinates on the process grid coordinates we know on which process the block is stored.
Notice that the blocks are stored in a local object (greyed rectangles) in a contiguous way. The local object can be larger than the total size of the blocks.

Let see how it works on the concrete example of a 5x5 matrix to be distributed on a 2x2 grid of processors, starting to count from 0 both for the rows and the columns.



Therefore we obtain the following distribution:

Processor 0 *a b e* *f g j* *u v y*	Processor 1 *c d* *h i* *w x*
Processor 2 *k l o* *p q t*	Processor 3 *m n* *r s*

The necessary information to describe such a distribution is as follows:

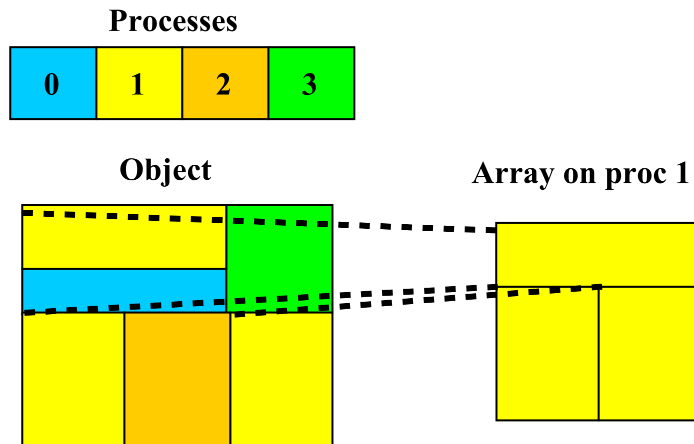
1. The processor grid shape (i.e. its size in all dimensions): (2,2) in the previous example
2. The elementary block size: (2,2) for this last example
3. The coordinates in the process grid which will contain the first global object block: (0,0) in this last example
4. For each process:
    1. the local object shape
    2. the coordinates in the local arrays of the first element of the first block

This way of describing the distributed objects is very concise but it does not allow to represent all the possible types of distribution.

11.4 'CUSTOM' distributors

This distribution method is the most flexible: for each process, we give the list of the blocks which are stored in it, and their place in the local object. This method is compatible with any distribution. For example, it is well adapted to a domain decomposition using a non-structured grids because the blocks can be placed anywhere in the local object.

Example:



The information needed to describe such a distribution are:

1. The shape of the local object
2. For each locally stored block
   1. the shape of the block
   2. the coordinates of the first element of the block in the global object
   3. the coordinates of the first element of the block in the local object

11.5 Examples of distributed objects

In the directory `session_11/cas_reel`, edit the file `toy_ocean.f90`. Answer the following questions. Use the ID card to help you.

- Is the unit parallel and of which type?
- How many processes the unit `orca_toymodel` can use?
- How many objects are defined (IN and OUT)?
- How many objects are distributed?
- What is the distributor associated with the object “field”?
- What are the object rank and the distributed object global size?

Looking more closely at the distributor:
- What is the distributor's type?
- How many processes does the distributor use?
- In which file is the distribution function ?

Look at the FORTRAN code:
- What is the variable which will contain the local distributed object?
- Why is this variable dynamically allocated?
- What is the size of this variable?
- What is the subroutine which determines this size and which parameters are involved?

Now open the distribution function in the file `ocean_distrib.f90`. Notice that this function, whose template was generated by PrePALM, offers two calling modes according to the argument `id_action`. This function is not explicitly called in the units code, but directly by PALM. The first operating mode returns the size of the array which will contain the distributor (this is useful for PALM to dynamically allocate the work array containing the distributor), the second mode returns a vector of integers containing the distributor

In our example, notice that the distribution function calls the same subroutine as the one used by the unit `toy_ocean` (`my_domain`). This function determines the field decomposition on each processor. You should know that if you have to write a distribution, it is not arbitrary: it depends entirely on the way the code has been parallelized. Thus, writing a distribution function often is as simple as a copy/paste of the part which parallelizes the code (the definition of the domains decomposition) rewritten with a syntax that PALM can understand.

Distribute!

- In the directory `session_11`, launch PrePALM
- Add the 3 following constants:

ip_nlon_ocean	182
ip_nlat_ocean	149
ip_nbproc_ocean	8

- Load both units: `toy_ocean.f90` and `plot_tcl.f90`
- Launch them in 2 different branches
- Set the right max number of processes for the application
- Make the model inner DO-loop runs from 0 to 100 by steps of 20 (6 times) by hardwiring the values for the input min_time, max_time and freq_time
- Add a DO-loop (0:100:20) around the unit plot_tcl
- Send the objects `lon`, `lat` and `msk` produced by toy_ocean to the buffer because they are generated only once by toy_ocean but the unit plot_tcl needs them each time it is launched
- Send the produced field to the unit plot_tcl. Do not forget to fill the field *time*. The thick communication line represents a parallel communication
- Concerning plot_tcl :
    - hardwire the DO-loop index as input "time"
    - ip_nlon_ocean as nlon
    - ip_nlat_ocean as nlat
    - for lon, lat and msk, recover these objects in the BUFFER
    - set 3000 for the field "refresh"
- In the branch where plot_tcl is running, after the unit launch, insert the following script: wish plot.tcl

Exercise 15:

Make the model work on just one time step, and modify the toy_ocean source code so that the produced fields depend on the process (for example you may initialize the fields with the value of the process rank) and so, the domain decomposition becomes visible. You can set the `refresh` object of the unit `plot_tcl` to -1 in order to keep the drawing on the screen. In this case, also add the character "`&`" (backward launching) to the end of the command : `wish plot.tcl`. This will allow the PALM application to finish before closing the drawing.

Test different values for the number of processes of the unit "toy".

In our example, only one out of the two units is parallel. The exchanged object is distributed on the toy_ocean side. Notice that it is possible to exchange distributed objects on both the source side and the target side with identical or different distributions; nothing is impossible with PALM! The fact that a unit is parallel does not change anything in the coupling algorithm. It is thus very easy, in

order to save time in a PALM application, to parallelize only the units which are the most time consuming.

Although the distribution function is sufficiently generic in our case to work with a variable number of processors, the distributed objects cannot, in the PALM_MP 3 version, be dynamic: This feature is under development.

11.6 Localisations and process associations

The example we have just run is rather simple, but we may encounter much more complex cases. For example in a parallel code, the domain decomposition of some fields can be done on just a subset of the processor set used during the run. For example, one processor is usually specialized for the I/O. In the same way, some objects cannot be distributed, but are simply replicated on all the processes. All of these characteristics must be communicated to PALM. This is done again via the ID cards (since the localisation is an attribute of the object, it is defined in the id card) and the graphical user interface while describing the communications.

Let's concentrate on the ID card: have a look to the toy_ocean unit and you will see that there are two fields `-localisation` in the object descriptions.

```
!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}
```

The localisations may specify two things for the objects of distributed units :

1) Make the difference between distributed and replicated objects
2) Specify on which processors the distributors apply, and in which order.

In practice, there are two ways to describe the localisations

1) use the predefined localisations (as in the example):
   DISTRIBUTED_ON_ALL_PROCS: the object is distributed on all unit processes
   REPLICATED_ON_ALL_PROCS: the object is replicated on all unit processes
   SINGLE_ON_FIRST_PROC (default localisation): the object is neither distributed, nor replicated; it has only one instance located on the unit process 0.
   This keyword can be used in the `-localisation` field for an object.

2) if the localisation does not correspond to a predefined choice, it has to be explicitly described For example let us suppose that your object is distributed on the proc 0, 4, 3 and 2 whereas your unit runs on 5 processors, in this case you should define in the id card the following localisation:

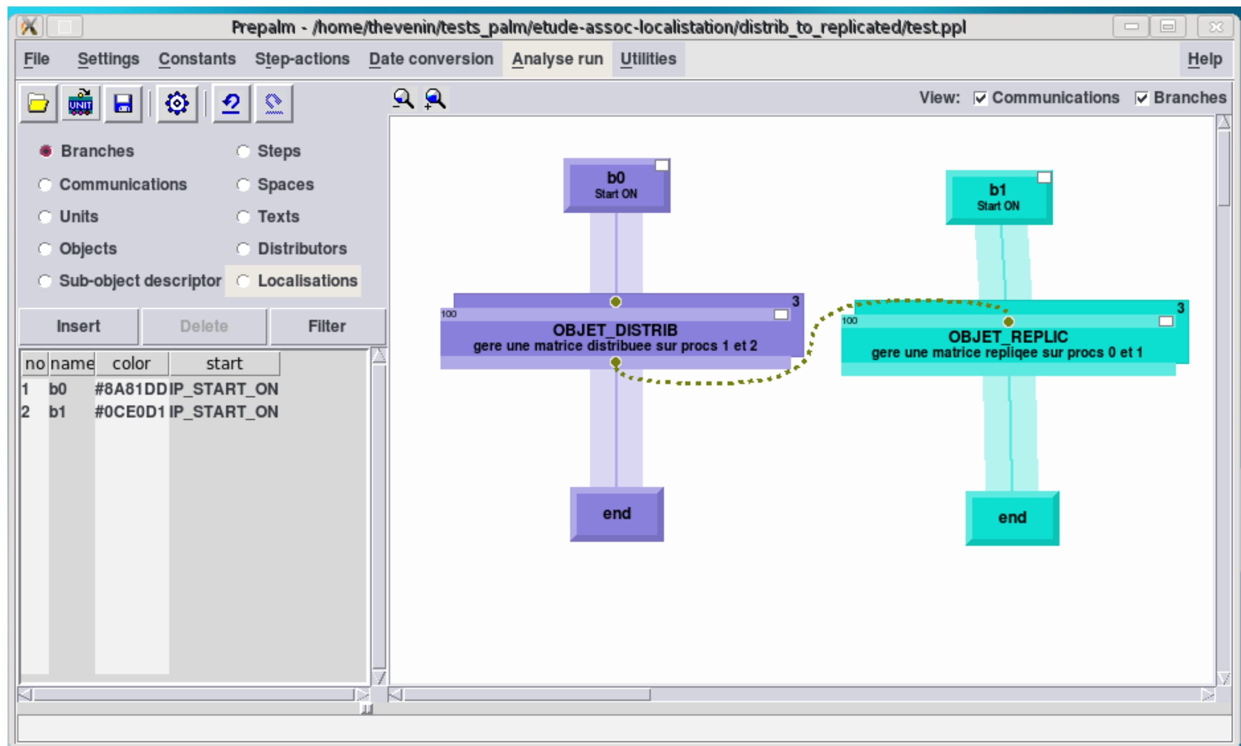
```
!PALM_LOCALISATION  -name name_of_the_localisation\
!                   -type distributed \
!                   -description {0;4;3;2}
```

The `-name` field associates a label to the localisation that can therefore be used in the -localisation field of an object. The `-type` field can be `distributed` or `replicated` (*cf.* infra). The field `-description` identifies the concerned processes. The syntax of the ranges is the same as in the *time stamps* and *tag* lists.

Once the localisations have been defined and associated to objects, the user still has to fill the `local assoc.` field in the windows describing the communication properties. It is important to distinguish the case of a distributed object from a replicated object.

An object is said to be **distributed** when it is split on several processes of a unit so that each one treats a local part of this object. The distributor of the object describes the way in which the object is decomposed (number of processes on which the object is distributed, blocks size and coordinates defined in the global object, local arrays sizes containing these blocks...). In this case, the localisation field describes the list of the unit processes on which the object is distributed, i.e. the numbers of the unit processes which will work on the local parts of the object described in the distributor. In the communication properties window, in the association field the user simply has to indicate the **rank of the first process** taking part to the distribution.

An object is said to be **replicated** if several unit processes to which it belongs work on an independent and full copy of this object. In this case, the localisation describes the list of the unit processes which need an instance of this object. In the communication properties window, the user has to indicate the **ranks of all processes** using a copy of the object. The syntax is:
start1[**:end1**[**:step1**]] [| **start2** [**:end2**[**:step2**]]] [; …]
Example :
Let's consider two parallel units that exchange an object. Both units run on 3 processes. On the first unit (`OBJET_DISTRIB`), the object is distributed on process 1 and process 2, while on the second unit (`OBJET_REPLIC`), the object is replicated on process 0 and process 1 :



In the identity card of `OBJET_DISTRIB` we read

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

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

Notice that in the object description we reference the previously defined distributor `distr` and the previously defined localisation `loc`.

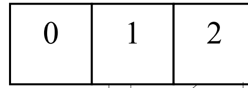
In the identity card of `OBJET_REPLIC` we read

```
/*PALM_LOCALISATION -name loc\
                    -type replicated\
                    -description {0;1}
*/
/*PALM_OBJECT -name field\
              -space matrice\
              -localisation loc\
              -intent IN\
```

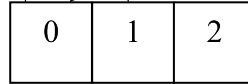
```
                  -comment {matrice repliquee}
*/
```

In the actual communication the object is gathered from the processes 1 and 2 of `OBJET_DISTRIB` and a full copy of it is sent to procs 0 and 1 of `OBJET_REPLIC`. In the unit code there are just simple `PALM_Put`'s and `PALM_Get`'s.

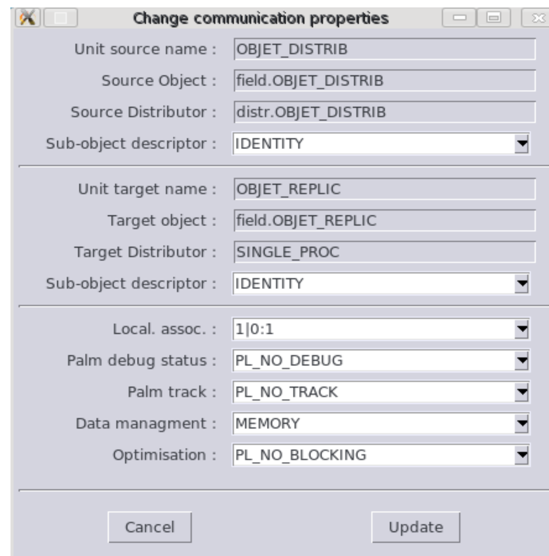
Unité OBJET_DISTRIB :



Unité OBJET_REPLIC :



When defining the communication the user has to accurately indicate how the instances of the object are exchanged between the units. For this example, we have (*cf.* the `local. assoc.` field)



When dealing with replicated objects, in most of the cases, this association can be deduced from the localisations, so it may be sufficient to select the `AUTOMATIC` association suggested by default by PrePALM. In this case, the association is treated as described in the array below:

Source	Target	Association
SINGLE_ON_FIRST_PROC	SINGLE_ON_FIRST_PROC	The object, not distributed is sent from proc 0 to proc 0 Equivalent to assoc: 0
SINGLE_ON_FIRST_PROC	DISTRIBUTED_ON_ALL_PROCS	Each part of the object, not distributed on the source side, is sent to the different processes on the target side Assoc: 0
SINGLE_ON_FIRST_PROC	REPLICATED_ON_ALL_PROCS	The object, not distributed on the source side, is sent entirely to all processes on the target side.

		Assoc: 0 | 0 : nbproc_tgt-1
DISTRIBUTED_ON_ALL_PROCS	SINGLE_ON_FIRST_PROC	The object, distributed on all processes on the source side is sent to process 0 on the target size. Assoc: 0
DISTRIBUTED_ON_ALL_PROCS	DISTRIBUTED_ON_ALL_PROCS	The object is distributed on both sides on all the processes. Assoc: 0
DISTRIBUTED_ON_ALL_PROCS	REPLICATED_ON_ALL_PROCS	The object, distributed on the source side, is gathered and then sent to all processes on the target side. Assoc: 0 | 0 : nbproc_tgt-1
REPLICATED_ON_ALL_PROCS	SINGLE_ON_FIRST_PROC	Not treated
REPLICATED_ON_ALL_PROCS	DISTRIBUTED_ON_ALL_PROCS	Not treated
REPLICATED_ON_ALL_PROCS	REPLICATED_ON_ALL_PROCS	Each process, on the source side, sends its object to the process with the same rank, on the target side. Assoc : 0 :nbproc_src-1

You should not worry; even if PALM offers this mechanism to be able to treat all kinds of parallel communications, it is quite unusual to have to describe a localisation which differs from the pre-defined ones.

11.7 Summary of the main concepts

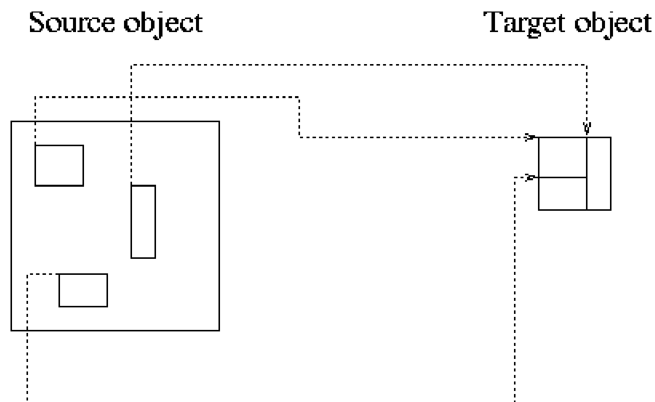
In this session you have seen how PALM handles the communication between parallel units.
In particular you have seen how the user can describe the way an object is stored in the memory of the single processes of the units. This lead us to introduce the concept of *distributor*, i.e. the syntactical entity describing a distribution. For the sake of efficiency, we defined two kinds of distributors, the *regular* ones, well suited for block cyclic distributions (on the style of ScaLAPACK, for instance) and the *custom* ones, less compact but able to describe any kind of distribution.
Finally you have learnt how to use the *localisation* attribute of an object to describe on which subset of processes it is hosted or replicated and the *local association* field of the communication properties to map the localisation on the source side onto the localisation on the target side.

12 Session 12: Sub-objects

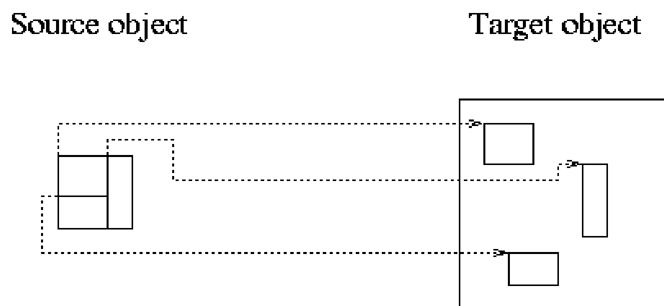
The *sub-objects* have been introduced in PALM_MP to grant an even higher level of independence between the units. It may be useful for example, to recover only a part of an object in a target unit without having to modify the code of the source unit. For this purpose we may use sub-object descriptors. A sub-object is seen as a set of sub-blocks of the object which it belongs to.

This feature allows:

- to recover in a target unit only a portion of an object produced by a source unit

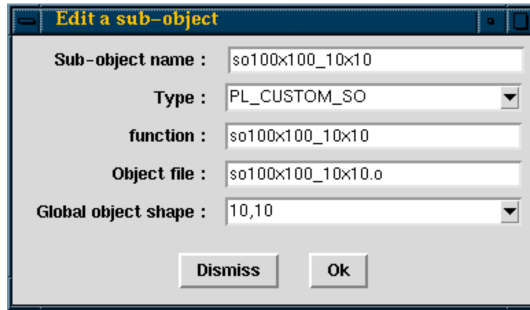


- to update in a target unit only part of an object by a PALM_Get,



The sub-objects must be defined in PrePALM and not in the units, since they are completely dependent on the application in which they are used. This is why they are not defined in the unit ID card.

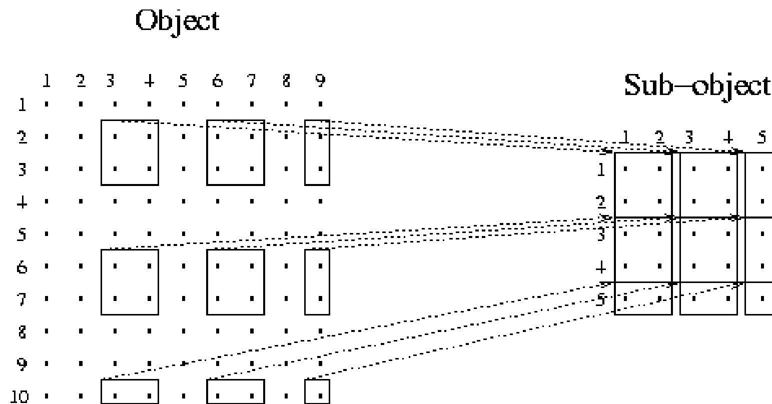
To define a sub-object, the user must check the box **`Sub-object descriptor`** in the Category Selector then click on the **`Insert`** button. A window pops up. The user must enter the name of the sub-object, its type, the name of the function which describes it, the name of the compiled file which contains the function, and the shape of the object for which it is defined (in the following we will call this object a *global object*).



The different types of sub-objects correspond to the different models which describe them. A sub-object can be of type **PL_CUSTOM_SO** or of type **PL_REGULAR_SO**. We find here the same terminology as for the distributions descriptors. The REGULAR model will be used to easily describe regular sub-objects and the CUSTOM model will be used in the other cases.

A regular sub-object is a sub-object in which the blocks have the same size (except possibly for the last in each dimension) and are regularly spaced in the global object.

To describe a sub-object, the user must write a function adapted to the sub-object type which he created. A template of the various functions is provided by PrePALM when checking **Create regular sub-object file** or **Create custom sub-object file** in the Make PALM file window.



Example of a regular sub-object

The functions describing the sub-objects are built with the same philosophy as those of the distributions. The user has to fill a vector of integers with the information describing the sub-object.

To describe a regular sub-object, the user must provide:

- the sub-object rank,
- its shape,
- the elementary block shape,
- the number of blocks in each direction,
- the size of the gap between each block in each dimension,
- the coordinates of the top left corner of the top left block of the sub-object in the global object.

To describe a custom sub-object, the user must provide:

- the sub-object rank,

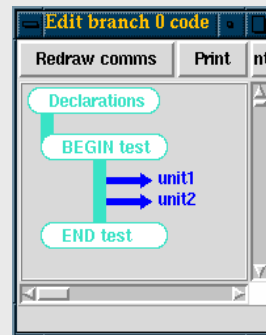
- its shape,
- the number of blocks of the sub-object,
- the description of each block (shape, coordinates of the top left corner in the global object and in the sub-object).

When a sub-object is created in PrePALM, the user can use it when he defines a communication by indicating the name of the source and target sub-objects for this communication (the default value for the sub-object being IDENTITY, which means that the sub-object is identical to the object).

In the directory `session_12`, you will find two units. The unit `unit1.f90` produces a 2d array of 100x100 reals and issues a PALM_Put of these data. The unit `unit2.f90` issues a PALM_Get of a 10x10 real array and print out those elements. We are going to see how unit2 can pick just the centre of the array generated by unit1.

Take a chunk of an object!

➢ In the directory `session_12`, launch PrePALM and Create the following application:



➢ menu `File` => `Make Palm files`, check the choice `create custom sub object file` and create the file `palm_cust_so.f90`

➢ Rename this file as: `so100x100_10x10.f90` and edit it:
You need to modify 3 lines in this file:
  - The name of the subroutine: **`so100x100_10x10`**
  - The number of blocks: 1
  - The vector of integers which contains the descriptor:
    `ida_descr = (/10,10, 1, 10,10,45,45,1,1/)`

➢ In PrePALM, select the category `Sub-object descriptor`, in the main window top left pan, then on the `Insert` button, just below

➢ Fill the dialog box as follow:

Insert a sub-object

Sub-object name : so100x100_10x10
Type : PL_CUSTOM_SO
function : so100x100_10x10
Object file : so100x100_10x10.o
Global object shape : 100,100
Dismiss | Ok

➢ You just need to create the communication by using the sub-object descriptor on the target side

12.1 Summary of the main concepts

This short session taught you how to associate an object on one side of a communication with only a compliant portion (or, to be precise, a collection of sub-blocks with compliant global shape) of the object on the other side.
This feature is of relevant importance to avoid heavy intervention on the source code when just a subset of a field is exchanged during a coupling. A typical example of application is the coupling via the surface layer of a 3D field. Instead of extracting the 2D array containing the surface level and issuing a PALM_Put of it, it is much simpler to PALM_Put the whole 3D object and ask PALM to exchange only the sub-object corresponding to the surface layer.

