



PALM_MP software

Training manual

Thierry Morel and the PALM Team

CERFACS/ Global Change and Climate Modelling Team

January 2008

TR/CMGC/08/6

Table of Contents

| | |
|---|----|
| Table of Contents..... | 2 |
| 1 Session 1: Graphic user interface..... | 4 |
| 1.1 Introduction..... | 4 |
| 1.2 Launching PrePALM..... | 4 |
| 1.3 Inserting a branch..... | 5 |
| 1.4 Editing the branch code..... | 6 |
| 1.5 Compilation options setup..... | 7 |
| 1.6 Generating the PALM service files..... | 7 |
| 1.7 Compiling the application and executing..... | 9 |
| 2 Session 2: Launching units..... | 10 |
| 2.1 Introduction..... | 10 |
| 2.2 From a stand alone code to a PALM unit..... | 10 |
| 2.3 An example of a PALM unit..... | 10 |
| 2.4 ID cards..... | 11 |
| 2.5 Loading the units ID cards..... | 12 |
| 2.6 Launching the units..... | 12 |
| 2.7 Parallel computing..... | 14 |
| 2.8 The performances analyser..... | 15 |
| 3 Session 3: the blocks..... | 17 |
| 3.1 Launching the units inside the driver executable..... | 21 |
| 4 Session 4: More about the branches..... | 23 |
| 4.1 Launching by another branch..... | 23 |
| 4.2 The steps..... | 23 |
| 4.3 The scripts..... | 24 |
| 4.4 Launching a MPI parallel unit..... | 25 |
| 4.5 Launching an OpenMP parallel unit..... | 26 |
| 5 Session 5: the communications..... | 28 |
| 5.1 Introduction..... | 28 |
| 5.2 Preparation of the units, the PALM primitives..... | 29 |
| 5.3 The communications in PrePALM..... | 32 |
| 5.4 Time lists..... | 33 |
| 5.5 Hardwiring values..... | 35 |
| 6 Session 6: Predefined units..... | 37 |
| 6.1 Introduction..... | 37 |
| 7 Session 7: Derived data type objects..... | 40 |
| 7.1 Introduction..... | 40 |
| 7.2 Memory contiguous objects..... | 40 |
| 7.3 Non contiguous objects..... | 41 |
| 8 Session 8: Time interpolation..... | 46 |
| 8.1 Introduction..... | 46 |

| | | |
|------|--|----|
| 8.2 | Units Preparation | 46 |
| 8.3 | Monitoring the application in real time | 47 |
| 8.4 | Steps, events and actions..... | 49 |
| 8.5 | The memory slaves | 51 |
| 9 | Session 9: Space inheritance and dynamic objects | 53 |
| 10 | Session 10: Assembling objects in the BUFFER..... | 55 |
| 11 | Session 11: Parallel communications..... | 57 |
| 11.1 | Introduction..... | 57 |
| 11.2 | The distributors | 58 |
| 11.3 | Block cyclic distributors | 59 |
| 11.4 | 'CUSTOM' distributors | 60 |
| 11.5 | The distribution functions | 61 |
| 11.6 | An example of a distributed object | 61 |
| 11.7 | Localisations and process associations | 64 |
| 12 | Session 12: Sub-objects | 67 |
| 13 | Session 13: Read and write in files, geophysical fields interpolation..... | 71 |
| 14 | Session 14: Using a minimiser..... | 75 |
| | Palm Glossary | 79 |
| | List of PALM functions | 84 |
| | Identity Cards..... | 87 |

1 Session 1: Graphic user interface

1.1 Introduction

The use of the PrePALM graphic interface is a mandatory step in the development of a PALM application. It is very important to know all the subtleties of this interface in order to take the maximum advantage of the coupler functions. In general you may spend more time in the PrePALM tool than in modifying the source code of the programs to be coupled. The principal reason of this is due to the fact that the coupling algorithm is entirely described in the graphic user interface. Moreover, most of the coupler functions are defined via the graphic 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 for the definition the PrePALM path and command:

```
alias prepalm 'setenv 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 graphic 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
```