13 Session 13: Read and write in files, geophysical fields interpolation

As we have seen above, the `PALM_Get/Put` primitives allow the PALM units to, respectively, require information or to make them available. The sending or the receiving of data is effective only when communications are described (correctly!) in PrePALM. In general these data are provided or dispatched towards other units of the same application. Everything occurs in memory and the data handled by the `PALM_Get/Put` primitives are lost after the execution of the application.

While handling these data (for which the data-processing characteristics were described in the ID cards) it seems natural to be able to store them also in a permanent way in files. Conversely, we may well imagine that units requiring data could read them directly from files without having to create a unit dedicated to this task. This is the role of the PrePALM files: the idea is to be able to directly connect the plugs corresponding to the Put/Get of the units, to existing or new files. A call to `PALM_Get` in a unit will start a reading, a call to `PALM_Put` a writing.

The file format selected in PALM is NetCDF, a standard well-known in the climate community, which offers several advantages:

- The format is self-descriptive; the file contains a header which describes what is contained in the file.
- The file access is direct; the records can be read/written in any order.
- The data are stored in an optimal way in terms of data size because they are written in a binary format, but unlike the FORTRAN binary, this format is portable from one machine to another. The NetCDF binary preserves the machine precision.
- The NetCDF library is installed on most computers. If this is not the case, its installation is very easy.
- A number of pre- or post-treatment software are using this format.

As an illustration, let us take again the toy model of the realistic case from session 11. At each time step, the model `toy_ocean` produces 2d fields on a relatively complex discrete spherical grid (`orca2` grid from the largely adopted model developed at LOCEAN in Paris, which is a structured but non regular grid). We will store these fields in a file and then we'll spatially interpolate them on a different grid (grid from the Météo-France ARPEGE model). This case would correspond, for example, to the use of sea surface temperature data issued from an ocean model as a forcing in an atmosphere model.

The first thing to be made is to describe the file format which will contain these surface fields for each time step. For PrePALM, NetCDF files are described like units, through ID cards.

Store!

- Open the file `champs_orca.id` (=orca_fields.id): ID card of the file we will manipulate.
- Notice the keyword `PALM_FILE` instead of `PALM_UNIT`.
  Also notice the heading `-shape_label` in the spaces definition. It is an additional information compared to the usual spaces definition in units. Beside that, one finds the same information as in the ID cards of the PALM units.

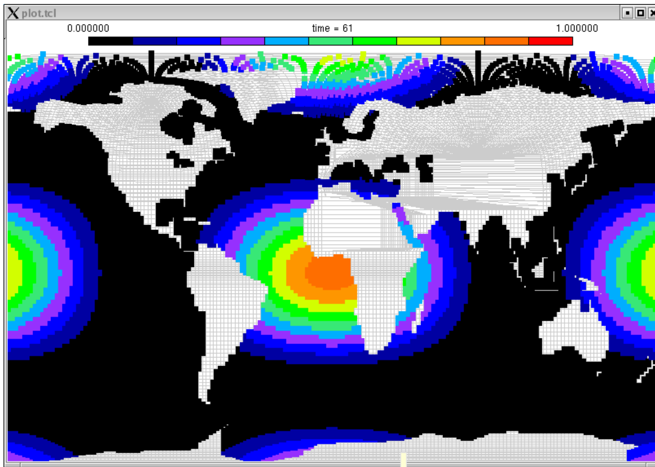
- ➢ With PrePALM, open the file `creation_fichier_orca.ppl` (=create_orca_file.ppl). To understand the application, answer the following questions:
  - On how many processes does the toy model run?
  - How many time instances of the object `field` will be produced?
  - Why is the communication between the object `field` and the file appearing as a continuous thick line?
  - Why are the other communications appearing as thin dotted lines?
  - What is the NetCDF file name which will be created?
- ➢ Run the application to generate the model output file.

Now, the spatial interpolation of the fields contained in this file will be performed by a pre-defined unit that you can find directly in the PrePALM toolbox (menu `File` => `Load algebra unit` => `Interpolation` => `Geophysics` => `DSCRIP.ALG`). The development of this unit was based on the CERFACS OASIS3 coupler interpolation routines.
This coupler, whose usage is widely spread in the climate community, has the advantage (from version OASIS4 on) to move the interpolation in the parallel executables of the models to be coupled. The interpolation can thus be done in parallel, although this is not yet possible with PALM. Moreover, OASIS4 is easier to setup and install on some machines because it does not require MPI2. For more information on grid to grid interpolation you may refer to the documentation of the OASIS coupler.

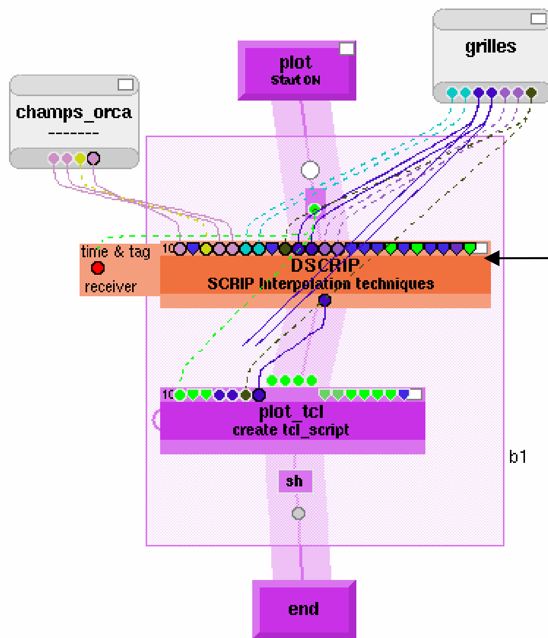
We have already the ID card of the file which we want to interpolate. However it was made for a file in a writing mode, the objects are `-intent IN`. We want now to access this file in a read mode. We have two solutions to redefine this file ID card. Either we copy the file `champs_orca.in` and we replace the `-intent IN` by `-intent OUT`, or we use a PrePALM utility which can create the files ID cards starting from existing NetCDF files. You do not have to do it now because this work was already done in order to simplify this exercise.

The interpolation unit works like this:

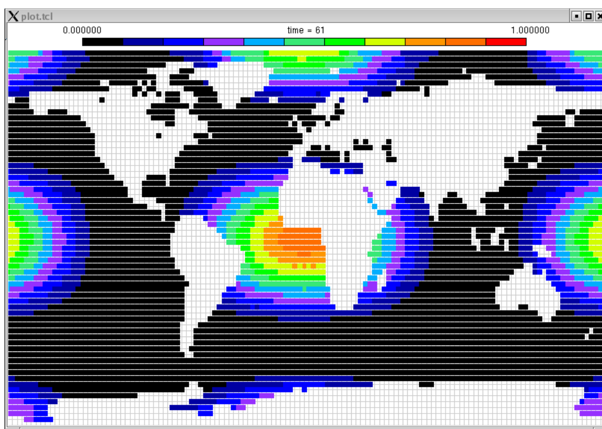
- It asks for the grid type, the grids, the connectivities of the two grids, one being regarded as the source and the other as the target.
- It asks for the field to be interpolated on the source grid.
- The different interpolation methods are also an input of this pre-defined unit. Most of the time, these values are hardwired.
- It returns the field interpolated on the target grid.
- For optimisation reasons, when the unit runs the first time, some data (which may take a long time to compute) are stored in a work file depending on the options given by the user. In the subsequent runs, the unit just reads the needed data from this file.



Fields computed on the ocean grid (orca 2)



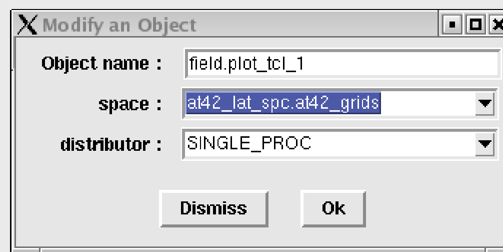
OASIS spatial interpolation



Interpolated field on the ARPEGE AT42 grid

Interpolate!

- In PrePALM open the file `session_13.ppl` which contains a branch calling the interpolation unit and the fields visualization utility. The file containing the fields on the orca grid is already connected in PrePALM. Notice that one will interpolate only a few time instances in this file (DO-loop).
- In the menu `Utilities` select `Generate id_card of files`, choose the file `at42_grids.nc` which contains the grids on which you will interpolate the fields (beside the at_42 grids, this file contains connectivities for the orca grid).
- The ID card which has been just generated by PrePALM is called `at42_grids.id`. Load it, and insert an instance of this file in the canvas (middle click in the canvas), enter the appropriate file name in the `filename` field.
- Create the communications between the file and each of the 2 units.
- Create the communication between the interpolation unit output field and the visualization unit. Before that, notice that the two plugs are yellow, indicating that it is thus necessary to define the output field space. For that, click on the interpolation unit output plug, then double click on the object selected in the categories window. A menu inviting you to modify the space is proposed to you; choose space `at42_lat_spc.at42_grids`:



- Run the application.

13.1 Summary of the main concepts

The main aim of this session was to teach how to access NetCDF files in a way that makes very easy to interchange I/O on files with coupling communications. If the user describe the file contents (NetCDF header) in an identity card (N.B. PrePALM provides a utility to create the id card of an existing NetCDF file), the output to a file can be implemented as a communication linking a `PALM_Put` to an input plug of a "*file unit*" and conversely, the input from a file can be implemented as a communication linking a `PALM_Get` to an output plug of a "*file unit*".

This mechanism could be easily extended to other self-descriptive file formats, as HDF5, GRIB, etc. Users wishing to do it are strongly encouraged to contribute.

In the application example, you have seen how to use the grid to grid interpolation unit from the PrePALM algebraic toolbox.

14 Session 14: Using a minimiser

In the predefined algebra toolbox, PALM provides some *minimisers* which may be useful to the user for some applications. Some minimisers are coded in “reverse communication”: they are adapted to PALM because they do not request a dedicated application process.

As an example, we will minimise a cost function of the form: $\boldsymbol{J}(\mathbf{x}) = \mathbf{x} \; \mathbf{B}^{-1}\mathbf{x}$. The gradient of this function is **grad** $\boldsymbol{J}(\mathbf{x}) = 2*\mathbf{B}^{-1}\mathbf{x}$. We will use the CGPLUS minimiser implementing a conjugated gradient algorithm.

We will need only three user units. The first one, named `init`, will give a first value of the function (a vector having all elements equal to 1). The second unit (`compute`) will take a vector in input and will multiply it by a diagonal matrix $\mathbf{B}^{-1}$ (the elements of the pre-inverted $\mathbf{B}^{-1}$ matrix are 1, 2, 3, ..., `ip_vectsize`) and it will return the $\mathbf{B}^{-1}\mathbf{x}$ result. The third unit (`result`) will just print out the result.

The gradient $2*\mathbf{B}^{-1}\mathbf{x}$ can be easily calculated just by multiplying $\mathbf{B}^{-1}\mathbf{x}$ by 2 with the algebra unit `DSCAL`. The expression $\mathbf{x} \; \mathbf{B}^{-1}\mathbf{x}$ is calculated with the scalar product from the `DDOT` unit. In our example, the analytical solution is $\mathbf{x} = \mathbf{0}$ to be compared with the result returned by the minimiser.

To simplify the communications all `PALM_Put/Get` calls are made without time nor tag (`PL_NO_TIME`, `PL_NO_TAG`).

The CGPLUS minimiser is an iterative process that works like this: starting from a first value of the function (f) and its gradient (G), it calculates a new point where the value of the function and its gradient must be computed to make a new iteration. For each iteration, the minimiser raises a flag telling if the minimization process has finished or if this processing should be continued. The criterion of convergence and the maximum number of iterations are inputs of the minimiser.

The algorithm will thus be a loop around the minimiser (do while) that will stop only when the convergence is reached or if the maximum iteration count is exceeded. The unit which gives the first value of the function will be launched outside the loop.

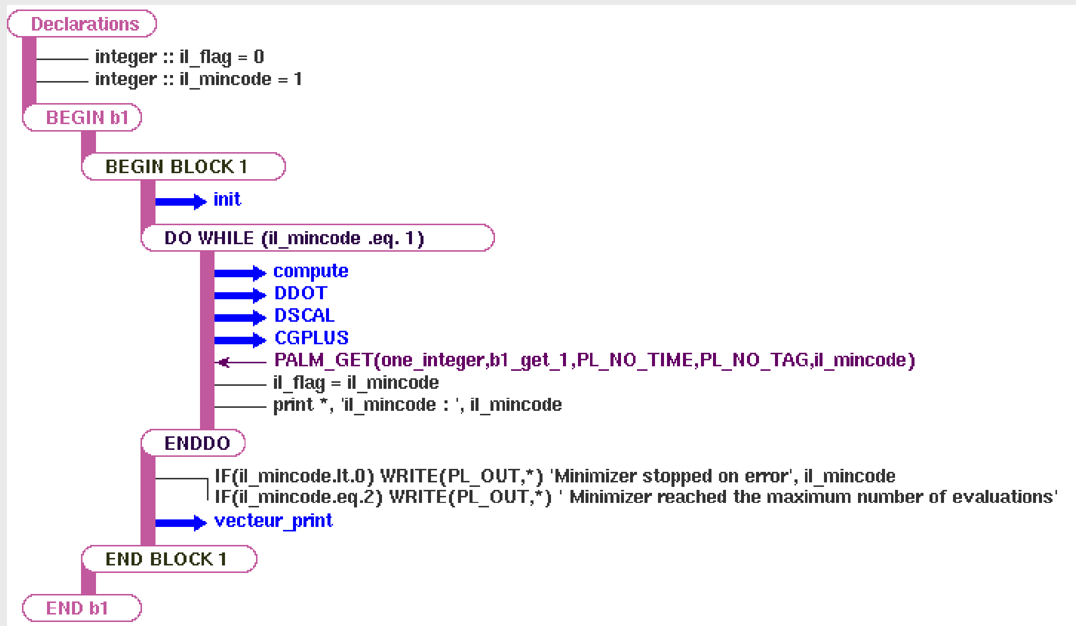
Minimize!

- Open PrePALM and define those constants:

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

- Load the 3 users units: **`init`**, **`compute`**, **`vecteur_print`**
- And the three algebra units: **`DDOT`**, **`DSCAL`**, **`CGPLUS`**

➢ Define the following algorithm:



➢ Create the communications:
- From the vector first value (object `first_guess` of unit `init`) to `compute` (`vector`), `DDOT` (`X`) and `CGPLUS` (`x`)
- From `compute` (`result`) to `DDOT` (`Y`) and `DSCAL` (`X`)
- From the `DDOT` result to (`f`) of `CGPLUS`
- From the `DSCAL` result to (`g`) of `CGPLUS`
- From the `CGPLUS` result (`x`) to `compute` (`vector`) and `DDOT` (`X`)
- From the `CGPLUS` result (`result`) to `vecteur_print`. It is also necessary to give a correct space to both of these objects (space defined to NULL). For this select the objects and edit them by double clicking above in the left window of PrePALM:

no	name	space
25	result.CGPLUS	vect_space.compute

- From the `CGPLUS` flag to the `PALM_Get` on the branch

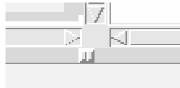
➢ For the other plugs, hardwire the following values:
- 2.0 for `alpha` of `DSCAL`
- 1 and 0 for `iprint1` and `iprint2` of `CGPLUS`
- 1.d-12 for `eps` of `CGPLUS`
- `il_flag` for `iflag` of `CGPLUS`
- 0 for `irest`, 2 for `method`
- `ip_iter` for `nbmaxiter`
- `ip_eval` for `nbmaxeval`
- `.false.` for `finish`

➢ Test your application

Let’s take advantage from the fact that in this application we have only 11 communications (this is still a small number, compared to the large number of communications that can occur in real applications), to show some of the nice features of the graphical user interface.

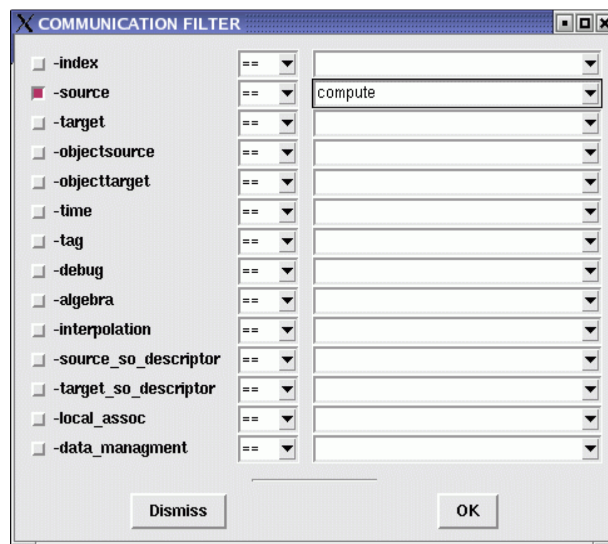
Select the communication category of PrePALM:

Make the window “attributes” larger with the slider:



A click on a single column title allows you to sort all communications by one of the attributes. Try it!

Also try the Filter button that opens the following dialog box:



This enables you to select a subset of the communication list based on one or more criteria. The example above will list the communications for which the source unit is `compute`. This function, which does not apply only to the communications, is very useful: for example when you have to modify an application developed by another person (or yourself if you did not use it for a long time) and to understand exactly what happens, or to examine the attributes, etc.

In the menu `Utilities` you have also some very interesting operations which may facilitate the repetitive actions. You can for example put the attribute `TRACK_ON` on all communications, in order to have more information in the log files on what PALM is doing. Usually the problems encountered with PALM originates from communications badly described in PrePALM (field time tag...) or badly coded in the units themselves, like the use of an object space or name different from what is declared in the ID card, or a description with time stamp turned ON, but sending with `PL_NO_TIME`, etc.

Finally, you can customize the graphical look of the canvas in the menu `Settings` => `Canvas settings...` => `Radius of plugs`, `Unit height`, `Branches width`, etc, etc.
Try also to close/open some unit representations by clicking on the top right white rectangle or to close/open all of them in a single move from the menu `Utilities` => `Units` => `Close all` or `Open all`

14.1 Summary of the main concepts

In this session you have learnt how to set up an application using a reverse communication minimiser. Since the minimisation is the base of many optimization procedures, this session is of particular interest for the users going to implement a data assimilation suite or an automatic design/ shape optimization application.
In the second part of the session you have learnt some useful practical tricks to make the use of PrePALM easier and quicker.

15 MPI-1 Mode

15.1 Introduction

The original PALM development is based on the MPI-2 standard. In addition to the functions of the MPI-1 mode, it exploits two functions of the MPI-2 mode:

- the dynamic launching of new processes (`MPI_Comm_spawn` and related functions),
- the client/server mechanism (`MPI_Comm_connect` and related functions).

To port PALM on a given computer, this has to be equipped with a MPI-2 implementation, if not complete, at least supporting these functions. The majority of the public domain distributions such as LAM/MPI, OpenMPI (not to be confused with Open-MP; cf. §4.5) or MPICH2 implement these functions and thanks to these distributions PALM has worked efficiently and has been validated in a fully public domain environment. Nevertheless, some proprietary distributions released by supercomputer manufacturers, and optimised on their computers, do not accept the MPI-2 functions. We can take as an example the IBM BLUE-GENE L supercomputer where it is completely impossible to install another MPI version, apart from that given by IBM.

To solve these porting problems on this kind of machines, we have developed a "light" version of the PALM coupler: this is what we call the MPI-1 mode of PALM. One can choose this version when installing PALM by activating a key on the automatic configuration system when installing PALM (cf. § 20.3 and `configure --help` for further details) and through the PrePALM graphical interface when generating the application service files.

The principle of this version lays on starting all the programs since the very beginning of the application, exploiting the MPMD mode of MPI-1. This mode in not part of the MPI-1 standard, but can be found almost everywhere on the various implementations of MPI-1. In this "extended" MPI-1 mode, many different executables can be launched at the same time, sharing the same MPI_COMM_WORLD communicator. We are, thus, in a MPMD configuration, although in pure MPI-1.

15.2 Restrictions at the level of the PALM coupler

In MPI-1 the PALM executables (units and blocks) cannot be re-launched many times during the simulation, so the loops and the other control structures have to be systematically encapsulated in blocks. We have seen in session 3 that being obliged to encapsulate in a block the loops containing some executables was not without consequences on the applications. As an example, if the programs do not free their memory, when launching a few consecutive runs, the machine's capacity may be overcome. Most of the programs are not conceived to work "in a loop" since this requires a programming effort which is not necessarily compatible with the original development design.

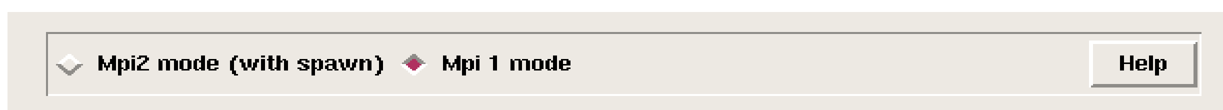
So, before choosing the PALM MPI-1 mode, one has to analyse his application to see whether the coupling can be done. For couplings such as fluid/structure interaction, where the codes are to be launched just once at the beginning of the simulation, this mode is perfectly adapted and with no loss of performances.

We have to point out that, for the parallel codes, with the MPI-1 mode it is necessary to use the `PL_COMM_EXEC` communicator instead of `MPI_COMM_WORLD`, since the `MPI_COMM_WORLD`

communicator concerns all the application executables, including the PALM driver. It is not the case with the MPI-2 mode for a program "spawned" by the driver which keeps a private `MPI_COMM_WORD`. In this latter case, the `PL_COMM_EXEC` communicator is a copy of the `MPI_COMM_WORLD` communicator.

15.3 Executing an application in MPI-1 mode

To install the PALM MPI-1 mode, one has to use the option `--with-mpi1mode` during the configuration (`configure`) *Cf.* § 20.3. This mode has also to be activated in the `Make PALM files` menu of PrePALM to create the appropriate service functions. One has to select the box corresponding to the required mode. A help button reminds the characteristics of this mode.



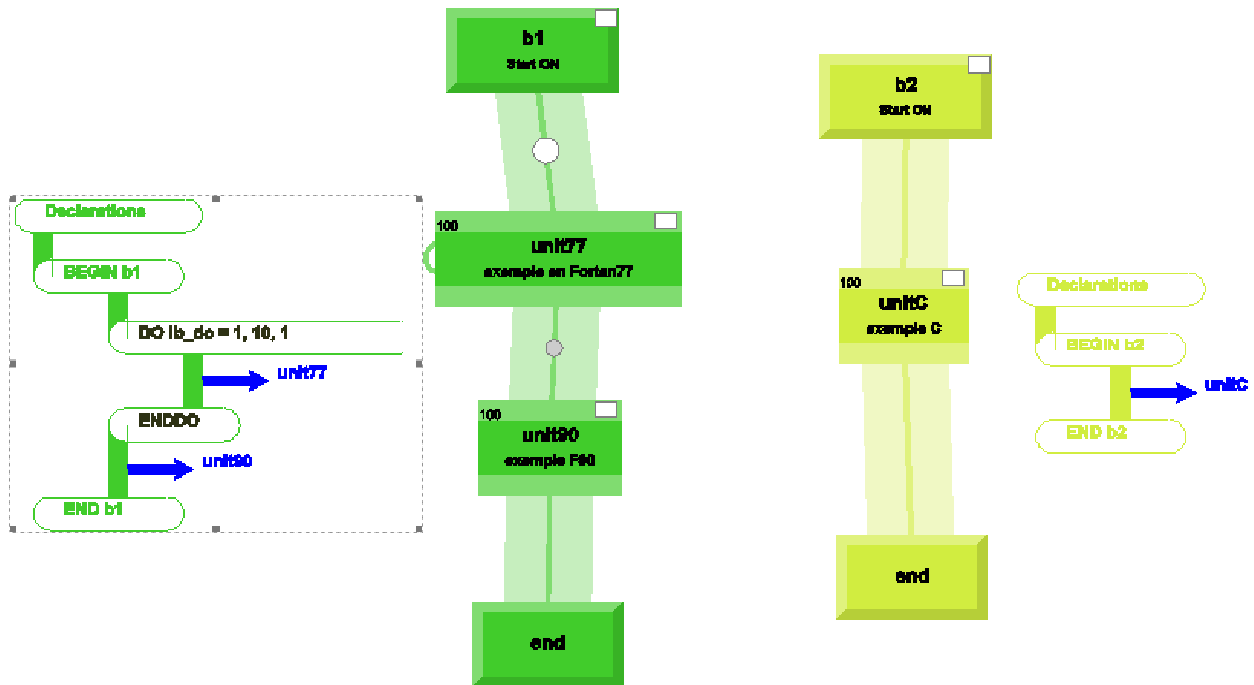
Pay attention: this choice must be totally coherent with the PALM library one uses. If one prepares the PrePALM files in MPI-1 mode, it is compulsory that the PALM library specified in the PALMLIB of the `Make.include` file ends with the `_mpi1mode` extension. For further information on the PALM compilation, see the corresponding section (cf. Chapter 20).

Together with the service files, a new shell script (`run_mpi1.sh`) has been created by PrePALM. This file adopts a syntax for the `mpiexec` command which is compliant with LAM/MPI, MPICH2 or Open-MPI. You are possibly lead to edit this script to adapt it to your local MPI distribution. If in MPI-2 mode you simply had to start the driver `./palm_main`, in MPI-1 mode you have to invoke `./run_mpi1.sh`.

15.4 An application example in MPI-1 mode

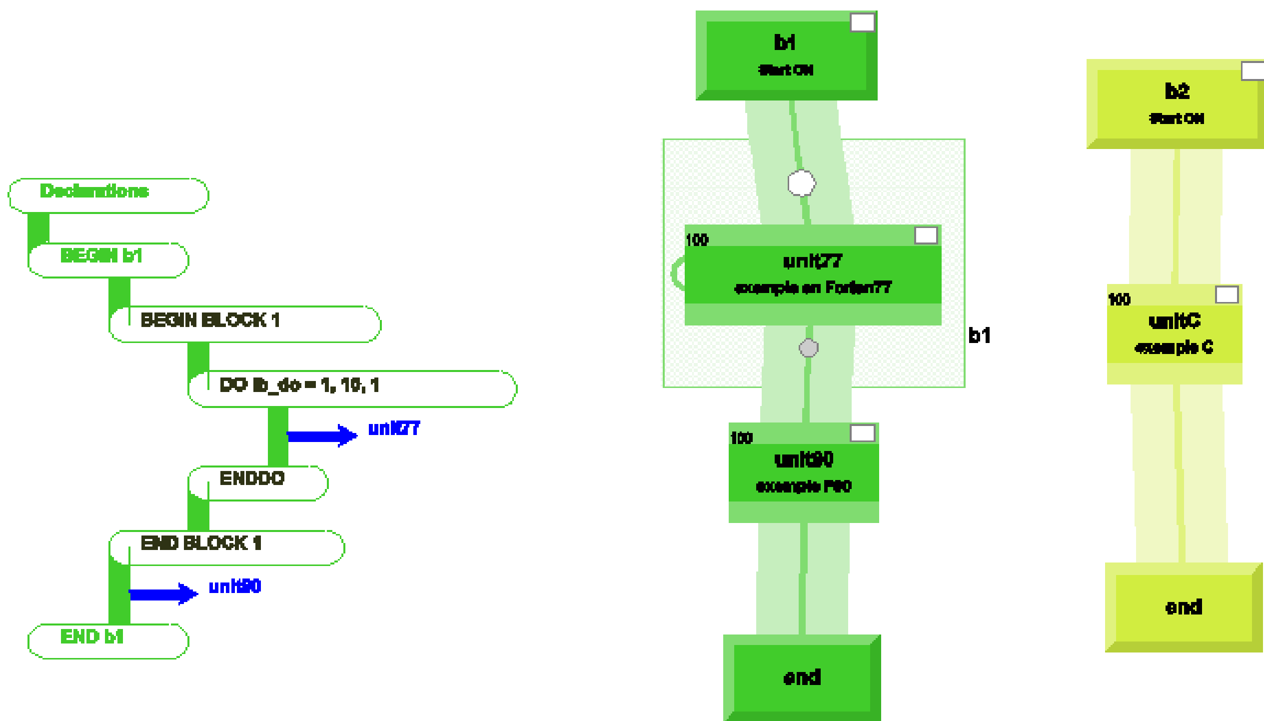
Let's go back to the units of the section 2 of this textbook (`unit 77`, `unit 90` and `unit C`) and examine the differences when executing them in MPI-2 or in MPI-1 mode.

Let's start from the following application in PrePALM:



In MPI-2 mode, this application requires 3 executables (`main_unit77`, `main_unit90` and `main_unitC`) in addition to the PALM driver. Since `unit90` is on the same branch as `unit77`, the first will work after the second and will reuse the processor and the memory resources of `unit77`. `Unit77` is launched 10 times in a loop: every time the program is loaded and executed and it reuses the same resources. This PALM application needs 3 processes to run in MPI-2 mode, one for the driver and one for every branch. It can also easily run on two processes, since there is no communication between the two branches and therefore no deadlock risk.

In MPI-1 mode, this application couldn't work as it is, since `unit77` is in a loop. The only way for it to run is to put a block around the loop. Just like this:



In this case we still have three executables : `main_block_1` (replacing `main_unit77`), `main_unit90` and `main_unitC`. In MPI-1 mode these three executables are launched since the beginning of the application by the `run_mpi1.sh` script containing:

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

So we need 4 processes to launch this application in MPI-1 mode. Keep in mind that the `unit90`, although it is loaded in memory since the beginning of the application, is not really executed, because it waits for the starting order given by the PALM driver, which is going to send it only when the loop around `unit90` is complete. PALM grants that the application results in the MPI-1 mode are identical to the MPI-2 mode.

For this application, it would be even better to gather `unit90` in the block in order to have only three executables: `main_block_1` (including `unit77` and `unit90`) and `main_unitC`. Thus the launching command would be:

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

In this case, the number of the process needed for this application would be just three. In order to pass from three processes to two, it would be necessary to pass `unitC` to the `b1` branch and to include it within the block.

Pay attention: It is important to keep the executables' launching order as it is given in the `run_mpi1.sh` script. PALM counts on it to create the MPI communicators of the different executables; the slightest inversion could have catastrophic effects.

Many PALM applications can run in "extended" MPI-1 mode with some little modifications of the algorithm defined in PrePALM. Unfortunately, the consequence is often a "waste" of resources. Some applications, where the codes run in loop with no possibility to put them into blocks, are not possible in MPI-1 mode. As a consequence, the MPI-2 mode remain the most general case of the dynamic coupler PALM for it is much more flexible.

15.5 Summary of the main concepts

In this section you have learnt how to compile and use a “restrained” version of PALM that does not need MPI-2. This is suitable on some particular supercomputers where the available MPI-2 distributions cannot be installed or would cause a major loss of performances.
The IBM Blue Gene L is one of these machines, therefore a section has been dedicated to the use on this specific MPP platform.
Your attention has been driven to the limitations that this “degraded” mode imposes to the full flexibility of PALM, making the MPI-2 mode the recommended way to install and use PALM.

16 Grid-based Interpolation with CWIPI library

16.1 General information

The CWIPI library developed at ONERA under LGPL license (sites.onera.fr/cwipi/) allows exchanging information between parallel codes based on unstructured meshes. CWIPI, which is part of the OpenPALM distribution, can be used alone as a coupler or via the OpenPALM coupler to take advantage of both the PALM and CWIPI libraries as well as the PrePALM graphical interface.

The great advantage of CWIPI is that it allows to exchange coupling fields on different meshes on the source and target side by interpolating the quantities on the fly. CWIPI's functionalities are based on a 3D spatialization of the data; for CWIPI a coupling field is associated with a mesh described as unstructured elements. It should be noted that a structured mesh can be described as an unstructured mesh if you encode an interface. CWIPI also has the advantage of almost seamless parallel code management based on domain decomposition; its communication scheme is well suited to couple massively parallel codes.

16.2 The bases of unstructured meshes in CWIPI

In CWIPI, meshes are based on the definition of basic elements such as:

- segments for 1D elements,
- triangles, quadrangles or polygons for 2D elements,
- tetrahedra, pyramids, prisms, hexahedra or polyhedra for 3D elements.

All these elements are described in a 3D Cartesian reference frame which must be common to the various codes. CWIPI needs geometrical information that must be described in memory through the use of specific primitives, it requires constraints on the form and type of tables to be communicated to it. It is unusual to provide internal data structure of the solvers to CWIPI since::

- in the code these data structures are not necessarily described in the CWIPI format,
- the coupling zones are generally local, e.g. an exchange can take place on a sub-surface or a sub-volume of the mesh.

16.3 First steps with CWIPI under OpenPALM

In this example we will perform a coupling by interpolating a 2D field between two surface meshes. Go to the session_PCW/base directory, Here (in the polyg.f90 file) is the first code we will couple, starting with its ID card:

```
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: A new keyword describes a CWIPI coupling, the user can describe several CWIPI couplings in the same code. Associated with this coupling are CWIPI objects (13) that identify the fields to send or receive. As for spaces and PALM objects, the names given to CWIPI couplings and objects are internal to each code. In our case, the exch1 object has the INOUT attribute which means that we will send and receive this object in the code. As for PALM, the instrumentation of each code is independent of the other coupled codes; it is by describing the application in the PrePALM graphical interface that we will connect a "coupling" of a code to another code. Now let's look at the FORTRAN code:

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

19: A cwipi Fortran module is made available to the user, this module contains the generic constants for the call to CWIPI primitives. Next comes the declaration of code variables:

```
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
```

We will define a mesh based on a collection of 11 points (nvertex) of a 3D Cartesian landmark, these points will allow to define 5 elements (nelts). We will come back later on the tables coords, connecindex, and connect which will be used to define the mesh size.

```
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
```

As a prelude to CWIPI exchanges, via a PALM_Get, the code asks how to initialize the array of fields it will send, with a possible choice between sending an array containing the X-coordinate or the Y-coordinate (as described in the identity card of the unit (7)), sending a field in which the values of a mesh coordinate is used to carry out simple tests with a quick visual check of the values of a grid coordinate.

```
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)
```

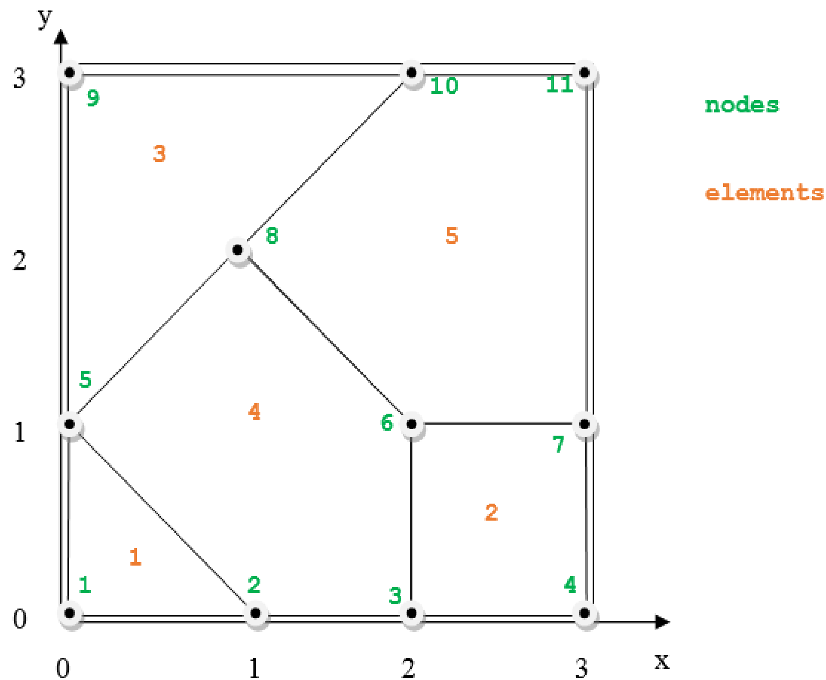
All CWIPI primitives interfaced for OpenPALM start with PCW_. The code must call the PCW_Init primitive (46), then it defines some of the coupling characteristics "cpl1" (defined in the ID card) via the PCW_Create_coupling primitive (50). In our example the code informs cwipi that:

- it is parallel code with a domain decomposition (51),
- we are going to manage 2D elements, it will therefore be a surface coupling, even if for CWIPI the surfaces are defined in a 3D reference frame (52),
- we assume a local geometric tolerance corresponding to 10% of the size of a mesh for the search of points to be located in this mesh (53), we will see later on the interest to act on this tolerance,
- the mesh will not move during the simulation (54),
- the exchanged fields will be defined on the vertices of the elements (55),
- field control outputs are done at each iteration with a definition of the output formats in ASCII "Ensight" format, readable by the Paraview visualization software.

The three tables necessary for CWIPI to call PCW_Define_Mesh are then constructed:

```
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
```

The coords table (62) contains the coordinates of the nodes of the mesh (see figure below), it is a double precision 1D table of size nvertex*3; the coordinates are interlaced (x1, y1, z1, x2, y2, z2,..., xn, yn, zn). The table of integer connecindex (64) of size (nelts +1) contains the information that allows to know which type of element is described one by one in the connectors table, the first index of this table is always 0, then we cumulate the number of vertices of the next element. In our case we have 1 triangle, then 2 quadrangles, then 2 surfaces with 5 vertices. Elements should always be described in ascending order of vertices. Finally the table connec (65), of connecindex(nelts+1) size contains the description of the elements one by one, in this example we took care to separate the 5 elements by spaces for more readability.




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

All mesh characteristics are communicated to CWIPI via the PCW_Define_mesh primitive (67).

Warning: in CWIPI, tables passed to the PCW_Define_mesh primitive are "mapped" in memory, without copying. In the step PCW_Define_mesh CWIPI only keeps the memory addresses of the sent tables, it only processes this data when it triggers the localization algorithm which, if not explicitly requested, only occurs during the first exchange on this coupling. Therefore, it is important to not destroy or modify the arrays connec, coords and connecindex before the CWIPI coupling ends.

```
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
```

The table values (77) containing the field to be sent is initialized either to the X coordinate value or to the Y coordinate value (choice of the user in PrePALM). We then proceed on to the data exchange phase (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 can send and receive simultaneously with a PCW_Sendrecv, the integer stride (91), here equal to 1, is used to send several fields in the same table. If stride is greater than 1, the table must be filled by interlacing the fields, for example for a stride of 3 with fields u, v and w, we would have (u1, v1, w1, u2, v2, w2,..., un, vn, wn) in the values and localized tables. This code will be paired with another one that will also define a mesh. If the meshes of the two codes are coincident (that they overlap exactly the same geometric surface, all points of a target mesh will be located in the source mesh, otherwise CWIPI will return the number of points that have not been located in the nNotLocatedPoints variable (98). A special treatment of the non-localized points may therefore be necessary. Contrary to our example, in a coupling between real codes the data sending/receiving phase is usually called several times.

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

The two last CWIPI primitives allow to destroy the coupling and terminate the CWIPI session (101 & 102). To test this "code", it can be coupled with itself, just launch two instances of this PALM unit in two different branches.

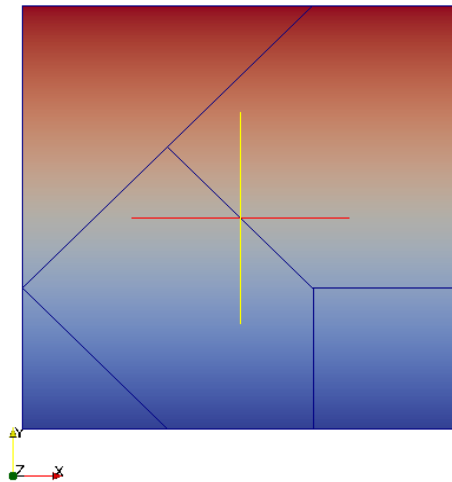
- Go inside the session_PCW/base directory
- Create two branches with PrePALM.
- Load the polyg.f90 unit ID card.
- Launch an instance of polyg in each of the two branches.
- For the PALM object coord_id, choose (right click on the plug) 1 for one of the codes and 2 for the second, so the fields sent by the two instances of the the code will be different (x or y coordinate).
- You must now specify which code communicates via CWIPI with which other, click on the CWIPI zone of one of the units, PrePALM offers you to insert a coupling, choose the only coupling that it offers to you.
- A thick gray line linking the two units appears, click on this line and insert the suggested communication.
- Generate service functions, compile and run the application.

In our case, we asked for graphical outputs (Output frequency fixed to 1 of the PCW_Create_coupling), these outputs are stored in a subdirectory named cwipi containing a sub-directory by code and by object name. It is recommended to set this output frequency to 0 once the coupling is in place to avoid time-consuming disk access in computation time.

To view the result of the coupling with the paraview graphics software, go to one of the subdirectories cwipi and launch the command:

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

Once you press the Apply button of paraview you will be able to see the field received or the field sent by CWIPI:



We will now couple the polyg code with another code that defines a different mesh size.

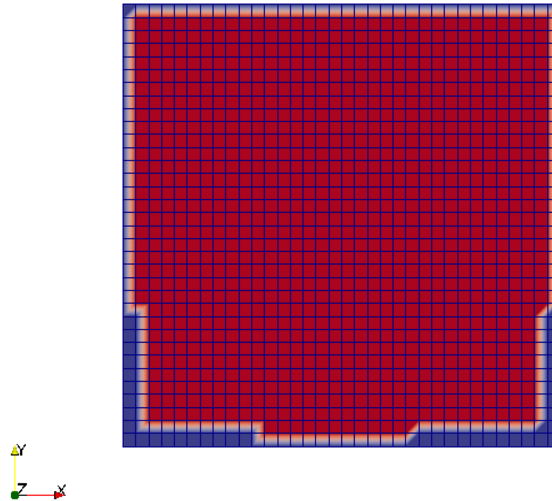
PCW Exercise Base 1

Open the rectangle.f90 file, answer the questions by looking at the Fortran code.
Questions:

- What type of CWIPI mesh is defined in this code, 1D, 2D or 3D?
- How many nodes for this mesh?
- How many elements?
- What types of elements?
- What portion of the 3D space is covered by this mesh?
- What is the field sent by rectangle?
- What happens if points are not located?

Replace one of the two polyg instances with a rectangle and test.

Rectangle code warns you that it finds 175 unlocated points. Go to the cwipi/cpl1. cpl1_rectangle_polyg subdirectory, and run paraview. By viewing the "location" field you will see which rectangle points are not located in polyg code:



Located points by CWIPI

When CWIPI detects un-located points, it creates a "location" field in the graphical outputs, the localized points have a value of 1 and appear in red with paraview, the un-located points have a value of 0 and appear in blue. The presence of un-located points is due to the fact that the domain size defined by the rectangle unit is larger than the one defined by polyg, with 2 additional rectangle meshe cells all around. The graphical output shows the effect of the geometric tolerance (fixed at 0.1 in polyg. f90); however, with this tolerance some rectangle points outside the polyg mesh are still localized, and other points remain unlocalized. We observe that this geometric tolerance is local, it is relative to each mesh cell. Depending on whether the nearest polyg element is larger or smaller some points are located or not. Now try to act on this tolerance to:

- not locate any points outside the polyg mesh,
- then locate all the points.

The choice you make to deal with these non-localized points in your couplings will certainly depend on the problems treated, but one thing is certain: you must always check and treat the non-localized points when coupling with CWIPI.

It should also be noted that the value of the geometrical tolerance conditions CWIPI's calculation time for searching for neighbours, the lower the value, the less time CWIPI will spend there. In our example we could go as far as a very low tolerance (0.0001 for example), a zero tolerance leads to not locating anything. In our example the two meshes are inscribed in plane Z=0, if the meshes follow a 3D shape like a sphere, it is probable that a too low tolerance would lead to many un-located points, the choice of the value of the geometrical tolerance is therefore a question of experience, in general we start with a value of 0.1 and then we adjust according to the obtained

results, a good way of proceeding is to read the geometrical tolerance in an input file to not recompile the code for each test.
We will now move to a parallel version of the rectangle code. To simplify the example, the decomposition of the domain is carried out only on the X axis, each process will have the same number of elements defined by default or in a rectangle_par.mesh input file. To view the differences between parallel and sequential code, open both files with the tkdiff utility:

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

You can see:

- the include of mpif.h to call MPI_Comm_rank and MPI_Comm_size primitives,
- re-calculating the local coordinates xmin and xmax on each domain,
- that none of the PCW_ primitives have been modified.

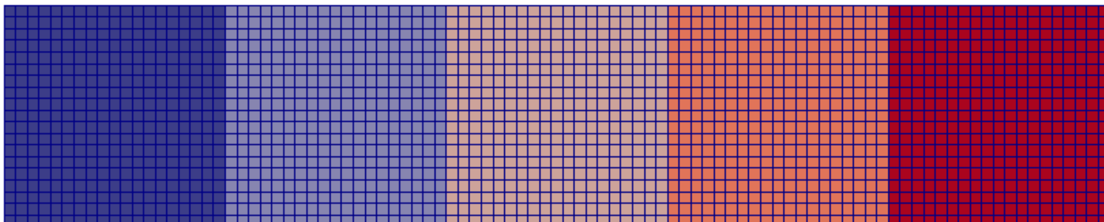
PCW Exercise Base 2

- Build two branches and run the rectangle and rectangle_par units.
- Give 5 processes to rectangle_par.
- In settings -> palm execution settings give a sufficient number of processes for the application.
- Build the two mesh definition files for example like this:

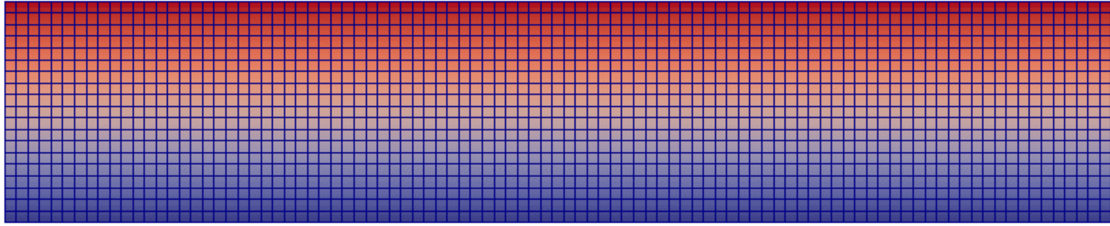
rectangle.mesh:	rectangle_par.mesh:
0. xmin	0. xmin
5. xmax	5. xmax
50 nx	20 nx
0. ymin	0. ymin
1. ymax	1. ymax
10 ny	20 ny

- launch the application and display the results of rectangle_par with paraview.

With these settings for meshes you should get these graphical outputs in the output directory of the parallel code:



Domain decomposition on the 5 processes (field partitioning) a color is assigned to each process.



Received and interpolated fields from rectangle code (field R_C0_exch1. exch1_co)

As CWIPI works with meshes in a common coordinate system for all processes, the instrumentation is almost transparent between a parallel code and a mono-process code. Each process defines only the part of the global mesh that it knows. The locating phase, carried out during the first exchange, allows CWIPI to know where to find the information to interpolate fields (what process and what element for each point to locate). Once this is done, CWIPI interpolates the fields in the mesh by assigning weights relative to the baricentric coordinates of the point concerned in the localized mesh. The operation of locating and calculating baricentric coordinates is carried out only once, which makes it possible to go much faster for subsequent exchanges if they are done in a loop, which is generally the case for multi-physic couplings. The localization phase, more expensive than the interpolation and exchange phase, is nevertheless optimized in CWIPI as it relies on an octree algorithm subdividing recursively the 3D search space. If the field values are given on the mesh nodes (CWIPI_SOLVER_CELL_VERTEX) the field is interpolated, if these values are given at the cell center (CWIPI_SOLVER_CELL_CENTER) the field is not interpolated.

Most of the PCW_ primitives are collective operations so synchronizing between the processes of the same code and the other processes of the code with which information is exchanged. So we have a quite different functioning compared to the primitives PALM_Get/Put. With CWIPI, for example, it is not possible to send information between two codes that would run one after the other on the same branch, as you can do with PALM_Get/put.

16.4 A more complete exercise

This session uses a simple example to give a hands-on experience of the CWIPI primitives related to interpolation of data on unstructured meshes. These primitives start with the prefix PCW and make calls to the CWIPI library.
In this exercise two codes, one in fortran (*fortran_surf.f90*) and one in C (*c_surf_coupling.c*) exchange data and perform interpolation on a mobile three-dimensional mesh of the following shape (refer to Figure 1 below):

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

The z coordinate is subject to a temporal oscillation of amplitude *ampl*, of frequency *freq* (omega = 2.pi.freq) and of phase shift *phi*. These quantities are defined separately for each code and are stored in the configuration files *dataC.dat* for the C code and *dataF.dat* for the Fortran code.
These files contain the following data:

```
1        ! initial iteration
10       ! final iteration
0.10d0   ! frequency
0.012d0  ! amplitude
0.1d0    ! phase
0.1d0    ! geometric tolerance
```

Note that the time step is 0.1

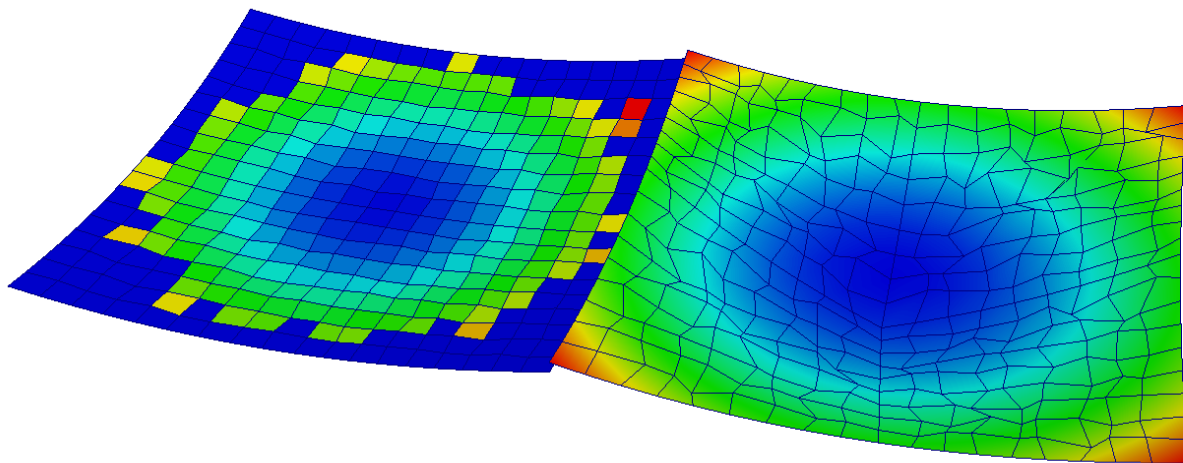


Figure 1: Exchange surface used by the simulations: left=Fortran, right = C

The mesh defined in the Fortran code is *cell centered*, i.e. the computed and exchanged quantities are defined at the centre of each cell. The mesh in the C code is of *cell vertex* type, i.e. the variables are localised on the mesh nodes. The data is exchanged via a crossover *send/receive* command, which grants symmetric instrumentation and execution of the codes. The exchanged data fields are of analytic type, so it is easy to cross-check the obtained results. The codes exchange a scalar field whose values are the z coordinates of the nodes or cells.

The user guide takes a 4 step approach through four exercises in order to get used gradually to the important concepts of the PCW_ primitives.
The first exercise (the longest and most important) describes how to set up the communication between the Fortran and C codes, while executing only one time step.
The second exercise still keeps running on one single time step, but illustrates how to manage non localised points detected by the coupler.
In the third exercise, time iterations will be performed assuming that the coupling surfaces remains static.
In the last exercise, a solution is found for a coupling problem with moving surfaces.

The useful primitives are:

- Initialisation of the coupled application
  - PCW_Init
  - PCW_Finalize
  - PCW_Create_coupling
  - PCW_Delete_coupling

- Mesh definition
  - PCW_Define_mesh
  - PCW_Update_location
- Data exchange and monitoring
  - PCW_Sendrecv
  - PCW_Get_not_located_points
- Application control
  - PCW_Dump_application_properties
  - PCW_Dump_notlocatedpoints
  - PCW_Dump_status

The functions are described in detail in the reference section at the end of the user guide (Chapter 25).
The last paragraph of this chapter will present the remaining PCW_ function for advanced use of CWIPI.

16.5 Definition of the coupling in PrePALM

For all exercises in this chapter, the PrePALM environment is identical. Let's build it now.

To start with, have a look at the ID cards of the codes. They are given at the beginning of the source code file. As for every OpenPALM application, you can find the the PALM_UNIT keyword, where you can define the unit name as well as some parameters which should be quite well-known by now. One can see 6 objects (PALM_OBJECT keyword) which are received by the units. These objects correspond to the data to be read from the configuration files dataC.dat and dataF.dat. The variable geom_tol will be used by CWIPI as geometric tolerance during the mesh localisation.

The keyword PALM_CWIPI_COUPLING is completely new and declares a CWIPI coupling environment for the units. These environments will be associated with exchange objects (PALM_CWIPI_OBJECT keyword). These exchange objects can be for reception only (intent IN), for transmission only (intent OUT) or both (intent INOUT). It is possible to create several couplings in one unit. For example a CFD code can have primitives to exchange data simultaneously with a thermal conduction code, a radiation code, etc.

C code ID card:

```
/*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
*/
```

Fortran ID card:

```
!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
```

To build the application, let's load the ID cards in the preface of the codes' source files.

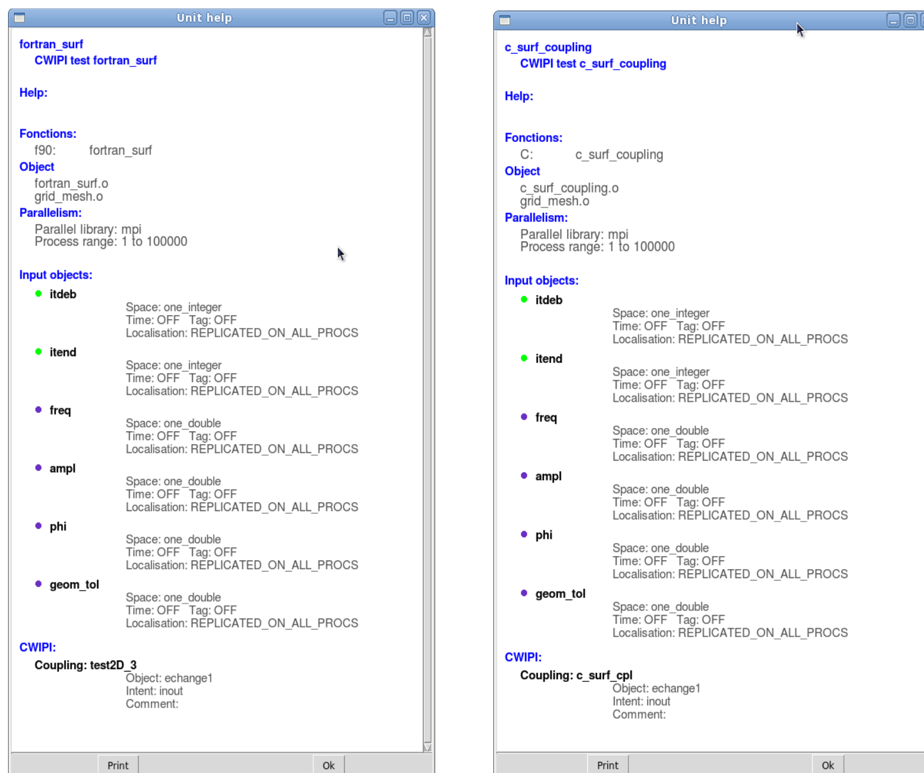


Figure 2. Unit ID cards loaded in PrePALM.

We will now create two branches b1 and b2 which are going to call the units fortran_surf and c_surf_coupling respectively. One can notice in Figure 4 that there is a small CWIPI rectangle in the lower left of the units which reminds you that these units can communicate via the CWIPI library. Both units are parallel. The first unit shall use 4 processes while the other one shall use one (caution: the units support parallel computing, but nevertheless they cannot be executed on an arbitrary number of processes, the code checks that the process count is a power of 2). Finally, set the working directory to ./CODEF for the Fortran code and to ./CODEC for the C code in the unit property dialog (Figure 3) Make sure these folders exist in the application's execution path. Creating separate working directories for the two codes is especially useful for complex codes which read and write huge amounts of data in their directory.

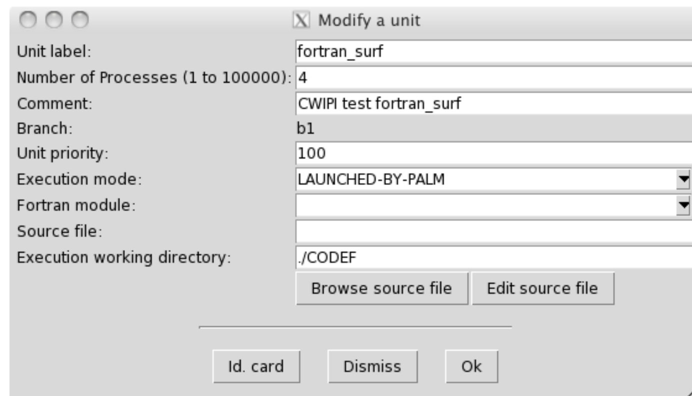


Figure 3. Property editor for th Fortran unit.

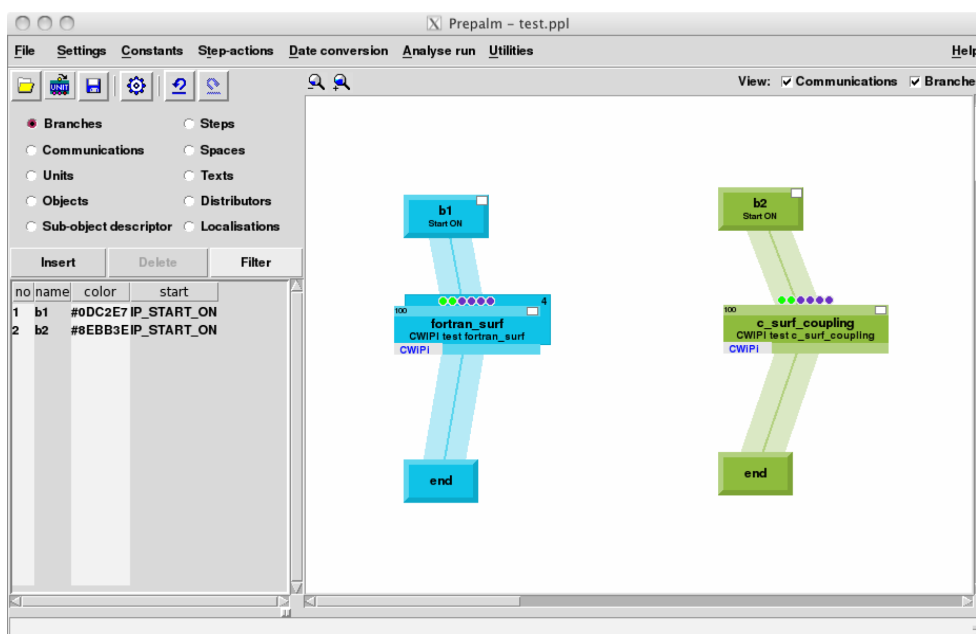


Figure 4. PrePALM canvas with two branches and the two units.

You can create a CWIPI connection between the two units by clicking on the CWIPI rectangle in one of the units. PrePALM then proposes to insert a coupling between the two units via the coupling environments described in each unit ID card. A line between the units symbolises the CWIPI communications. If you click on this line, you can open a new menu to match the coupling objects in the 2 units. (Figure 5)

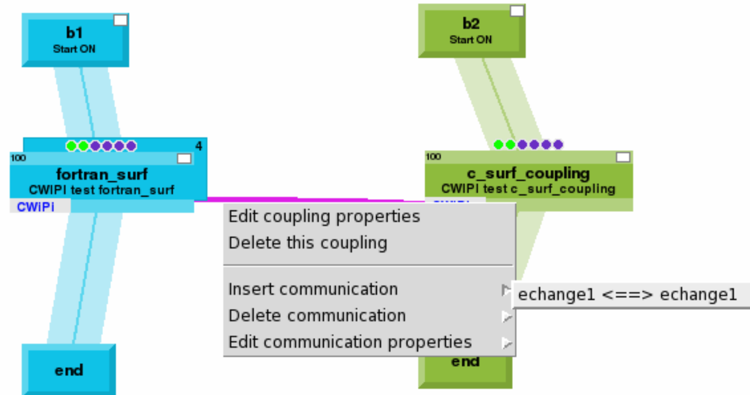


Figure 5. Matching of coupling objects between the two units.

Now declare the variables read from the configuration files and received in the units via PALM_Get. Insert a Fortran region which reads these parameters from the configuration files: dataF.dat (in the Fortran unit's branch) and dataC.dat (in the C unit's branch) as shown in Figure 6.

Figure 6. Source code of the branches

To create a direct communication between the variables in the branch fortran sections and the units, use a “hard-wired” setting of these objects (right click on the corresponding plugs and select the variable).
The PrePALM application is finished. You just have to generate the service files as for a regular OpenPALM application (menu File->Make PALM files). Pay attention to selecting MPI-1 mode in the dialog.

16.6 Exercise 1: initial instrumentation

The stubs of Fortran and C code in the folder lack any function call to the coupling primitives. As the instrumentation of both codes would be long and repetitive in the scope of the training session, the user shall choose to instrument either the Fortran or the C code according to personal taste. The explanation in the user guide is based on the Fortran code. The sections to be completed in the source files are marked with “To fill” --> “End to fill”. The PCW_ primitives can be found at the end of the user guide.

The different steps of instrumentation are:

- initialisation of the coupling
- creation of the coupling environment
- definition of the mesh support
- data exchange
- processing of the received data
- deletion of the coupling environment
- finalisation of the coupling

16.6.1 Initialisation of the coupling

The initialisation is done via the PCW_Init primitive. The PCW_Dump_application_properties primitive can be called at any time to print the properties of the CWIPI environment to the log files, this should look like:

```
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 Creation of the coupling environment

The creation of the coupling environment is done via the PCW_Create_coupling primitive. Remember, the codes are parallel, the exchanged data has two dimensions and the mesh is static. The following variables are defined in the Fortran code and can be used directly in the function call:

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

It is important to keep the variable cl_coupling_name during the whole CWIPI session, since it contains the name of the coupling environment. The name is the same as in the ID card and must be passed to several primitives for all operations on this environment.
As a start, use a geometric tolerance of 0.1

16.6.3 Definition of the mesh support

In CWIPI, a coupling is currently uniquely linked to a mesh. The mesh is attached to the coupling via the primitive PCW_Define_mesh which is given as arguments the coupling name, the node count (nvertex), the cell count (nelts), and the tables of coordinates (coords) and connectivity (connecindex and connec).
Take your time to find out how the coordinate table is built. Its size is 3*nvertex (CWIPI always uses a three-dimensional Cartesian coordinate system). The x, y, z coordinates of node n° i are interlaced and stored in the following way:

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

The table connecindex of size nelts +1 indicates the number of vertices for each element. The element i is composed of n vertices:

n = connecindex(i+1) – connecindex(i)

The table connec contains the indices of its vertices (1 based array).

If the mesh contains various element types, they must be sorted in the following order:

- linear elements : bars
- surface elements : triangles -> quadrangles -> polygones
- volume elements : tetrahedra -> pyramids -> prisms -> hexahedra

The internal connectivity of the elements can be seen in Figure 7.

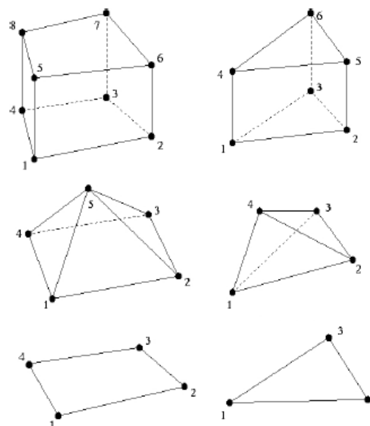


Figure 7: Element ordering

One can also define polyhedra. This is done via the function PCW_Add_polyhedra. The definition of polyhedra is done separately since it is more complex. The description of polyhedra contains two parts:

- description of the connectivity of the polygon faces in the polyhedron (same definition as polygons in PCW_Define_mesh)

- description of the connectivity between the polyhedron and its composing faces. Beware, this connectivity has an orientation :
  - "numf" if the normal vector of the face points towards the outside of the polyhedron (with numf = face index)
  - "-numf" if the normal vector of the face points towards the inside of the polyhedron (with numf = face index)

16.6.4 Data exchange

On one single coupling entity, which means a single mesh, you can perform several exchange operations which may correspond to various variables or different time instants of the same variables. The cross exchange is done via the primitive PCW_Sendrecv which requires the following arguments:

- name of the coupling context
- number of interlaced data fields in the exchange. The data is interlaced in a similar way as the coordinate table: in case of the exchange of 2 interlaced variables, the index (i-1)*2 + 1 designates the first variable value at the vertex i if cell vertex mode is configured, in element i if the mode is cell centered. The value of the second variable can be found at the index (i-1)*2 +2.
- the current iteration time step and the corresponding time value for visualisation
- the names and address/reference to the data fields to be sent and received

Besides the received data fields, the primitive PCW_Sendrecv returns the number of non located points. Furthermore the error code tells whether the data exchange has encountered a major problem or not. This error code can be interpreted by the function PCW_Dump_status which writes the result (success/failure) of the exchange into the log files.

16.6.5 Processing of the received data

In this example, the received data is simply written to Ensight data format for graphic rendering with Ensight software. The source code is already written for this part, so nothing remains to be done.

16.6.6 Deletion of the coupling environment

The primitive PCW_Delete_coupling deletes the coupling specified by the coupling name, and finally the primitive PCW_Finalize will close the CWIPI session and terminate properly the active CWIPI environment.

16.6.7 Running the application and analysing the results

Once the instrumentation completed, compile the project with the command *make*. If you are in PALM MPI1 mode the application is launched via a script *run_mpi1.sh* created directly by PrePALM:

```
#!/bin/sh
```

```
mpiexec -np 1 ./palm_main : -np 1 ./main_c_surf_coupling : -np 4
./main_fortran_surf
```

If you are in PALM MPI2 mode, simply run the command:

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

Execute this script. Check that everything has run correctly, especially analyse the file *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 **************************
```

The file *palmdriver.log* informs you that the session has terminated normally and that there has not been any communication using the PALM library. Indeed, the hard-wired data setting done by right clicking on the plugs are not part of this summary, neither are the communications with the CWIPI library.
The Fortran code has created files for the visual rendering of the received data which are stored in the folder ./CODEF/. Furthermore CWIPI writes the exchanged fields (received and sent) as well as the partitioning of the domain into the subfolders:

- ./CODEF/cwipi/ test2D_3.c_surf_cpl_fortran_surf_c_surf_coupling/ for the Fortran code
- ./CODEC/cwipi/ test2D_3.c_surf_cpl_c_surf_coupling_fortran_surf/ for the C code

For visualisation of the fields, you can use the paraview software. Note that 4 files must be loaded for the Fortran code executed on 4 processes (CHR_0000.case to CHR_0003.case). These are the files created by the Fortran code, whereas the CWIPI output data is grouped into a single file.

Figure 8. Exchange surface with additional information (left to right)
- field received by Fortran (CHR_000.case)*
- Fortran partitioning (from CWIPI output files)
- filed sent by the C code (CWIPI output files)

For further exercise and a better understanding of the exchange mechanisms, try changing the variable name and try transferring a vector of 2 or more components (stride >= 2).

16.7 Exercise 2: detection of non located points

Take the previous coupling application and increase the amplitude from 0.012 to 0.015 in the file *dataF.dat*. Big surprise when you look at the output files (CHR_*.case): the data field is completely messed up. This is due to the presence of non-located points which can be observed in the CWIPI output files (Figure 9).

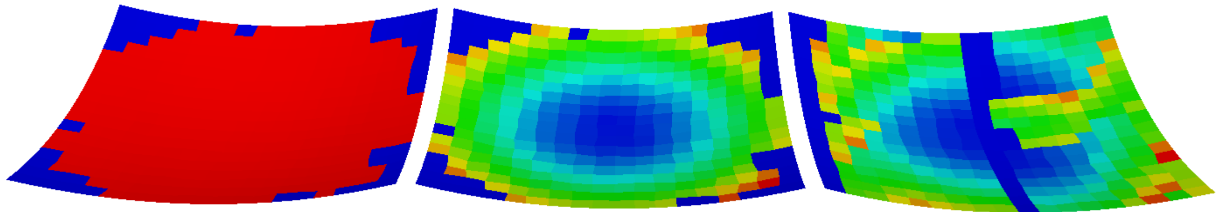


Figure 9.Fortran exchange surface (left to right):
- located points (red) and non-located points (blue) from CWIPI output
- correctly located data in CWIPI output files
- output of Fortran code, perturbed field

After a data transfer it is very important to make sure that all point have been located correctly on the source mesh. For this purpose, one can check the argument *not_located_points* returned by the *PCW_Sendrecv* primitive.

The primitive *PCW_Dump_notlocatedpoints* writes the indices of the non-located points into the OpenPALM log files. Furthermore, the primitive *PCW_Get_not_located_points* allows the program to retrieve the indices of non-located points via an integer array whose length has been set in accordance with the number of non-located points. The PCW_Sendrecv primitive only returns the values of located points leaving out all non-located points, therefore the indexing of the received data array is impacted. For clarification, if one considers a data exchange on 5 nodes and the third node could not be located, the array returned by PCW_Sendrecv contains:

- index 1 : value at node 1
- index 2 : value at node 2
- index 3 : value at node 4
- index 4 : value at node 5
- index 5 : 0

The existence of non-located points may not be a problem for the coupling algorithm, but it is still absolutely necessary to process the returned array and take into account the offset in received data caused by non-located points.

For this exercise, you should perform a test to check for non located points and retrieve their number. Print the indices of non-located nodes and print the located nodes with their corresponding value:
“x y value”
The meshes are randomised by the code during their creation, so the non-located nodes may be different for every subsequent run of the application.

Depending on the real-life application, all points might have to be located correctly. In order to eliminate the problem of non-located points, you can increase the geometric tolerance in the primitive *PCW_Create_coupling*. Try to evaluate the impact of a higher tolerance on the initialisation time.
If the surfaces are not perfectly superposed (for example if one domain is a sub-domain of the other), the received values in the array must be sorted so that they match with the nodes or elements of the associated mesh. Use the primitive *PCW_Get_not_located_points* to rebuild the array *relocalvalues* for graphical output in in the Fortran code.
There is also the primitive PCW_Reorder which does this job automatically by assigning a default value to every non-located point, but you might still need the flexibility of the primitive *PCW_Get_not_located_points* to process the points individually.

16.8 Exercise 3: time-varying coupling

Use the previous coupling application and increase the value of *itend* in the configuration files *dataX.dat*. Caution, itend must be identical in both files; otherwise one code will have finished the execution and the other code will wait indefinitely. Set the other parameters as in exercise 1 to ensure the localisation of the points. When executing the application directly, the following error message appears:

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

This is due to the fact that the mesh is created inside the time loop. At the second time step, CWIPI informs you about the problem that the mesh is already defined.
Make the necessary modifications so that the mesh is only defined once per execution.
When the application is operational, load the CWIPI output files. The number of solutions depends on the parameter output frequency given to the *PCW_Create_coupling* primitive (1 if intermediate output is desired for each coupling, 2 for every second coupling, etc.). In addition, the time step and time value must be incremented at each iteration, otherwise the coupler exits with an error message.

Note that the surface in the CWIPI output files is static. The surface written by the Fortran code is mobile and moving with the time. However, CWIPI cannot see this mobility since it has never been told that the surface has moved. You can check this behaviour if you select different flapping frequencies for the two codes. The data exchange still has only localised points.

16.9 Exercise 4: time-varying coupling with moving coupling surface

Use the previous coupling and find a method to notify the coupler about the moving surface.

16.10 Advanced topics with CWIPI

16.10.1 Definition of the interpolation points

You can redefine the definition of the the interpolation points which are the cell centers by default or the vertices depending on the solver type. For example this feature is useful in a finite element solver if you want to obtain values at Gaussian points for integration or if the solver needs to make available a mesh different from the points where is wants to retrieve the information (for instance the complete 3D mesh is available, but data shall be only retrieved on a surface).
This overloading is done with the function *PCW_Set_points_to_locate*.

16.10.2 Asynchronous communication

The asynchronous communication allow for optimisation of the exchange and to have better control of the synchronisation points between the codes. A data exchange with *PCW_Sendrecv* can be replaced with the following call sequence:

```
PCW_irecv(…)
PCW_issend(…)

…
Other instructions
…

PCW_Wait_irecv(…)
PCW_Wait_issend(…)
```

The synchronisation points are located in the PCW_Wait functions. In order to optimise application computation time, some source code may be placed between the PCW_issend/PCW_ireceive and PCW_Wait calls. This mechanism is based on asynchronous I/O in MPI, theoretically this allows to have some overlap between the calculation time and communication time, but the efficiency depends strongly on the application and on the quality of the MPI implementation. The other benefit of asynchronous I/O is to avoid the potential of *deadlock* due to simultaneous calls of two transmissions or two receptions in the two codes.

16.10.3 User defined interpolation

You can also overload a different interpolation method by defining a user function in C or Fortran in accordance with the following prototype:

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)
```

FORTRAN:

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

CWIPI passes all relevant data to the user function in the arguments. In particular the user has access to the result of the geometric localisation.

A user function can be registered in CWIPI via the following function calls:

- *PCW_Set_interpolation_function* for a user function written in C
- *PCW_Set_interpolation_function_f* for a user function written in Fortran

This feature is interesting for finite elements solvers of discontinuous Galerkin methods which use higher order elements. In this case, the user function can perform a more precise interpolation based on the basic element functions. Another application could be a higher order interpolation on Cartesian grids by Lagrange interpolation.

16.10.4 Python interface

As for Palm, the CWIPI library can be used in Python units via the Cython interface. Data exchange is based on buffers declared as *numpy* arrays. All primitives are grouped in the *Coupling* class.

The code extract below should give you a first glance on how to use the Python primitive calls. For more details on the Python interface, refer to chapter 18 dedicated to Python applications. Python uses an object called *Coupling* containing the attribute *coupling_id*, so all *PCW_* primitives are already aware of this parameter, it does not have to be specified at each function call.

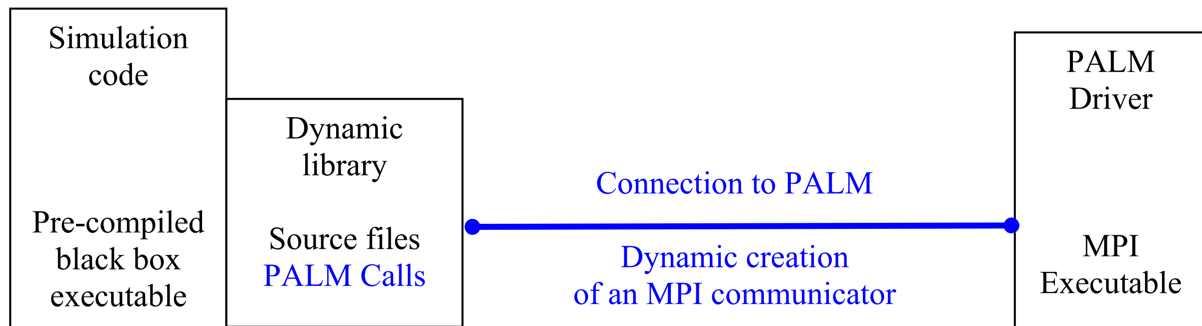
```
import mpi4py.MPI as MPI
import numpy as np
import palm
import PCW
import palm_user_param as pu


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
```

A complete CWIPI example written in Python and based on the stub in this chapter can be found in the directory *corrige_python*.

17 Connection of an external code to a PALM application

17.1 Introduction

Due to the difficulty of "PALMing" certain program whose source codes are not available, it has been developed, for the PALM coupler, a way to work through a dynamic library. Some commercial codes offer the possibility to include some user defined treatment that have to be compiled and archived in a dynamic library which is loaded at run-time. PALM exploits this principle.



We have to keep in mind that a dynamic library (`.so` file in unix/linux, `.dll` for Windows) is a collection of sub-routines connected to an application through its name (and possibly its path). The executable knows just the functions' prototype and the path to access them. When the main program runs, the library is dynamically loaded in memory at the first call (it could also not be loaded, if the program doesn't call any of its entries while it runs). This technique allows to propose some user defined functions, since they are not directly modifying the main program. Another advantage of the dynamic libraries is to create smaller executables, since the binary code of the library is not copied in the executable image.

We have seen that a PALM unit is a FORTRAN subroutine or a C function with no arguments. The main program is created by PrePALM (service file `main_*.c`). For the user this approach has the advantage that he does not have to explicitly call some functions such as `PALM_Init` or `PALM_Finalize`. These calls are in fact directly included in the main created by PrePALM. We've also seen that this technique allows PALM to create some blocks around various units, the blocks permitting to concatenate several units in just one executable, which would be impossible to do if the codes included their main program. This encapsulation imposes a strong constraint: we need to have access to the source codes of the main program. This may be too intrusive to adapt an existing program to PALM, especially for commercial codes.

If we need to couple a code of which we cannot edit the sources, we can launch it independently from PALM (avoiding therefore to spawn the unit with `MPI_Comm_Spawn`), and to establish afterwards a communication context via the creation of an MPI communicator, as illustrated in the previous figure.

17.2 How it works

The two primitives that drive the connection/disconnection with PALM are `PALM_Connect()` and `PALM_Disconnect()`. **N.B.**: in a parallel code, these primitives have to be called by all the processes.
Once `PALM_Connect` has been called, the user can access the full PALM environment, and therefore call all the others PALM primitives as, for instance, `PALM_Put`, `PALM_Get`, etc and write to the usual PALM output files.
Remark: `PALM_Connect` is based on the client/server capabilities of MPI-2, therefore only the MPI-2 mode of PALM can be used in this situation.

Constraints:

1. If the code itself already uses MPI, the dynamic libraries must be compiled with the **same MPI distribution and version**. This constraint is quite hard because the commercial distribution of computational codes are not necessarily linked against PALM compliant MPI releases. Moreover, if several codes are coupled to PALM via PALM_Connect, they all have to be compiled with the same MPI version.
2. The **MPI library must support dynamic libraries**. Pay attention because this is often an option that has to be explicitly turned on when installing MPI
3. PALM must be installed with the **`--with-shared_lib` option turned on** when issuing the `configure` command (*cf.* Chapter 20 on PALM installation).

The usage of this extension is illustrated on two examples that you can find in the `chapter_16` directory. Coupling a single processor or a parallel code does not change the procedure.

17.3 Connecting a single processor code to PALM.

In this case the computational code is a toy model that mimics the mechanism of the extension of a pre-compiled code with user defined functions compiled in an external library. You'll find the sources (in C) of the toy model in the `code.c` file of the `chapter_16/connect_code` directory.
Here are the contents (comments are in french, ask for translation if needed)

```
/* 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");
}
```

This code calls three user defined functions. The first one (`udf_init`) in the initialisation step, the second one (`udf_inloop`) at every iterate of the time loop, the last one (`udf_end`) before exiting the program. These user defined functions are compiled in a dynamic library from the sources in `udf_vierge.c`.
Initially, neither the code, nor the library make any reference to PALM.
The `make_code` script, compiles the application in stand alone mode. Notice that the code being single process it does not make any reference to MPI either.

Compile and run the code with these two commands

```
>make -f make_code
>./code
```

To connect this code to PALM, you have to modify the user defined functions and create the appropriate dynamic library. As for any other PALM unit, the user has to write the unit identity card. It is almost identical to the usual ones, but for the additional `-functions` field where we have to indicate the command to start the computational code. We use here the `mpirun` command, because, even if the initial code did not use MPI, with the new dynamic library it will become part of an MPI context.

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

The syntax of the launching command (here `mpirun -np 1 ./code`) depends, of course, the installed MPI library. Here we gave the LAM/MPI command as an example. The `-functions` field could rather contain the path of a shell script setting up the appropriate environment or even be empty if the user takes care of launching the executable before starting the PALM application.

The remaining entries in the identity card are absolutely standard and define spaces, objects, etc.

```
/*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}
*/
```

In the sources of the user defined functions, listed here, you'll notice in blue the lines that have been added to include the references to MPI and the PALM calls.

```
#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;
```

```
}
```

The `PALM_Put/Get`, `PALM_Write` and other PALM calls are exactly the same as in classical units. Keep in mind that there are no limitations to the PALM functions with this mode of connection.
In the PrePALM canvas, the only difference is that you have to indicate how the unit is executed. In the unit properties window (it opens when you insert it in the branch code, or afterwards if double clicking on the rectangle). In this case, choose `EXTERN-TO-CONNECT` in the `Execution mode` field. You'll see that a grey rectangle will surround the unit symbol, as in the following example:

In our toy case, the fields produced by the model are multiplied by a constant at every time step. To do that, we use the BLAS `SSCAL` unit from the PALM algebra toolbox.

Remark: this connection technique is based on the MPI-2 client/server capabilities. On some platforms (e.g. NEC SX series) the communications go through the TCP/IP protocol instead of exploiting the full bandwith of the interprocessor bus. We could not expect the same performances of the communications as with the standard units. Moreover the “external” units cannot be part of a block. If the user can access the sources of the code he has to couple, it is therefore strongly recommended to transform it in a standard PALM unit.

17.4 Connecting a parallel code to PALM.

In the `chapter_16/connect_code_par` directory, you'll find a full example of how to connect an external parallel code.
The main differences with the previous case are:

- the presence of a distributor (`dist_d3x100.f90`) for the exchanged array because the object is distributed on all the processes
- no calls to `MPI_Init` and to `MPI_Finalize` in the user defined library because these calls are already issued by the parallel main code.

```
/*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 To go further: IP connection of an external code

As seen previously, a strong limitation of the connection type presented before is that if the code to connect already uses MPI, the dynamic libraries of this code and PALM must be compiled with the same MPI distribution and version. This constraint is quite hard because the commercial distribution of computational codes are not necessarily linked against PALM compliant MPI releases. Moreover, if several codes are coupled to PALM via PALM_Connect, they all have to be compiled with the same MPI version. To overcome this too strong constraint, the external code connection has been extended with IP socket protocol.
The concept of sockets has been introduced in the Berkeley Unix distribution (an historical UNIX system, parts of which are sill used today), explaining why we sometime talk about *BSD* (*Berkeley Software Distribution*) *sockets*. This is a model allowing inter process communications. The processes can either communicate within the same machine or through a TCP/IP network. There are two types of communications:

- The connection oriented mode that use the TCP protocol. In this communication mode, a stable connection is established between the processes so that the destination address has not to indicated at each data exchange.
- The connectionless mode that uses the UDP protocol. This mode requires the destination address at each communication.

The TCP connected mode is used in PALM. As for the opening of a file, a socket communication uses a descriptor to identify the connection on which the data are exchanged. As a result, the first operation to do consists in calling a function that creates a socket and that returns a descriptor to uniquely identify the connection. A socket is the combination of an IP address and a port number (connection address on a machine). This combination then becomes a unique address in the world allowing an univocal connection.
Once the socket is created, the server listens to possible messages. As explained, in the following, this server is a mirror unit of the external code and it is integrated in the coupling scheme instead of the code. The external code is a client of the mirror unit: it sends requests to the server. In order to distinguish that a code is a client of a PALM application, the PALM primitives PALM_XXX are interfaced as PALMIP_XXX. When the client calls a primitive PALMIP_XXX, it sends to the server a request asking it to remotly execute the PALM primitive PALM_XXX. The server executes the PALM primitives and returns the error code to the client. Moreover, the IN arguments of PALM primitives are sent to the server and the OUT arguments are received from the server. Note that the function PALM_Write(PL_OUT,...) works in C but the Fortran primitive Write(PL_OUT,...) cannot work because the file unit PL_OUT is opened by the server which is coded in C.
In a local PALM communication, the user has the responsibility to manage the buffer size used for the Get/Put/Dump primitives to avoid accessing non-allocated memory. For the Palm via IP interface, the size of messages passed through the socket connection must also be managed. This is why the primitives PALMIP_Get/Put/Dump**_sized** takes an additional argument *size* which defines the size (in bytes) of the message transferred through the socket. This number must be the same as the size of the local buffer to avoid segmentation faults. If the application does not use distributed objects (for parallel communication) or sub-objects, the standard primitives PALMIP_Get/Put/Dump can be used which determine the static size of the space via the function call PALM_Space_get_size. But for distributed objects or subobjects, the size of Get/Put/Dump is different from the size of the complete space and the simple primitives PALMIP_Get/Put/Dump must be avoided.
The following figure presents the main working principles of a PALM application with IP protocol for the external connection of a code. Notice that the external code is integrated in the PrePALM canvas as a mirror unit. This unit plays the role of a server in the PALM application, replying the requests of the client; the external code.

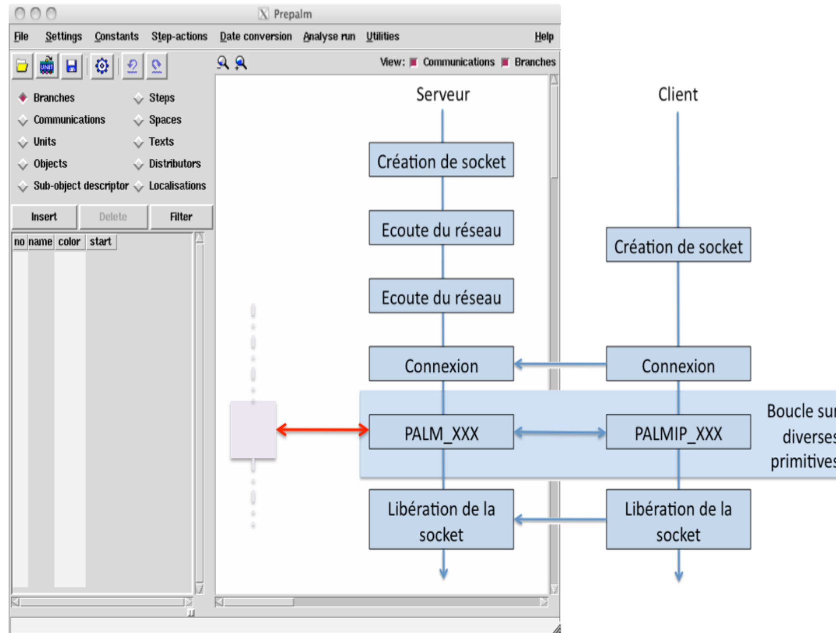


Illustration 1: Working principle of a PALM application with an IP connection to an external code.

To illustrate the use of the IP protocol in PALM, the example of secion 17.4 "Connecting a parallel code to PALM" is adapted and presented here. In the directory *chapter_16/connect_code_par_IP,* you will find the complete example. In this tutorial, we aim at connecting a code to a PALM application that run on the same machine but with two different implementation of MPI:

- MPICH2 for the PALM application,
- LAMMPI for the code to connect.

To do so, the PALM library must be installed on the machine with the two distributions of MPI. **Then, attention must be paid during the execution of this tutorial to open two distinct terminals, each one pointing (via the PATH environment variable) to a particular distribution of MPI for the compilation and for the execution.** The sources of the code to connect are located in the directory *chapter_16/connect_code_par_IP/code*. It is strictly the same code as in section 126 "Connecting a parallel code to PALM". The compilation is simply achieved by typping *make* in the terminal pointing to LAMMPI. The *makefile* file is thus

```
# ==========================================================

include Make.include
OBJS = code.o
all : code
code : $(OBJS) lib_dyn_udfv
       $(CC) $(CCFLAGS) -o $@ code.c ./udf_so.so $(LIBS)
lib_dyn_udfv :
        $(CC) $(CCFLAGS) $(SOFLAGS) -o udf_so.so udf_vierge.c
lib_dyn_udf :
        $(CC) $(INCLUDES) $(CCFLAGS) $(SOFLAGS) -o udf_so.so udf.c
clean :
        \rm -f udf_so.so code.o code
.SUFFIXES : .f90 .F90 .F .f .c .cc .C .c++ .cpp .o .so
.c.o:
        $(CC) $(INCLUDES) $(INCPALM) $(CCFLAGS) -c $< -o $@
.c.so:
```

```
        $(CC) $(INCLUDES) $(INCPALM) $(CCFLAGS) $(SOFLAGS) -c $< -o $@
```

and is linked to the following *Make.include* file:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
PALMHOME = $(PALM_MP)/linux64r4lam
CC = mpicc
CCFLAGS = -tp k8-64
SOFLAGS = -shared -fpic
INCLUDES = -I$(PALMHOME)/include
LIBS = $(PALMHOME)/lib/libpalmip_client.so
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
```

In the *Make.include*, we see that the path *PALMHOME* points to the LAMMPI installation of PALM. Moreover, the *LIBS* variable contains the dynamic library of for the IP client of PALM. The *code* executable as well as the dynamic library *udf_so.so* needed by code and containing empty UDF routines are created. To execute this program on 2 processes, the standard procedure for LAMMPI is used:
> *lamboot*
> *mpirun -np 2 ./code*
> *lamhalt*

User functions (*udf.c*) are modified in a way to call IP primitives to generate the connection with the PALM world and to perform exchanges. The Id card of the corresponding unit is recalled below. Compared to the one of section 126 "Connecting a parallel code to PALM", only the fields *functions* and *object_files* of the attribute PALM_UNIT are modified. As already mentioned, the server part of the IP communication is done by a mirror unit, which is a bridge between the PALM world and the network. Hence, the description of the unit indicates in a standard way the file acces that correspond to *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}
*/
```

The source code of the mirror is always the same for all applications and codes to connect. Blue parts below underline the fields that can be modified by the user. Notice the definition of the ports on the machine on which the server will listen to requests. Notice also that each process of a parallel code will communicate on a specific port, defined here by it's MPI rank.

```
#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);
}
```

The user functions are presented below. Note that to establish a univocal communication, it is important to specify the IP address of the machine where the server runs as well as the port number defined in ad equation with the mirror. In this example, we will se that the PALM application generates a file *code.palm_connect* which contains the IP address on which the server runs. This choice is of particular interest for applications running on parallel architectures where the IP address to take into account for the communications is the one of the node where the server is

running. Finally, the source of the UDF illustrate that the PALM primitive for the client are unchanged: there is only a specification *IP* in the name of the primitive, which allow to identify that we are dealing with a client code of a PALM application.

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

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

/* variables destinÃ©es 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 destinÃ©es Ã  retourner le nombre d'itÃ©rations */
/* appelÃ©e une seule fois par le code, Ã  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 Ã  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 connectÃ© */
  *max_time = 15;
  /* remarquer qu'aprÃ¨s 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;
}
```

To compile this modified user functions and to create the dynamic library *udf_so.so*, simply type *make lib_dyn_udf* in the terminal linked to LAMMPI.

The PALM application *connect_code_par_IP/application_openpalm/boucle.ppl* is simply edited with PrePALM. Compared to the one of section 126 “Connecting a parallel code to PALM”, the independent code is directly represented by the mirror unit in the caneva. The call to the primitive *MPI_GET_PROCESSOR_NAME* in a fortran region on the branch where the mirror is running allows to identify the IP address and to create the file *code.palm_connect*. Note that the client code is launched with the script *lance_code.sh*.

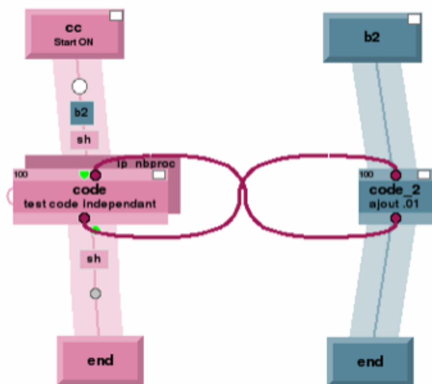


Illustration 2: PrePALM caneva of the IP connection application

This script *lance_code.sh* allows to create a symbolic link to the dynamic library *udf_so.so* in the execution directory as well as to execute the code with the launcher of LAMMPI:

```
#!/bin/sh
#

if test -f udf_so.so
then
  echo "la librairie dynamique est presente"
else
  ln -sf ../code/udf_so.so .
fi
/usr/local/lam7.1.4/bin/mpirun -np $1 ../code/code &
```

The compilation of the PALM application is done in a standard way in the terminal pointing to the MPICH2 distribution of MPI. The *Make.include* file looks:

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
PALMHOME = $(PALM_MP)/linux64r4mpich
F90 = mpif90
F90FLAGS =
LF90FLAGS =
F90EXTLIB =

F77 = mpif77
F77FLAGS =
LF77FLAGS =
F77EXLIB =

FPPFLAGS =

CC = mpicc
CCFLAGS =
```

```
LCCFLAGS =
CCEXTLIB =

C++ = mpiCC
C++FLAGS =
LC++FLAGS =
C++EXTLIB = -pgf90libs

OMPFLAGS =

INCLUDES = -I/usr/local/include -I../../SRC/HEADERS

LIBS= -lblas -lX11 $(PALMHOME)/libpalmip_server.a
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ #
```

We note in the *Make.include* file that the path PALMHOME point to the PALM version installed with MPICH2 and that the *LIBS* variable contains the dynamic library of the PALM IP server.
To execute this coupling scheme, it is necessary to initiate the LAM environment on the machine by typing *lamboot* in the terminal pointing to LAMMPI. Then, the PALM driver is executed in the terminal pointing to MPICH2:

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

To conclude, the IP connection also works with the MPI1 mode of PALM. To use this mode, it is necessary to replace MPI_COMM_WORLD by PL_COMM_EXEC in the mirror source code (with an inclusion of the PALM library).

17.6 Summary of the main concepts

In this session you have seen how to connect an *external code* to PALM. By external we mean that the code is not encapsulated in an executable (unit or block) started by PALM during the coupling algorithm. The code can therefore be a black box commercial software, under the condition that it provides a way to call user defined functions. This PALM extension relies on the quite common use of dynamic libraries of user defined functions. In the examples you have seen the slight differences for connecting a single processor code rather than a parallel one. Pay particular attention to the remarks pointing out the software environment constraints that this approach implies.

Moreover, if the code is not compatible with the MPI version used to compile PALM or if the MPI-1 mode of PALM is mandatory, the solution consists in using an IP connection with a mirror unit. This solution can also be used for code coupling on heterogeneous architectures.

18 Writing PALM units in Python

Initially the Python interface was based on the SWIG library as described in chapter 19, but since OpenPALM version 4.1.4 it has been rewritten in order to be more compatible with parallel codes whose main program is an assembly of Python modules. Actually the SWIG based solution only allows server/client based MPI coupling and forces the programmer to use the MPI2 version of OpenPALM without access to the CWIPI library.

The current solution works equally well in MPI1 and MPI2 modes of OpenPALM, and is based on the NUMPY [1] tools for array handling, MPI4PY [2] for the MPI interface (these are standard tools for parallel codes using python), and CYTHON [3] to build the interface with PALM. Therefore you must make sure that these tools are installed on your machine, or install them from their open source repositories.
[1] http://numpy.scipy.org
[2] http://mpi4py.scipy.org
[3] http://www.cython.org

The interface files (interface_palm.pyx for PALM and interface_pcw.pyx for CWIPI) written in CYTHON are provided in the folder PrePALM/TEMPLATE. They are automatically copied into the project folder when the PALM service files are created. The PALM Makefile instructs the compiler to build the library palm.so for use in the python code.

You must add the path towards python, cython and the other libraries into the Make.include file:

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

One can easily obtain these paths from a command shell via:

```
$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 Python unit

If you create a Python unit, you have to fill in the ID card so that PrePALM can recognise the Python unit:

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

After loading the ID card, the unit can be inserted into a PrePALM branch like any C or FORTRAN unit.

The OpenPALM objects have to be declared in the same way as for C and FORTRAN units.

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

PrePALM will automatically generate the Makefile able to compile the Cython interface into a dynamic library which will be loaded by the Python script. After executing *make*, you will find the file palm.so in the project folder.

Then you have to include this dynamic library (created via Cython) into the Python application. You must also include numpy for handling the data arrays.

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

To access the PrePALM constants, you have to import the module palm_user_param.py. There are two choices:

- using a separate namespace :
  ```
  import palm_user_param as pu
  ```
  you can now access the variables via the namespace *pu*:
  ```
  print pu.const1
  local_var = pu.const2 + pu.const3
  ```
- in the main namespace :
  ```
  from palm_user_param import *
  ```
  this makes the variables directly available in the code :
  ```
  print const1
  local_var = const2 + const3
  ```

Usually you will choose the second solution, but you have to pay attention when choosing the variable names to avoid name conflicts.

The application body is included in a procedure having the same name as in the -functions attribute of the ID card.

```
def test_send():
```

18.2 Object oriented Python interface

In Python, there is little flexibility for argument-passing in the function calls because variables are not typed. All objects are passed by reference, however elementary variable types (int, float) are passed by value.
The OpenPALM interface for Python is built on classes to simplify the passing of output variables as class attributes.

You must first create a Palm object which will be used for the exchanges. You can directly give the constructor all parameters needed for the initialisation of its attributes. The missing attributes are initialised to default values.

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

The attributes time and tag have been omitted on purpose, since their default values PL_NO_TIME and PL_NO_TAG are perfectly suitable for this example.
The attributes can be modified at any moment

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

In this example, a dynamic space is used, so the shape must first be communicated to the Palm driver. All attributes (rank, shape) have already been initialised at the creation of the Palm Object, so you can directly call the function:

```
po.space_set_shape()
```

Then you have to create a numpy array of type *double* containing an integer range from 0 to *dim1*dim2 - 1* and you can send it directly since the *po* object contains all information needed for *put*:

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

18.3 Dynamic communication via OpenPALM

Let's put this transmission unit test_send.py in a PrePALM branch, followed by a unit test_receive.py which receives the data.

In this example we need 2 constants in PrePALM:
dim1: integer, value = 7
dim2: integer, value = 5

The module test_receive.py has to retrieve the dimensions of the dynamic space and get the data. It uses also a PalmObject, but this time via a 'NULL' space.

```
#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}
```

In the class PalmObject you can directly call the sequence of name inquiry, rank and shape inquiry since the methods use the attribute *dynspace* reserved for dynamic spaces, while the attribute *space* remains set to 'NULL'.

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

You must create an empty numpy array with the correct dimensions according to the shape attribute to receive the exchanged data, and you are set:

```
matr = np.empty(po.shape, dtype = np.double)
po.get(matr)
```

18.4 Parallel codes: Get MPI communicator

In a parallel Python unit with MPI, the Python interface uses MPI4PY. In this case the different processes have to share the same MPI communicator. OpenPALM provides this communicator via the function get_mycomm.

The MPI communicator object can be used after importing mpi4py:

```
import mpi4py.MPI as MPI

Mycomm=MPI.Comm()
palm.get_mycomm(Mycomm)
```

You can now use MPI commands on this communicator:

```
rank = Mycomm.Get_rank()
size = Mycomm.Get_size()
```

18.5 Python help function

Once you have compiled the Cython Palm module, it will provide an online help function inside an interactive Python console:

```
>import palm
>help(palm)
```

This command lists all OpenPALM functions and their use in Python. The same help text can be requested at the command line via:

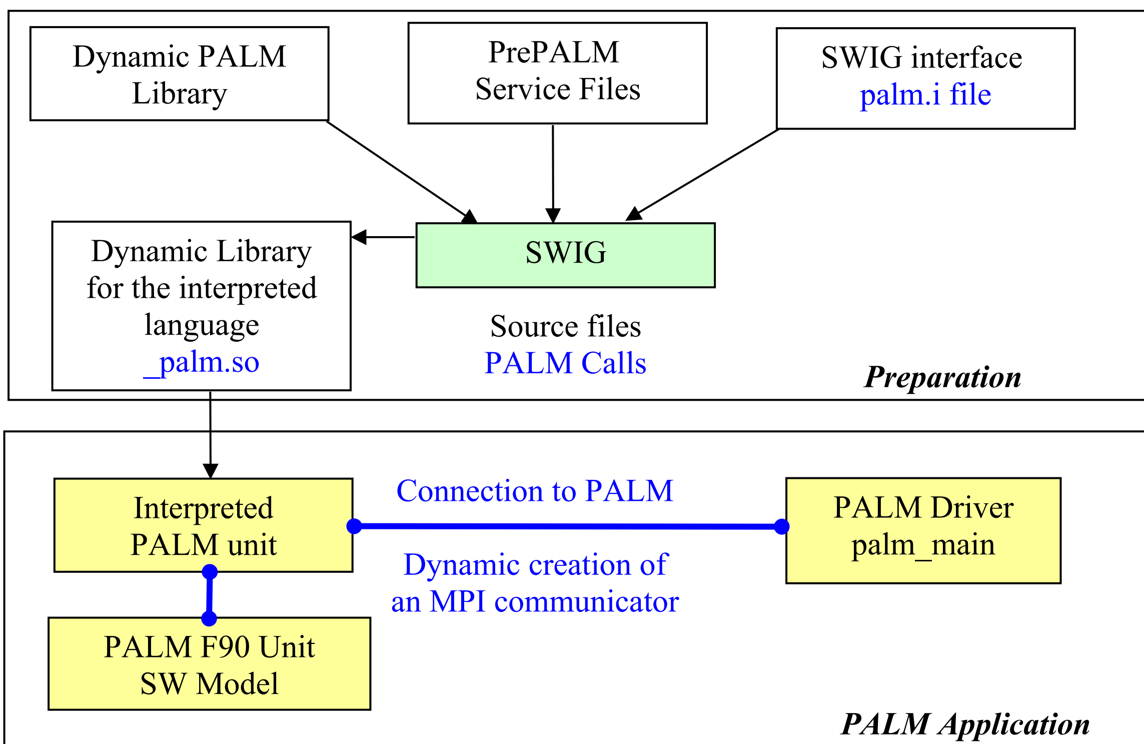
```
pydoc palm
```

19 Writing PALM units in interpreted languages such as Perl or Tcl/Tk

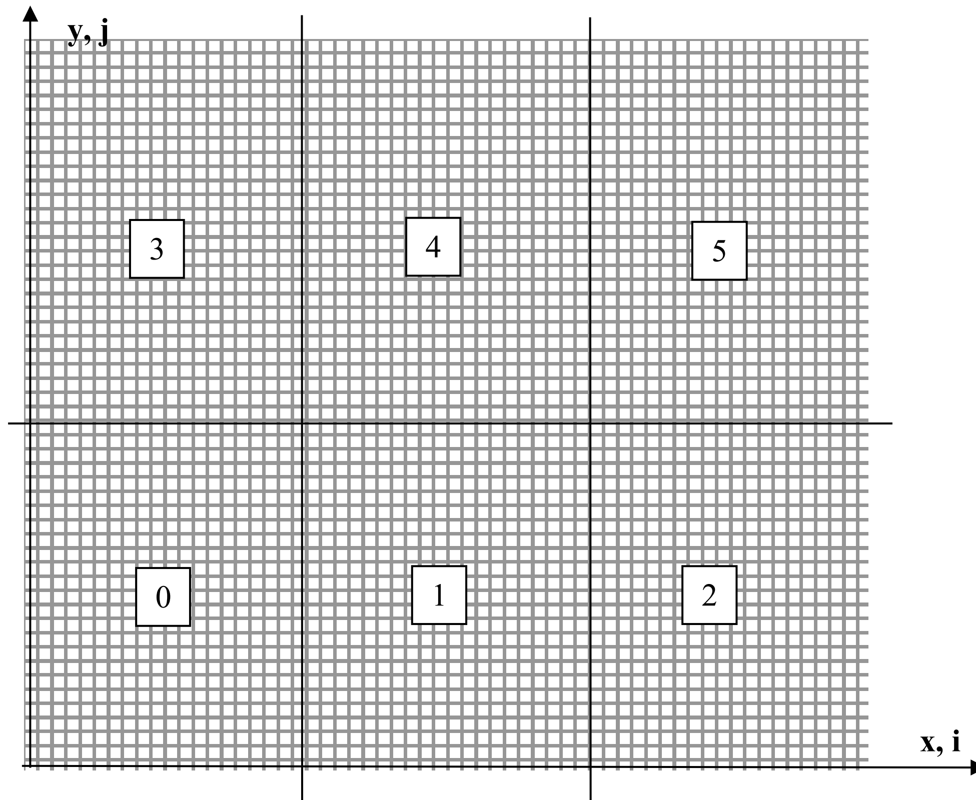
19.1 Introduction

Most of the interpreted languages used in scientific computing, such as Perl or Tcl/Tk, authorise to interface some pre-compiled functions written in other languages. Here again, they are loaded and executed as dynamic libraries (`.so` files in Unix and Linux, `.dll` files for Windows).
We can take advantage of this opportunity to write PALM units in interpreted languages. The aim is to effectively exchange data through the PALM API (mainly `PALM_Put/Get`) thus avoiding the use of intermediate files. One interesting application is the possibility to pre or post-process data in parallel with the application consuming or producing them. Another one is the interactive steering of a computational code through a graphical user interface.
This mechanism is based on the connection (via the MPI-2 client/server capabilities) of external codes (the interpreted procedure, in this case) that you have seen in chapter 17 and on the generic interface tool **SWIG** (that can be freely downloaded from www.swig.org). The user does not need a thorough knowledge of SWIG because the PALM interface declaration file (`palm.i`) is already provided with the PALM distribution. On the contrary, it is mandatory to install SWIG on the machines meant to compile and run the PALM application.

As an example of application, starting from the same parallel code driven by PALM (a Shallow Water model, often referred to as SW), we are going to build three different couplings.
In the first one, SW will interact with a Python unit, in the second one with the same unit written in Perl, and in the third one with a visualisation and steering tool written in Tcl/Tk. The principle is the same in all cases and the SWIG interface declaration file (**`palm.i`**) is the same for all languages.
The sources of the applications and the `palm.i` files are available in the `chapter_17/unit_python`, `chapter_17/unit_perl` and `chapter_17/unit_tcl` directories.



The SW (Shallow Water) model represents a rectangular domain. It is based on the Saint-Venant equations, integrating with an explicit time stepping scheme the water height H and the U and V components of the 2D velocity field. The discrete quantities are represented on a (i,j) regular structured gird. The size of the domain, the grid step (i.e. the number of grids cells per direction), the simulation length and some other model parameters are specified as PrePALM constants. The model is parallelised by domain decomposition in the two directions. The number of domains in the x and y directions are specified as PrePALM constants as well. The distribution functions are coded as functions of these PrePALM constants and are therefore valid for every combination of grid size and domain decomposition.



Example of domain decomposition on 6 processors
(`ip_nbproc_x` = 3 and `ip_nbproc_y` = 2 in the PrePALM constants)

The identity card of the SW unit is the following:

```
!================== 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}

!================================================================================
```

This unit receives in input at every time step (in the innermost loop) the `putflag` flag, indicating if a PALM_Put of the `hn` field, containing the water height, has to be issued.
As output, SW sends at the very beginning of the execution the number of time steps it is going to perform. Afterwards, at every time steps it sends the value of the current time counter in the object `time`.

For PALM, the SW unit works this way:

- at the beginning of the simulation, the unit provides the total number of iterations
- at every time stepping iterate, the unit
  - sends the current iterate number
  - gets a flag indicating the action to perform
  - if the flag is equal to 1, sends the current value of the `hn` field

19.2 PALM unit in perl

The perl unit that we propose as an example simply recovers some time instances of the water height field H from the SW model and prints them out in the PALM log files. You'll find the application sources in the `chapter_17/unit_perl` directory.

After having received the total number of iteration from the SW unit, the perl unit enters its internal loop with the same number of iterations. In this loop it receives the current iterate count from the

SW model. If this corresponds to an output time, the perl unit sends a flag to SW to ask for a PALM_Put of H, and issues a PALM_Get to recover the field and then prints it out. To be more generic, the space for the input field is set to NULL in the perl unit. We use therefore the specific PALM primitives (as seen in session 9) to recover the information on the size and shape of the space.

unit_perl.pl:

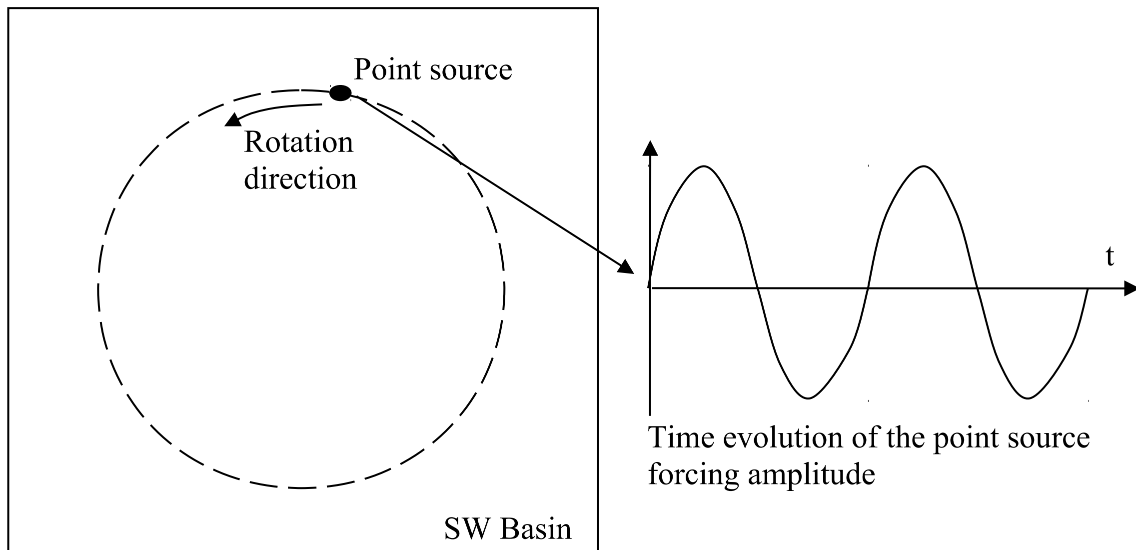
```
#PALM_UNIT -name unit_perl\
#          -functions {SH run_unit_perl.sh&}\
#          -comment {unit_perl}
#PALM_OBJECT -name putflag\
#            -space one_integer\
#            -intent OUT\
#            -comment test
#PALM_OBJECT -name time\
#            -space one_integer\
#            -intent 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 PALM unit in Tcl/Tk

This example is still close to the previous ones. The model is stricly the same, but, to introduce some new features, we'll rely on the the graphical capabilities of Tcl/Tk, often used in the GUI implementations. In our case it will allow to code a very simple visualisation tool to represent the 2D H field. We'll go a little further, giving the user the possibility to interactively modify a parameter of the SW model.

The square basin is forced by a point source moving, at constant speed, along a circle centred in the basin. This source modify locally the water height with an amplitude varying in time as a sinus. To illustrate the steering capabilities, the user can reverse the source rotation direction by a simple click in the Tcl/Tk graphical interface and he'll visualize the effect on the results in the visualisation window.



To build the PALM application you follow the same procedure as in the previous examples.

Here are the sources of `unit_tcl.tcl`, a simple visualisation and steering interface written in Tcl/Tk that calls the PALM primitives directly from the tcl code

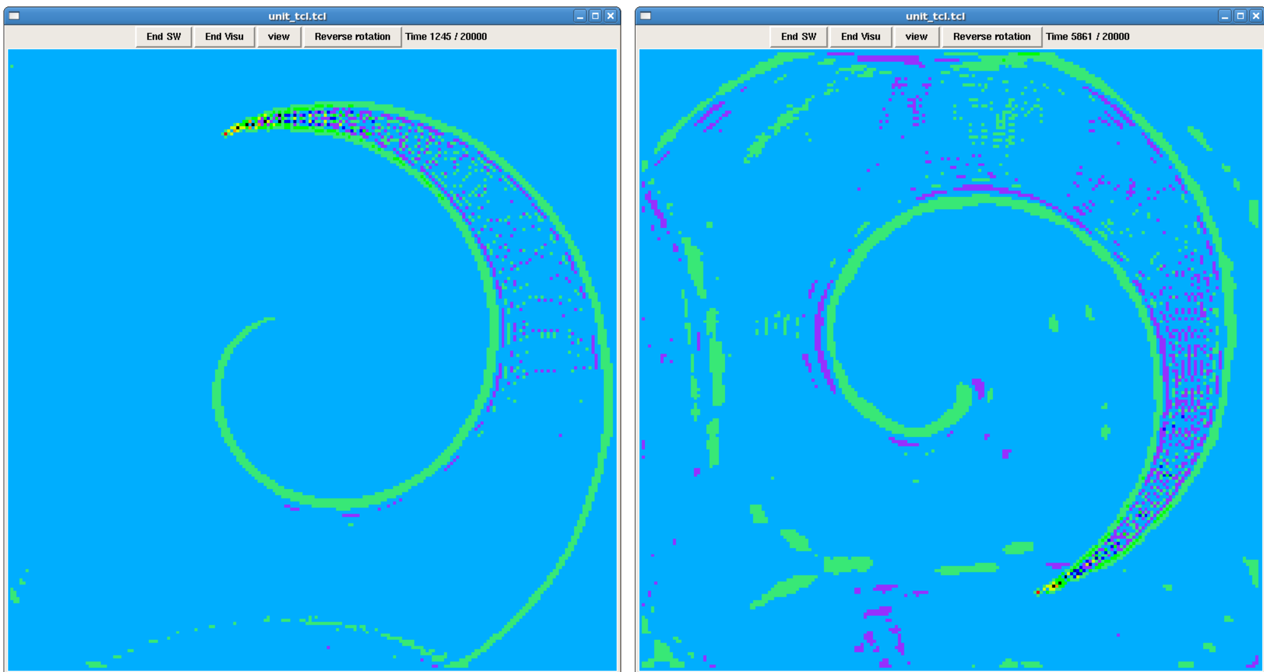
```
#!/bin/sh
# the next line restarts using wish\
exec wish "$0" "$@"

#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
```

Once the application has been compiled, you can start the PALM application that, in turn start the the graphical user interface:



unit_tcl.tcl : Interactive visualisation of the water height before and after reverting the source rotation direction.

This interface is just an example of how easily you can interface a script or a graphical software and transform them in PALM units.
Notice that we exploit in these examples a number of PALM features:

- the parallel remapping: even if the model is parallel the full H field is gathered before being received by the visualisation tool
- the space inheritance: the graphical interface is able to adapt itself to any grid size
- the dynamic connection of external units
- the MPI based data exchange, granting high performances.

19.4 Summary of the main concepts

This session was dedicated to a particular case of the situation presented in the previous chapter. In this case, the external unit to be connected is not pre-compiled but rather written in an interpreted language. The interface with the dynamic PALM library is created with the SWIG tool.
Some paradigmatic example of on-line post-processing and interactive steering are provided, also illustrating how to couple a F90 model with a perl or Tcl/Tk interpreted procedure.

20 PALM Installation

20.1 Introduction

In the PALM distribution you'll find the source codes of the PALM library, of its interface and of all the sessions of the training. OpenPALM is free software LGPL v3.

The first thing to do you have to decompress the gzipped tar archive of the distribution:

```
> tar -xvfz distrib.tgz
```

Two directories are created: `PrePALM_MP` and `PALM_MP`. The first one contains the graphical user interface PrePALM, the second one the PALM library. The interface can be locally installed on the user workstation or PC, while the library has to be compiled on the different platforms where the PALM coupled applications are meant to run.

20.2 Installation of the PrePALM graphical user interface

20.2.1 Pre-requirements

The graphical interface PrePALM is written in Tcl/Tk with some C. Therefore you need these two environments on the machines where PrePALM has to run. The Tcl/Tk version has to be at least 8.3 PrePALM can run under Windows if a Linux emulator is installed. We recommend Cygwin.

A small C program is used to interpret the STEPLANG language: it is therefore necessary to compile this component. A pre-compiled version working on i386 to i686 and x86_64 platforms is provided with the PALM distribution.
The most widespread public domain algebra libraries (such as BLAS, LAPACK, ScaLAPACK) interfaced in the PALM algebra toolbox are not provided with the PALM distribution and should be installed (if they are not already pre-installed) on the machines where the final application has to be compiled and executed. On the contrary, the geophysical interpolation library based on the OASIS coupler and on the SCRIP algorithms is provided with the PALM distribution.

20.2.2 PrePALM command definition

The graphical user interface is written in Tcl/Tk which is an interpreted language. Therefore there is no need of compilation. Nevertheless every user has to set an environment variable containing the installation path and an alias as a shortcut for the GUI.
Accordingly to the preferred shell you should add to the `.cshrc` or `.bashrc` or `….rc` file:

csh, tcsh :

```
setenv PREPALMMPDIR path_to_PrePALM
alias prepalm '$PREPALMMPDIR/prepalm_MP.tcl \!* &'
```

sh, bash :

```
function prepalm {
export PREPALMMPDIR=path_to_PrePALM
$PREPALMMPDIR/prepalm_MP.tcl $* &
```

```
}
```

Optionally you can set the `PREPALMEDITOR` environment variable pointing to your preferred editor that PrePALM will start every time it proposes to access an external editor. The default is `vi`. If you rather prefer `emacs` you should add to the shell configuration file:

csh, tcsh :

```
setenv PREPALMEDITOR emacs
```

sh, bash :

```
export PREPALMEDITOR=emacs
```

20.2.3 STEPLANG interpreter installation

Steplang is the command language used to describe the event driven actions manipulating the objects stored in the *buffer*. If you need to recompile its interpreter, enter `STEPLANG` the directory

```
> cd PrePALM_MP/STEPLANG/
```

Modify, if needed, the simple `Makefile` and issue:

```
> make clean
> make
```

If everything go right, you should end up with the `steplang-i386` executable..

20.2.4 Installation of the OASIS library, if needed

The OASIS library takes care of the grid to grid interpolation of geophysical fields on a spherical system of coordinates. It works for most the structured or non structured grids used in the climate modelling community. You'll find more details in the documentation of the OASIS coupler, developed at CERFACS.

The source code of this library is included in the PALM distribution in the `PrePALM_MP/ALGEBRA/Interpolation/Geophysic/DSCRIP_lib` directory. Edit the `Makefile`, if needed, and simply issue

```
>make.
```

20.3 Installation of the PALM library

20.3.1 Pre-requirements

The PALM library includes the objects used to generate the PALM driver (`palm_main`) and the user defined entities (units and blocks). This library has to compiled on the platform where the PALM application will eventually run. The installation procedure is based on the automatic configuration tool `autoconf`. Remember that PALM has been implemented in FORTRAN 90 and C.

To install PALM it is therefore necessary to have access to:

- A FORTRAN 90 and a C compiler. They have to be compatible. The best idea is to use the two compilers from a same distribution and at the same version
- An MPI library that implements the MPI-2 standard (one does not need MPI-2 if he is only going to work in MPI-1 mode. *Cf.* Chapter 15). The MPI library has to compiled with the same compiler as in the previous item.

Optionally, depending on the PALM features you are going to use, you may need

- the standard scientific libraries BLAS and LAPACK (possibly optimised by the manufacturer).
- the parallel algebra libraries PBLAS and SCALAPACK
- the NetCDF I/O library
- the sources of the minimisers of which the interface is available in the PALM algebra toolbox

Remark: you do not need superuser rights to install PALM on your machine.

20.3.2 Installation

You install PALM with simply three commands from the `PALM_MP` directory of the distribution:

```
> ./configure [OPTION]... [VAR=VALUE]...
> make
> make install
```

The only step requiring some attention is the first one, for you have to choose the proper options for the configuration. They depend on your compilers, on the platform, on the MPI distribution and, finally, on the flavour of PALM (single proc, MPI-1, MPI-2) that you are going to install.

You can obtain a summary of the available options with the command ./configure –help that will answer:

```
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.
```

For normal usage, you have to concentrate on the bold blue options only. The remaining options are dedicated to the PALM developers.
In any case we suggest to explicitly choose the FORTRAN 90 and C compiler.
We have tested PALM with most of the available compiler suites. Amongst them, notice:

- gcc and gfortran, form the GNU suite
- gcc and g95
- pgcc and pgf90 from the PGI suite PGI
- intel compilers suite
- pathscale compilers suite
- xlc and xlf90 on IBM
- sxmpif90 and sxmpicc on NEC vector supercomputers.

The most thoroughly tested configurations (the ones used at CERFACS) are (pgcc, pgf90) and (gcc, gfortran).

Remark: it is absolutely mandatory that the C and FORTRAN compilers are compatible and to use them for compiling (in this given order):

- the MPI library
- the PALM library
- the object libraries for the PALM units
- the PALM applications.

Once you have chosen the compilers, you could maybe have to choose an MPI distribution and indicate it as an option of `configure`.
For the MPI-2 PALM mode, the following public domain distributions have been tested and validated:

- LAM/MPI version 6 and following: option `--with-lam=`*path where LAM/MPI is installed*
- OPENMPI version 1.2.7 and following: option `--with-openmpi=`*path where OPENMPI is installed*
- MPICH2 version 1.0.7 and following: option `--with-mpich=`*path where MPICH is installed*

For the MPI-1 PALM mode, almost every MPI distribution implements the MPI-1 standard with an appropriate quality and completeness.

20.3.3 Example of installation on a Linux workstation

Let's suppose that we have to install PALM on a 64 bits Linux workstation where the PGI compiler suite is installed (`pgf90` and `pgcc` commands), and where LAM/MPI 7.1.4 has been compiled with

these compilers and installed in `/usr/local/lam7.1.4/`. Moreover let's suppose that we want to install the PALM dynamic libraries but that we do not need the automatic promotion of 4 bytes REAL variables to 8 bytes double precision. The configuration command is:

```
./configure --enable-64bits –with-lam=/usr/local/lam7.1.4/ --with-shared_lib \
--with-F90=pgf90 --with-CC=pgcc
```

20.4 Summary of the main concepts

This chapter is a short summary of how to install PALM and its graphical interface. Every platform has its own specificity and sharing the experience will help avoiding the most frequent and common difficulties. Feel free to send your feedback on the installation to the PALM team.

21 Some more or less specific utilities

21.1 Default value and choice from a list of pre-defined values for the units input plugs

We have seen in the previous chapters that quite often the input objects described in the identity cards (and therefore corresponding to a `PALM_Get` in the unit code and to an upper plug on the canvas representation) are scalar flags or parameters that toggle a function of the unit. This inputs are filled with regular communications or hardwired (right click).
If it is suitable, it is possible to indicate in the identity card a default value for these objects. If this is the case, when the unit is inserted in the branch code, the corresponding plug will already be closed and set (hardwired) to the default value. You can, of course, modify it afterwards.
Let's take as an example the *producteur* unit from session 5 and let's add an integer input for specifying a working mode. If we want it to take the default value 1 we simply add in the id card:

```
!PALM_OBJECT -name run_mode\
!            -space one_integer\
!            -intent IN\
!            -default 1\
!            -comment {mode de fonctionnement de l'unite}
!
```

and we will add the corresponding `PALM_Get` in the code.

We can go a little further and define a “closed” list of value from which we can choose the one to hardwire. Let's imagine that in our example the unit can have 3 (and only 3) different working modes. With the `-closedlist` attribute we can limit the user choice to these values and, therefore, to reduce the risk of errors.

```
!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}
```

Every list item has to be enclosed in braces. The first string is the value of the variable, the following is taken as a comment that will be displayed when hardwiring the input.

21.2 Some subtleties on the time stamp: conversion to/from dates

21.2.1 Introduction

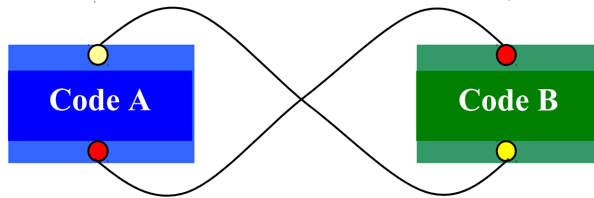
The “*time stamp*” argument of the `PALM_Put/Get` primitives is coded as an integer. This allows to have a simple and generic interface, but deserves some further explanations. In the session about communications we have insisted on the importance of this stamp because it allows:

- to handle easily the inner and outer loops on time of the codes to be coupled
- to interpolate the objects in time if the time steps of the units do not coincide

- to define different coupling frequencies in different applications without having to intervene in the unit code, but simply acting on the "*time list*" attribute of the communication.

In practice, when coupling codes with a different time step, we can adopt different strategies.

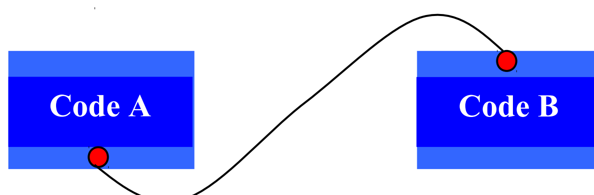
21.2.2 Two-ways coupling a.k.a. strong coupling



Two-ways coupling: *Units A and B run simultaneously: the synchronisation is ensured by the communications*

In general, in a strong coupling, *i.e.* a two ways exchange in an iterative process, the user has to choose the exchange frequencies looking for the least common multiple of the model time steps. Notice that in this case the two models have to run in parallel on two separate branches.
Since `PALM_Get` is blocking and the exchanges are two ways, the communications implicitly synchronise the execution, because each code waits the results from the other one. The time stamp attribute, therefore, is not strictly necessary but it is nevertheless recommended if we want the unit to be generic and reusable in other couplings. Activating the time attribute of an object in the identity card still allow to choose between two ways of describing the coupling exchanges. Either the user let the model know the initial time, final time and frequency of the exchange (via a configuration/input file, the PrePALM constants or PALM communications) and the model issues only the strictly necessary `PALM_Get/Put` with the `PL_NO_TIME` stamp, or the model systematically issues its `PALM_Get/Put`'s at every time step, with a meaningful time stamp and the "*time list*" attribute of the communication, acting as a filter, select which exchanges will actually take place.
Even if the *time list* syntax allows to associate different values of the time stamp on the source and target side of a communications, it is nevertheless recommended to find a common time reference and to use the same timing conventions for all the units participating in communications.

21.2.3 One-way coupling a.k.a. forcing



One-way coupling: *Data flow only from unit A to unit B. If the exchanges concern different instants it is important to associate a "time stamp" attribute, to the object to distinguish different temporal instances of the objects.*

The one-way coupling, or forcing, is slightly different. In this case the data exchange is always directed from one unit (*source*) to a second one (*target*). In principle, the target can be on the same branch as the source and begin running after completion of the source.
If the objects are all produced without a time stamp (attribute `PL_NO_TIME`), the first objects risk to be overridden by the most recent versions before being received, since the `PALM_Put` primitive is non-blocking. Using a *time stamp* different from `PL_NO_TIME`, there is no risk of overwriting the different temporal instances. The user should nevertheless pay attention not to saturate the internal PALM memory storage (called *mailbuff*) with the objects temporary stored while waiting for their consumption. Do not forget that a coupling working on given platform could lead to run off of memory on another machine. Exploiting the branch parallelism and with some optimised synchronisation (further specific communications or blocking steps. *Cf.* Chapter 8) the user can easily find a viable alternative.

21.2.4 Conversion of integer time stamps from/to dates

Especially in geophysics applications, where the exchanged fields are naturally associated to dates, the user can choose to give to the integer time stamp the meaning of a date. To do that, one has to choose a reference date, a time unit (ranging from seconds to days) and the kind of calendar to count on. The integer time stamp will simply indicate the number of time units separating the reference date from the current date, counted on the given calendar.
The reference date, the unit and the calendar are chosen once for all the units in the graphical user interface. Afterwards a PALM primitive can compute the date to/from integer conversion in the unit sources and the corresponding calculator is also available in the graphical user interface to express the *time list* ranges for the communications in a coherent way.
In practice, when setting up the application, the user has to choose in the `Date Conversion` menu the `Calendar` amongst `Standard` (usual Gregorian calendar for which every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100; the centurial years that are exactly divisible by 400 are still leap years), `NoLeap` (for which all the years are 365 days long, neglecting leap years), `360` (for which all the months are 30 days long, used in some climate simulations), `Julian` (for which every year that is exactly divisible by four is a leap year, neglecting the centurial years exceptions). Afterwards he chooses the time unit in the `Reference time step` list from seconds to days. Let's recall that if the hour is chosen, in the time stamps, two consecutive integers will refer to objects separated in time by an hour. Finally he has to set the `Reference date` for the conversion (detailed up to the time unit). It is the origin of the counter and it can be optionally shifted from 0 (`First integer for this date` field).
In the unit code the `PALM_Time_convert` primitive (*cf.* the reference guide for the full syntax) will convert in a unique way every date to an integer and vice-versa. There is no reference to the calendar, the time unit or the reference date in the primitive arguments. Changing the PrePALM settings will change the results (which allows to reuse the unit as it is in different couplings), but the main point is that in a given applications, all the units will obtain the same integer for the same date. In the graphical user interface, the user needs to know the integer time stamps that will be used when describing the time range lists in the communications properties or when programming the event driven actions in Steplang. For this reason, in the `Date conversion` menu, he can access a date converter that gives the same results as the primitive calls. The converter can also compute time ranges: a typical example could be the computation of the integer time stamps expressed in days to indicate the first day of the month over 50 years.

21.3 Dynamic verbosity settings

The PALM verbosity settings are quite detailed because finding a good trade off between information and output volume can greatly help in debugging an application. Indeed, if the verbosity levels are high, the PALM output files (`palmdriver.log` and branch`_XXX.log`) can become really bulky. Even if you can select the verbosity level per message category in PrePALM, the output can still be too heavy if the problem appears quite late in a long simulation. To avoid this situation, PALM implements some primitives that allows to modify the verbosity levels at run-time from inside the codes.
Once you have bounded the region to debug you can raise the verbosity level for the relevant categories when entering the region and lower it again afterwards with simple calls to **`PALM_Verblevel_set`**`(int category, int *level)`. Remember that these calls are useful only during the implementation and set up of an application but they should disappear in the production version because all the ascii output can slow down the application, especially on supercomputers.

21.4 Checking the object contents: palm_debug.f90/c

You'll have certainly noticed that the Make PALM files menu creates a `palm_debug.f90` or `palm_debug.c` template file and that in the communication properties window there is a `Palm debug status` selector. These tools can be used to check the objects contents when they are sent or received. Accordingly to the choice for `Palm debug status` (`PL_DEBUG_ON_SEND`, `PL_DEBUG_ON_RECV` or `PL_DEBUG_ON_BOTH`), the user defined procedures implemented in `palm_debug.f90` or `palm_debug.c`, depending on the language preferred by the user, are invoked respectively when the object is sent, received or on both sides.
Since the same subprogram is invoked for all the “debugged” applications, the appropriate procedures have to be triggered on the basis of the object name and space. The examples and comments in `palm_debug.f90/c` are explicit enough to draw inspiration. Let's recall that it is recommended to use this feature instead of intervening in the units sources.

Aside from debugging, this feature finds another application for run-time sanity checks: checking the exchanged objects against physical coherency tests (e.g. out of physical range values) can help detecting anomalies in the simulation and stopping the application (by a call to `PALM_Abort`).

21.5 Print out the object contents: the PALM_Dump primitive

The PALM_Dump primitive, usually called from inside palm_debug but also available for calls in the units sources, provides a simple way to print out some information about object contents such as its minimum or maximum value, the global sum, etc.
The syntax of the PALM_Dump primitive is described in the reference guide section.

21.6 Summary of the main concepts

In this section you have got to learn or revise some practical features that, even if not strictly necessary to set up and run a PALM application, can make it quite easier. In particular you have seen how to set a default value for a unit input or to restrain the possible values to a closed list, you have learnt how to associate the integer time stamps to dates and finally you have seen how to set the verbosity levels or check the objects contents to debug an application or to make it more robust.

22 Batch file for PrePALM

It is possible to generate PrePALM files without the GUI. For this, the user must construct a file with the extension .pml (PrePALM Meta Language). As documentation, some examples are provided in different “corrige” directories from training session directory. Is given below the .pml file to achieve the tutorial session_8.

To test this example :
> cd training/session_8
> prepalm corrige/session_8.pml
> 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 Palm Glossary

Action: steplang language instruction to be executed on an event; for example: deletion of an object stored in the buffer, or set the status of an assembled object to ready. *Cf.* § 8.4

Algebra: pre-defined unit for algebraic algorithms. The algebraic operations (linear combinations, linear systems solving, eigenvalues and eigenvectors computing, minimisers, ...) are implemented in the the PrePALM toolbox and can be used as any other unit in a coupling. *Cf.* Chapter 6

Application: a PALM application is the collection of elementary units plus the main driver that cooperate to execute a given algorithm, by starting the necessary tasks and performing the needed data exchanges between these components.

Barrier: can be part of a PrePalm STEP: barriers are used for the synchronisation of parallel applications. In order to synchronize two or more branches it is possible to force a rendezvous. A barrier is enabled by the attribute `PL_BARRIER_ON` of a STEP primitive invoked by the concerned branches. The branches will be blocked until every concerned branch has reached the step. *Cf.* § 4.2

Block: collection of several Palm units and control structures in a single executable file. *Cf.* Chapter 3.

Branch: a branch is a component of a PALM simulation. It is used for the description of the coupling algorithm. Several branches can be executed in parallel or sequentially. Each branch can start at the beginning of the simulation, or later on. A branch contains variables declarations and control structures. *Cf.* Chapters 1 and 4.

Buffer: memory area belonging to the main process: the Palm driver (`palm_main`). The buffer has to be considered as a common storage space accessible by all units. It can be distributed, spanning several processors and is dynamically managed, with allocations and de-allocations triggered during the algorithm execution. It allows an explicit management of the objects exchanged between units, like an interpolation, or a composition. *Cf.* Chapter 8.

Category: in Prepalm the bottom left pane displays the attributes of different entities, grouped in homogeneous collections called categories. With the top left selector you can switch the displayed category. *Cf.* § 1.2.

Communication: the mean for a unit (or a branch) to receive (get) or to release (put) an object. The PALM paradigm is based on end-point communications. A unit simply notifies that an object is asked (`PALM_Get`) or made available (`PALM_Put`). The user defines the correspondence between the two sides of the communication via the PrePALM interface. *Cf.* Chapter 5.

Computing code: source code of the program, written in a high level language like FORTRAN, C or C++.

Constant: constants can be defined in Prepalm. They may be needed for the algorithm definition in the branch codes and in several menus (time or tag ranges of a communication). Moreover, they can be used inside the units source code by the inclusion of a language dependent file generated by PrePALM. *Cf.* Chapter 3.

Control Structures: control structures like loops or logical conditions can be used to describe complex algorithms in PrePALM branches.

Coupling: action of executing two or several programming codes in a single application. PALM provides the possibility to launch these codes simultaneously or in sequence, and to control all data exchanges between them.

Daemon: background process dedicated to a specific service. LAM/MPI uses the `lamd` daemon, started by the `lamboot` command.

Derived Data Type: composite data type defined with elementary data types. *Cf.* Chapter 7.

Distribution function: a distributor can be described at run-time. In this case the ID card indicates the name of the user provided function which describes the distributor. *Cf.* Chapter 11.

Distributor: integer list or subroutine, written in a specific format interpreted by PALM, describing the way an array is distributed on processes in parallel communications. *Cf.* Chapter 11.

Driver: the PALM driver (`palm_main`) is the main program of the application. The tasks of the driver are: 1) control the execution of the branches with the dynamic launching of units and blocks or other branches, and 2) answer the requests from the executable files for all communications.

Dynamic Object: an object having a dynamic space. *Cf.* Chapter 9.

Dynamic Space: a PALM space is dynamic if the space size is known only at execution time. The size can also change during the simulation. *Cf.* Chapter 9.

Event: an event is the triggering condition for an action to be performed on objects stored in the Buffer as described in Steplang. An event can be the completion of a communication or a Step explicitly defined by the user. *Cf.* § 8.4.

Fortran90 Region: in the PrePALM graphical interface, the FORTRAN regions allow to execute FORTRAN instructions in the application branches. The source code of these regions must be in FORTRAN 90. *Cf.* § 1.4.

Granularity: in parallel programming, granularity refers to the size (in memory or in CPU time) of a chunk of code executed between two communications or two synchronizations.

Hardwired value: ability of the graphical interface to define a valid FORTRAN expression as the value returned by the PALM_Get function (with a right click of the mouse on the plug of the object). *Cf.* § 5.5.

ID Card: file used by the Prepalm graphical interface for the description of the Palm units properties: spaces, input/output objects and distributors. *Cf.* §2.4.

Inheritance: in the PrePALM graphical interface, a space can inherit the characteristics of another space. This is especially useful for spaces having a NULL type: it is then necessary to define a communication with PrePALM that transfers to this NULL space the characteristics of another space. *Cf.* Chapter 9.

Library: collection of subroutines compiled independently and gathered in a single file.

Localisation: PrePALM entity used to specify the processes involved in a distributor. Cf. § 11.6.

Mailbuff: in order to grant a full independence between the order of objects production and reception, the produced objects which are not immediately consumed have to be stored in a memory space acting as a mailbox. To avoid confusions with the MPI mailbox, this temporary storage space has been renamed *mailbuff* because it shares its memory location with the buffer.

Makefile: input file for the `make` utility containing all information needed to compile a PALM application. For a PALM application, this file is automatically generated by the PrePALM graphical interface.

Memory Slave: slave process launched by the PALM driver in order to extend the mailbuff memory needed by the application (for example for storing temporary objects). *Cf.* § 8.5.

Modularity: characteristic of an application describing its ability to be easily reorganized or to be re-used by parts.

MPI 1: the standard message passing library on top of which PALM is built. The version 1 of MPI is largely widespread and every supercomputer constructor provides an optimised version of MPI1. The MPI1 standard covers the need of SPMD applications. Further details can be found in the official MPI web site http://www-unix.mcs.anl.gov/mpi/index.html

MPI 2: the new extended standard of the message passing library MPI. It covers topics on process management, one-sided communications and parallel I/O which were not addressed in the MPI1 standard. This version is needed for the implementation of MPMD applications. Further details can be found in the official MPI web site http://www-unix.mcs.anl.gov/mpi/index.html

MPMD: parallel programming algorithm in which the parallel applications consists in a collection of independent programs executing concurrently. Processes can join or leave the application dynamically. The acronym comes from **M**ultiple **P**rogram **M**ultiple **D**ata.

Object: PALM objects are the data entities managed by the coupler in the exchanges of information between units. Objects are identified by their name. *Cf.* Chapter 5.

PALM: french acronym meaning "**P**rojet d'**A**ssimilation par **L**ogiciel **M**ultiméthode" which can be translated as "data assimilation project by multi-method software" or transposed to "Parallel Applications with a Lot of Modularity": this is the coupler described in this documentation.

PALM primitive/subroutine/function: any of the subroutines included in the PALM library, which can be called in the user code source.

to Palm: action to modify a computing code in order to use the PALM coupler.

Parallel Communication: communication involving at least one distributed object. *Cf.* Chapter 11.

Parallel Unit: single application component executing concurrently on several processors (SPMD).

Parameters: see Constants.

Plug: in PrePALM, the plugs are in fact small disks placed on top (input Get) or under (output Put) the rectangle representing a PALM unit. Each plug refers to an object being transferred between the units. *Cf.* Chapter 5.

Predefined Unit: PALM unit, accessible in the graphical user interface, which can be used with no modification in an application. For example a grid interpolation unit. *Cf.* Chapter 6.

PrePALM: the PALM graphical user interface. The PrePALM role is to build the application overall algorithm. All the information which does not depend on the algorithm and on the specific application is coded in the units and all the information on the algorithm and on the application is entered via PrePALM.
PrePALM helps filling the `.ppl` file and generates the files needed by the run-time application.
Since PrePALM is the user interface, most of this document explains the PrePALM usage. PrePALM is written in Tcl/Tk.

Priority: parameter associated to a unit or a block, used by the driver to schedule the executions in the most suitable order. When there is a shortage of processors, the unit with highest priority is executed first. When two units have equal priorities, the execution order is indifferent (and unpredictable). *Cf.* § 2.8.

Process: instance of a computing code being executed. A single processor can handle many processes simultaneously.

Process Associations: this concept refers to parallel communications and distributors: the process associations are needed in some non-trivial cases, in order to pass to the driver the exact description, process by process, of the data flow between PALM units. For simple cases, process associations can be automatically deduced from the localisation of the objects. *Cf.* § 11.6.

Processor: electronic component executing the computing code

Replay: function of the graphical interface allowing a visualisation of the sequential execution of all application components. *Cf.* § 2.8.

Resources: number of processes or memory size needed for an application. PALM manages a part of these resources.

Run Time Monitoring: function of the graphical interface allowing seeing in real time which unit, block or branch is currently executed. *Cf.* § 8.3.

Script: file containing system commands (written in a shell language like csh, ksh, etc...). PALM can start scripts directly from the branches. *Cf.* § 4.3.

Service files: service files are subroutines (FORTRAN or C) generated by the PrePALM graphical interface.

Shape: size of each dimension of a multidimensional array. It is one of the properties which define a space.

Space: a PALM space is an entity identified by its name, containing the description of the PALM objects like the data type and dimensions of an array. Spaces are local to units and therefore they

are described in the units ID cards. Notice that in the definition of the shape and of the element size (in the case of derived data types) the constants defined via PrePALM can be used. *Cf.* Chapter 5.

SPMD: parallel programming paradigm in which the parallel applications consists in a single program executing on more than one process. The different instances of the program can perform different tasks or handle different portions of data. The acronym means **S**ingle **P**rogram **M**ultiple **D**ata.

Step: relevant point where to perform some actions on buffer objects or to synchronize executions in several branches. The actions to perform when a step is reached are described in the step-actions section of the `.ppl` file or via the PrePALM interface. Actions can be the invocation of an algebra unit, or a synchronization barrier. *Cf.* § 4.2 and § 8.4.

Steplang: programming language, interpreted by the PALM driver, describing the actions to be executed on the BUFFER's objects. *Cf.* § 8.4 and Chapter 10.

Sub-Object: PALM feature allowing to handle only a part of an object. *Cf.* Chapter 12.

Tag: `PALM_Put/Get` attribute allowing differentiating two objects with identical characteristics (the same "name" and "time"). No differentiation can occur if the tag value is "`PL_NO_TAG`". It is a user defined integer. *Cf.* Chapter 5.

Time stamp: `PALM_Put/Get` attribute allowing differentiating two temporal instances of a given object. No differentiation can occur if the tag value is "`PL_NO_TIME`". *Cf.* Chapter 5.

Unit: the unit is the basic element used to build an algorithm with PALM. Units can be user defined or provided with the PALM algebraic toolbox. A unit can be a serial or a parallel code. Each unit is described by an ID card which lists the properties of the spaces, of the input and output objects and of the distributors used by the unit. *Cf.* Chapter 2.

Verbosity: level to be defined for the number of PALM messages in the output files. Can be modified at run-time. *Cf.* § 21.3.

24 Reference guide of the PALM primitives

24.1 C and Fortran formulation

Dependence on the programming language:

<u>Call from C/C++</u>:

```
int il_err ;
il_err = PALM_Example(arg1,arg2,…) ;
```

<u>Call from FORTRAN</u>:

```
integer il_err
CALL PALM_Example (arg1, arg2, …, il_err)
```

Remark: in the following the PALM primitives are written in the C format. For a FORTRAN use, do not forget to add the final return code integer argument

Application control:

int PALM_Abort()

Communications:

int PALM_Get(char *space, char *obj, int *time, int *tag, void *data)

int PALM_Put(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)

With:

space, obj: character strings of length PL_LNAME
time: integer or PL_NO_TIME
tag: integer or PL_NO_TAG
data: array containing the data

Date to integer or integer to date conversions:

int PALM_Time_convert(int * dir,int *day,int *month,int *year,int *hour,int *min,int *sec,int *time)

With:

dir: PL_TIME_INT2DATE or PL_TIME_DATE2INT

Verbosity level definition:

int PALM_Verblevel_overall_set(int *level)

With:

level: integer or PL_VERBLVL_NOTHING, PL_VERBLVL_WARNING, PL_VERBLVL_USRLEVEL.

and

int PALM_Verblevel_get(int category, int *level)

int PALM_Verblevel_set(int category, int *level)

With:

category: PL_VERB_BRANCH, PL_VERB_UNIT, PL_VERB_COMM,
PL_VERB_STEP or PL_VERB_GENERIC
level : 0, 10, 20, 30, 40 or 50

Messages in PALM outputs:

Writing messages:
void PALM_Write
Only for C and C++: the function PALM_Write has the same format as fprintf , in order to be close to the Fortran: write(PL_OUT, …
Example: PALM_Write(PL_OUT, ‘’message %i’’,23) ;
Maximum message size: 1024 characters

To force the actual writing of bufferised output on disk
void PALM_Flush(PL_OUT);

N.B. In FORTRAN you simply use the PL_OUT unit number in regular WRITE and FLUSH calls.

Checking the content of objects:
int PALM_Dump(int op, char *space, char *obj, int time, int tag, void *data)

With:

space, obj: character strings of length PL_LNAME
time: integer or PL_NO_TIME
tag: integer or PL_NO_TAG
data: array containing the data
op: PL_DUMP_MIN, PL_DUMP_MAX, PL_DUMP_SUM, PL_DUMP_ALL

Derived data types management:

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)

With:

space, item: character strings of length: PL_LNAME
buffer: array containing the packed data
data: array containing the unpacked data of type: item

Dynamic space management:

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)

With:

space, obj: character strings of length PL_LNAME
rank: number of dimensions
shape: integer array of size: rank (define each dimension)

Independent executable dynamic connection (with shared libraries) :

int PALM_Connect()
int PALM_Disconnect()

Others primitives :

int PALM_Get_myname(char *name)

Return the name of the instance of the unit in PrePALM.With name character strings of length PL_LNAME

int PALM_Barrier(MPI_Comm comm)

Allows synchronization on the MPI communicator comm. Based on MPI_Ibarrier and MPI_Test followed by a sleep. The environment variables PALMSPINWAITS and PALMNANOSLEEP have an influence on the behavior of this primitive.

PALMSPINWAITS : Number of MPI_Test attempts before sleep mode.
PALMNANOSLEEP : Passive waiting time in microseconds.

int PALM_Unit_set_progress(float *progress)

With :

progress : value in the [0:1] interval to inform PALM of the unit's degree of progress. Visible in PrePALM rider.

24.2 Python formulation

Python does not allow passing fundamental data types (int, float) by reference, which makes it impossible to write the functions in the same way as in C and Fortran.
In Python, a class is used to group all the input/output attributes for the function calls. The Palm primitives are methods called on this object. In order to get a clear understanding of how to use the class, check the contents of the chapter dedicated to Python.
Inside the palm module there is a class called PalmObject for communication, a class TimeConvert for time and date conversions and some primitives not associated with a class.

PalmObject class:

Public attributes:

object
string of length PL_LNAME containing the object name

space
string of length PL_LNAME containing the space name for transmission/reception operations
for dynamic spaces, *space* shall be set to "NULL"
used only by the functions space_set_shape, put and get

dynspace
string of length PL_LNAME containing the space name for operations concerning dynamic spaces
used only by the functions object_get_spacename, space_get_rank and space_get_shape

rank
integer defining the dimension count of the data array

shape
numpy integer array of dimension *rank* defining the shape of the data.
When setting the attribute, any input format among integer, tuple, list or array will be converted automatically to a numpy object.
The return type is always a numpy array.

time
integer defining the time stamp for the exchange

tag
integer defining the additional criteria *tag* for the exchange

Public methods:

Creator:
PalmObject(object='', space='', rank=0, shape=None, time=PL_NO_TIME, tag=PL_NO_TAG)

A new PalmObject for Palm communication is created from the given parameters. For any undefined parameter a default value is used (see below). Before using put and get methods, at least the attributes object and space must be set, for dynamic spaces, the rank and shape parameters must be obtained.

arguments:

object, space: string of size less than PL_LNAME
rank: integer
shape: integer or tuple, list or array of integers
time, tag: integers, default to PL_NO_TIME/TAG

PalmObject.space_set_shape()

Set the shape of a dynamic space The space, rank and shape attributes must be set before a call to this method.

arguments: none
return values: error status *il_err*
input attributes:

space: name of the dynamic space
rank: dimension of the data
shape: shape of the data

PalmObject.put(ndarray io_object)

Send data via current Palm Object
arguments: numpy object containing the data buffer to be sent
return value: error status *il_err*
input attributes:

object: name of the OpenPALM object
space: name of the space ('NULL' for a dynamic space)
time, tag: individual exchange identification

PalmObject.get(ndarray io_object)

Receive data via current Palm Object.
arguments: numpy object with a buffer large enough to receive the data. In case of dynamic spaces, the required size can be determined with a call of the functions *space_get_rank* and *space_get_shape*
return value: error status *il_err*
input attributes:

object: name of the OpenPALM object
space: name of the space ('NULL' for a dynamic space)
time, tag: individual exchange identification

The methods below are used for dynamic spaces, they use a dedicated attribute *dynspace* for the dynamic space name while the attribute *space* can remain set to 'NULL'

PalmObject.object_get_spacename()
Retrieve the space name of a dynamic object.
arguments: none
return value: error status *il_err*
input attributes:
object: name of the OpenPALM object
output attributes:
dynspace: receives the name of the dynamic space

PalmObject.space_get_rank()
Retrieve the rank of a dynamic space.
arguments: none
return value: error status *il_err*
input attributes:
dynspace: name of the dynamic space (different from 'NULL')
output attributes:
rank: receives the rank of the dynamic space

PalmObject.space_get_shape()
Retrieve the shape of a dynamic space.
arguments: none
return value: error status *il_err*
input attributes:
dynspace: name of the dynamic space (different from 'NULL')
rank: rank of the dynamic space
output attributes:
shape: receives the shape of the dynamic space

PalmObject.dump(int operation, ndarray data)
Dump contents of object into PL_OUT
arguments:
operation: operation to be performed on the data, among:
PL_DUMP_MIN, PL_DUMP_MAX, PL_DUMP_SUM, PL_DUMP_ALL
the operations can be combined by arithmetic summation
(PL_DUMP_MAX+PL_DUMP_SUM)
data: data to be dumped
return value: error status *il_err*
input attributes:
space: space name
object: object name
time, tag: individual exchange identification
output attributes: none

TimeConvert class:

Public attributes:

jour, mois, an, heure, min, sec: integers (day, month, year, hour, min, sec) defining the date in common format
time: integer encoding the date in Palm conventions according to PrePALM definitions.

The conversion is done automatically when writing or reading the attributes.

Public methods:

TimeConvert.convert_time()
    Manual call of the date conversion routine (OpenPALM -> calendar).
    arguments: none
    return value: error status *il_err*
TimeConvert.convert_to_time()
    Manual call of the date conversion routine (calendar -> OpenPALM).
    arguments: none
    return value: error status *il_err*

Primitives without class:

init(char *unit_name)
    Initialize an OpenPALM session.This command opens a connection to an OpenPALM session. It is used in the service files generated by PrePALM and usually need not be called manually.
    arguments: none
    return value: error status *il_err*
finalize()
    Finalize the OpenPALM session. This command closes the current OpenPALM session. It is called at the end of the service files generated by PrePALM and usually need not be called manually.
    arguments: none
    return value: error status *il_err*
get_mycomm(Comm application_comm)
    Return MPI communicator used by PALM. This function can be used to obtain the MPI communicator associated with the PALM execution. This communicator must be used by parallel applications running in OpenPALM environment.
    arguments: MPI Comm object into which the communicator is written
    return value: error status *il_err*
abort()
    Abort a PALM session. This command will be called on all running parallel processes and terminates the application.
    arguments: none
    return value: error status *il_err*
freeproc_nb()
    Get the number of available processors. This formulation is more suitable to Python than *get_freeproc_nb*.
    arguments: none
    return value: number of free processors

get_freeproc_nb(ndarray nbproc)
arguments: nbproc: numpy array to be filled with the processor count by the function
return value: error status *il_err*

write(char *string)
Writes a message into the Palm output file of the current branch (PL_OUT)
arguments: string to be printed
return value: error status *il_err*

space_get_size(char *spacename)
Get the size of a Palm space or a derived Palm space for correct sizing of the local arrays.
arguments: spacename: nom de l'espace (chaine de longueur PL_LNAME)
return value: size of the specified space

pack(ndarray buffer, char *spacename, char *item, int position, ndarray data)
Pack data. This method will store a piece of data at a defined position in a data structure. The structure is defined by instructions in the module's ID card.
A numpy array must be created in the correct size. The required size can be checked with a call to space_get_size.
arguments:
buffer: numpy array into which the data shall be packed
spacename: name of the palm space (string of length PL_LNAME)
item: name of the structure's member (string of length PL_LNAME)
position: Index of the element to be stored in case of an array (0 based)
data: numpy array containing the data to be packed
return value: error status *il_err*

unpack(ndarray buffer, char *spacename, char *item, int position, ndarray data)
Unpack data. This method will extract a piece of data at a defined position from a data structure. The structure is defined by instructions in the module's ID card.
arguments:
buffer: numpy array from which the data shall be extracted
spacename: name of the palm space (string of length PL_LNAME)
item: name of the structure's member (string of length PL_LNAME)
position: Index of the element to be stored in case of an array (0 based)
data: numpy array to which the data is written
return value: error status *il_err*

Not yet interfaced with Python:

connect, disconnect, query_put, query_get, verblevel_set, verblevel_get, flush

The functions below exist, to keep a similar interface to C and Fortran. However, it is recommended to use the methods on the PalmObject class in Python.

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 List of PCW primitives for the CWIPI library

25.1 C and Fortran Formulation

Dependence on the programming language:

Call from C/C++:

int il_err ;
il_err = PCW_*Example*(arg1,arg2,…) ;

Call from FORTRAN:

integer il_err
CALL PCW_*Example* (arg1, arg2, …, **il_err**)

Remark: in the following the PALM primitives are written in the C format. For a FORTRAN use, do not forget to add the final return code integer argument

CWIPI coupling control:

int PCW_Init()
Initialisation of the CWIPI library and setting up redirection of CWIPI output into the OpenPALM log files. Synchronisation point between all OpenPALM units which use CWIPI.

int PCW_Init_tned(int id_flag, MPI_Comm *id_outcomm)
Same as PCW_Init, but the id_flag parameter (value PL_CWIPI_ON or PL_CWIPI_OFF) allows user to use only some process in the coupling. If the value of id_flag is PL_CWIPI_OFF the other PCW primitives must not be called. The communicator id_outcomm include only processus called with the PL_CWIPI_ON flag.

int PCW_Finalize()
Terminates the CWIPI environment. Synchronisation point.

int PCW_Create_coupling(char *coupling_id, <= coupling identifier
int coupling_type, <= coupling type
int entitiesDim, <= mesh dimension (1, 2 or 3)
double tolerance, <= geometric tolerance for localisation
cwipi_mesh_type_t mesh_type, <= mesh type
cwipi_solver_type_t solver_type, <= solver type
int output_frequency, <= intermediate output frequency
char *output_format, <= format of visualisation files
char *output_format_option, <= options for visualisation files
int nb_locations) <= optionally, if mesh_type = CWIPI_CYCLIC_MESH, number of localisations to store in memory.

This command creates a coupling environment (coupling object) with the following parameters:
- coupling_type
in C/C++:

CWIPI_COUPLING_SEQUENTIAL
CWIPI_COUPLING_PARALLEL_WITH_PARTITIONING
CWIPI_COUPLING_PARALLEL_WITHOUT_PARTITIONNING

in FORTRAN:
CWIPI_CPL_SEQUENTIAL
CWIPI_CPL_PARALLEL_WITH_PART
CWIPI_CPL_PARALLEL_WITHOUT_PART

- mesh_type
CWIPI_STATIC_MESH
CWIPI_CYCLIC_MESS
CWIPI_MOBILE_MESH

- 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 of the coupling object.

int PCW_Define_mesh(char *coupling_id, <= coupling identifier (in)
int n_vertex, <= number of mesh vertices
int n_element <= number of elements
double *coordinates <= coordinates of nodes (x1, y1, z1, x2, y2, z2 ...)
int *connectivity_index <= connectivity definition (length n_elements+1)
int *connectivity <= definition of the elements by node index (length connectivity_index[n_element+1])

This function defines the coupling mesh, the connectivity must be sorted so that the elements appear in order of increasing node count:
1D elements : bars
2D elements : triangles, quadrangles, polygons
3D elements : tetrahedra, pyramids, prisms, hexahedra

int PCW_Set_points_to_locate(char *coupling_id, <= coupling identifier (in)
int *n_vertex <= number of nodes or cell centers to locate
doubles *coordinates) <= coordinates of the points to be located

Localisation of interpolation points at places different from the source mesh (which means at vertices for CELL_VERTEX and at cell centers for CELL_CENTER)

int PCW_Add_polyhedra(char *coupling_id, <= coupling identifier (in)
int nb_elem, <= number of polyhedra to be added (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 n_faces                           <= number of faces in polyhedra
        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]
```

Adds polyhedra to the mesh.

CWIPI communications:

```
int PCW_Sendrecv(char *coupling_id,           <= coupling identifier (in)
        char *exchange_name,                  <= exchange object identifier (in)
        int stride,                           <= number of interlaced fields to exchange (in)
        int time_step,                        <= time step for visualisation (in)
        double time_value,                    <= time value for visualisation (in)
        char *sending_field_name,             <= name of transmitted field (in)
        double *sending_field,                <= transmitted field (in)
        char *receiving_field_name,           <= name of received field (in)
        double *receiving_field,              <= received field (out)
        int *not_located_points)              <= number of non located points (out)
```

Cross exchange of data fields on the two meshes. This primitive is a synchronisation point.
In C/C++ this command can simulate a one way communication as in PCW_Send or PCW_Recv by using a NULL pointer for the received or sent field respectively.

```
int PCW_Recv(char *coupling_id,               <= coupling identifier (in)
        char *exchange_name,                  <= exchange object identifier (in)
        int stride,                           <= number of interlaced fields to exchange (in)
        int time_step,                        <= time step for visualisation (in)
        double time_value,                    <= time value for visualisation (in)
        char *receiving_field_name,           <= name of received field (in)
        double *receiving_field,              <= received field (out)
        int *not_located_points)              <= number of non located points (out)
```

```
int PCW_Send(char *coupling_id,               <= coupling identifier (in)
        char *exchange_name,                  <= exchange object identifier (in)
        int stride,                           <= number of interlaced fields to exchange (in)
        int time_step,                        <= time step for visualisation (in)
        double time_value,                    <= time value for visualisation (in)
        char *sending_field_name,             <= name of transmitted field (in)
        double *sendingfield,                 <= transmitted field (in)
        int *not_located_points)              <= number of non located points (out)
```

```
int PCW_Irecv(char *coupling_id,              <= coupling identifier (in)
        char *exchange_name,                  <= exchange object identifier (in)
        int tag,                              <= MPI tag of the communication (in)
        int stride,                           <= number of interlaced fields to exchange (in)
        int time_step,                        <= time step for visualisation (in)
        double time_value,                    <= time value for visualisation (in)
        char *receiving_field_name,           <= name of received field (in)
```

double *receiving_field, <= received field (out)
int *request) <= request identifier (out)

Initialise data reception via non-blocking communication. The reception is completed with the call of PCW_Wait_irecv.

int PCW_Wait_irecv(char *coupling_id, <= coupling identifier (in)
int request) <= request identifier (in)

int PCW_Issend(char *coupling_id, <= coupling identifier (in)
char *exchange_name, <= exchange object identifier (in)
int tag, <= MPI tag of the communication (in)
int stride, <= number of interlaced fields to exchange (in)
int time_step, <= time step for visualisation (in)
double time_value, <= time value for visualisation (in)
char *sending_field_name, <= name of transmitted field (in)
double *sending_field, <= transmitted field (in)
int *request) <= request identifier (out)

int PCW_Wait_issend(char *coupling_id, <= coupling identifier (in)
int request) <= request identifier (in)

Other primitives:

int PCW_Set_output_listing(int iunit) <= Fortran logic device number

Redirects the CWIPI output into the Fortran logic device defined by its number. By default, the output is written to the files with the name of the branches followed by the process number if the unit is parallel.

int PCW_Dump_application_properties()

Writes the coupling properties to the output file.

int PCW_Locate(char *coupling_id) <= coupling identifier (in)

Explicit localisation (by default, the localisation is done at the first data exchange except for non blocking-communications).

int PCW_Update_location(char *coupling_id) <= coupling identifier (in)

Explicit update of the localisation for mobile meshes.

int PCW_Get_n_not_located_points(char *coupling_id, <= coupling identifier (in)
int *n_not_located_points) <= number of non located points (out)

Gives the number of non located points at the previous PCW_Locate

int PCW_Get_not_located_points(char *coupling_id <= coupling identifier (in)
int n_not_located_points, <= number of non-located points (in)
int *not_located_points) <= indices of non-located points (out)

int PCW_Get_n_located_points(char *coupling_id, <= coupling identifier (in)
int *n_located_points) <= number of located points (out)

Gives the number of located points at the previous PCW_ Locate

int PCW_Get_located_points(char *coupling_id <= coupling identifier (in)
int n_located_points, <= number of located points (in)
int *located_points) <= indices of located points (out)

int PCW_Get_distance_located_points(char *coupling_id <= coupling identifier (in)
float *distance) <= distance to the elements (out)
The distance table must be allocated before calling this function.

int PCW_Dump_notlocatedpoints(char *coupling_id <= coupling identifier (in)
int n_not_located_points) <= number of non located points (in)

int PCW_Reorder(double *field_to_reorder <= field to be re-ordered (inout)
int field_size, <= number of points (in)
int stride, <= number of interlaced data fields (in)
double default_value, <= value to be set at non located points (in)
int *not_located_points, <= indices of non-located points (in)
int n_not_located_points) <= number of non-located points (in)
Re-organises the received fields by setting a default value at the non located points.

Localisations storage, from CWIPI 0.8.0

in memory:
int PCW_Set_location_index(char * coupling_id, <= coupling identifier (in)
int index) <= localisation index (in)

disk:
int PCW_Open_location_file(char * coupling_id, <= coupling identifier (in)
char *filename, <= file name(in)
char *mode) <= 'r' read, 'w' write
Open a file to store the localisations.

int PCW_Close_location_file(char * coupling_id) <= coupling identifier (in)
Close the file.

int PCW_Save_location(char * coupling_id) <= coupling identifier (in)
Store the current localisation, to call after an exchange.

int PCW_Load_location(char * coupling_id) <= coupling identifier (in)
Read the current localisation, to call before an exchange

25.2 Python Formulation

Similarly to the Python interface for Palm (chapter 24.2), the PCW module also uses a class called *Coupling* which groups the attributes and methods for communications and contains independent functions outside of the Coupling class.

Coupling class:

The interface is very similar to C, but the coupling id is saved along with the object, so that the argument char *coupling_id is never passed in the method calls as in C.

Public attributes:

n_not_located_points
integer representing the number of non-located points found during a CWIPI mesh location

not_located_points
numpy integer array of length *n_not_located_points* containing the indices of non-located points

irecv_request
integer, unique identifier for an asynchronous communication (obtained by irecv)
issend_request
integer, unique identifier for an asynchronous communication (obtained by issend)

Public methods:

Creator:
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)
Creates a Coupling object, for parameter description refer to C/FORTRAN manual

Destructor:
~Coupling()
Destroys the CWIPI Coupling object.

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.dump_notlocatedpoints()

Coupling.reorder(ndarray reorder_field, int field_size, int stride, double default_value)

Inserts the non-located points into the array *reorder_field* with their default value. Beforehand, the *get_n_not_located_points* and *get_not_located_points* methods must be called so that the attributes *n_not_located_points* and *not_located_points* are filled with the correct values.

arguments:

reorder_field: field received from CWIPI exchange methods
field_size: length of above-mentioned field
stride: number of interlaced fields
default_value: default value used for all non-located points

input attributes:

n_not_located_points: number of non-located points obtained via *get_n_not_located_points*

not_located_points: index array for non-located points. The reorder method sets all elements corresponding to these indices to the specified default value and shifts back all other values to have a properly aligned field.

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()

Other primitives:

init()

finalize()

set_output_listing(file output_listing)

dump_application_properties()

dump_status(cwipi_exchange_status_t status)

26 Identity Cards syntax

!PALM_UNIT			
Attribute	Description	type	
-name	Unit name	string	
-functions	List of the functions available to the user, and the programming language (F77 for Fortran77, F90 for Fortran90, C for C or C++ for C++) (*)	List	
-object_files	List of objects files (*.o) or libraries (*.a) needed for link phase of the unit	List	
-parallel	Type of parallelism : mpi for MPI omp for OpenMP no for mono-processor units	mpi omp no	option
-minproc	Minimum number of processors	integer	option
-maxproc	Maximum number of processors	integer	option
-comment	Comment	List	option
-class (**)	Allows to specify that the unit is an algebra unit (or not)	algebra or nothing	option
-library (**)	Algebra library name (in case of an algebra unit)	string	option
-mode (**)	Specific information on the execution context, in case of an algebra unit	sticky or no_sticky	option
-label (**)	Short title of the unit, in case of an algebra unit	text	option
-help (**)	detailed information on the unit	text	option

(**) specific attribute defined only for the predefined algebra units.
(*) If several functions are listed, the unit will sequentially execute each of them in the list order.

!PALM_SPACE			
Attribute	Description	type	
-name	Space name	string	
-shape	Array dimensions : the dimensions are separated by comas, everything must be within ()	Integer expression or constants	
-element_size	Size of each element of an array. There is the possibility to define derived data types by combining elementary data types PL_INTEGER, PL_REAL, PL_LOGICAL, PL_DOUBLE_PRECISION, PL_COMPLEX PL_AUTO_SIZE (*)	Integer expressions based on PALM generic constants	
-items	List containing for each element a tuple of two elements : the first one is the item name, the second is a space already defined.	List	(*)
-comment	Comment	List	Option

(*) In case of a space with a derived data type (see next paragraph), if attribute **-items** is used, then attribute **-element_size** must be initialised with value : PL_AUTO_SIZE.

!PALM_OBJECT			
Attribute	Description	Type	
-name	Object name	String	
-space	Space name	String or NULL(1)	
-distributor	Distributor name	String or NULL(2)	Option
-localisation	Localisation name	String or NULL(2)	Option
-time	ON if the communication time field is different of PL_NO_TIME, OFF otherwise	ON/OFF	Option
-tag	ON if the communication tag field is different of PL_NO_TAG, OFF otherwise	ON/OFF	Option
-intent	IN/OUT/INOUT depending if the object is used in PALM_Get, PALM_Put or both.	IN/OUT/ INOUT	
-default	Default value for an object, only for a scalar.	Value depends on space	Option
-closedlist	Only for scalar inputs, pre-defined list of choices for the object values. The syntax is { {val1 comment1} {val2 comment2} … }	List of items	Option
-comment	Comment	List	Option
-rank (3)	Rank of object = number of dimensions of array	Integer	

(1) Objects having a space defined as NULL in the identity card inherit the characteristics of remote objects when the communications are defined in the graphical interface.
(2) NULL for predefined algebra units
(3) Only for predefined algebra units.

!PALM_ DISTRIBUTOR			
Attribute	Description	type	
-name	Distributor name	String	
-type	Distributor type : `Regular` for a distributor of type regular `Custom` for a distributor of type custom `Regular_wh` for a distributor of type regular with halos	String : regular custom regular_wh	
-shape	Profile of the distributed object (span of each dimension, coma separated, embedded within parenthesis)	integers or constants expression	
-nbproc	Number of processes in the distribution	Integer or constant	
-function	Name of the distribution function	String	
-object_files	Name of the file object containing the distribution function.	String	
-comment	Comment	List	option

!PALM_ LOCALISATION			
Attribute	Description	type	
-name	Name of the localisation	String	
-type	Type of the localisation : `Distributed` for a distributed object. `Replicated` for a replicated object.	String : distributed replicated	
-description	List between {} describing the processes involved The list syntax is : **deb1**[**:fin1**[**:stp1**]] [; …]	String	

!PALM_ INTERN_COMM			
Attribute	Description	type	
-source	Name of source object	String	
-target	Name of target object	String	
-time	List of valid timestamps	List	
-tag	List of valid tags	List	
-debug	Launching, or not, the debug function of PALM	PL_DEBUG or PL_NO_DEBUG	
-track	Writing, or not, the communications information in the PALM output files	ON or OFF	
-localassoc	Association of the localisations	String	
-source_so_descriptor	Description of the source sub-object	String	
-target_so_descriptor	Description of the target sub-object	String	
-dtm	Type of memory management for this object	MEMORY or DISK	

This keyword is used for communications internal to a unit. It is necessary when gathering several units into a new single entity. The keyword allows the transfer of the objects between the elementary units composing the new entity without having to redefine the communications.