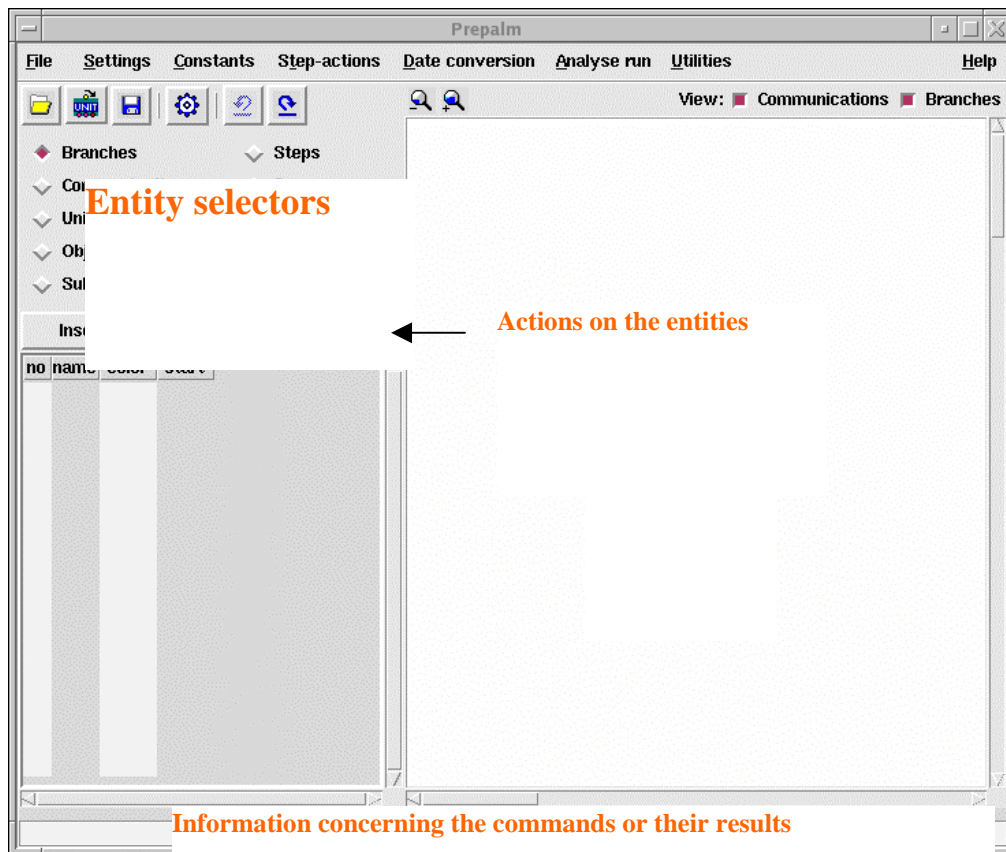
When the alias is defined, you may enter the following command to launch the graphic user interface:

```
> prepalm
```

Workout!

- Move to the directory :cours_palm/session_1
- Launch the **PrePALM** graphic user interface

The first time PrePALM is executed, a dialog box opens to invite the user to be registered. This command sends an email to the coupler development team. You should answer OK to open the main window of PrePALM.



PrePALM graphic 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.

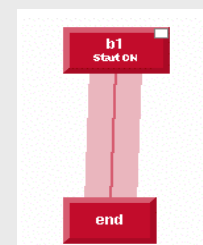
PALM is a dynamic coupler. The sequence of elementary actions follows the logic of a programming language with variables declarations, instructions and control structures (loops and conditional switches): all this is defined in the branches.

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, validate
- > In the canvas, you should see:



The top rectangle materializes the beginning of the branch, the bottom one materializes the end of the branch, and the large line connecting them will be used to materialize the progression between the units.

Do it!

- Double click on one of the two rectangles
- Modify the branch color: you have to guess on 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 graphic user interface
- Move the mouse on the various PrePALM zones to read the help messages.

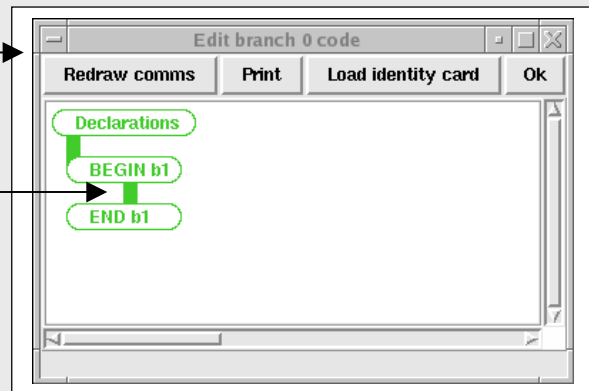
1.4 Editing the branch code

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

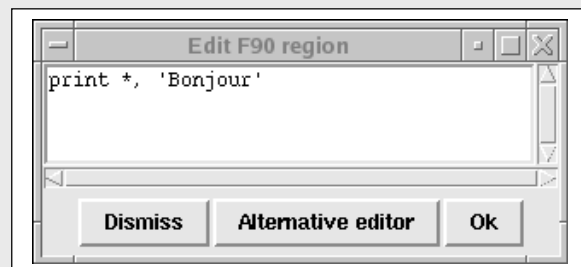


- Click here

- Select

Insert Fortran90 region

- 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 Makefile includes a file (`Make.include`) which is machine dependent. The `Make.include` file can be edited in the graphic user interface. In our case, we will simply read a file containing the appropriate compilation options.


This file is located in the directory `cours_palm/makefiles` and named `linux-32-r4-lam-pgi.mak`

Let's do it!

- Menu Settings => Palm Makefile options edit
- Load_options button
- Load the `linux-32-r4-lam-pgi.mak` file
- Check the box "save as default" then validate

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

Save!

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

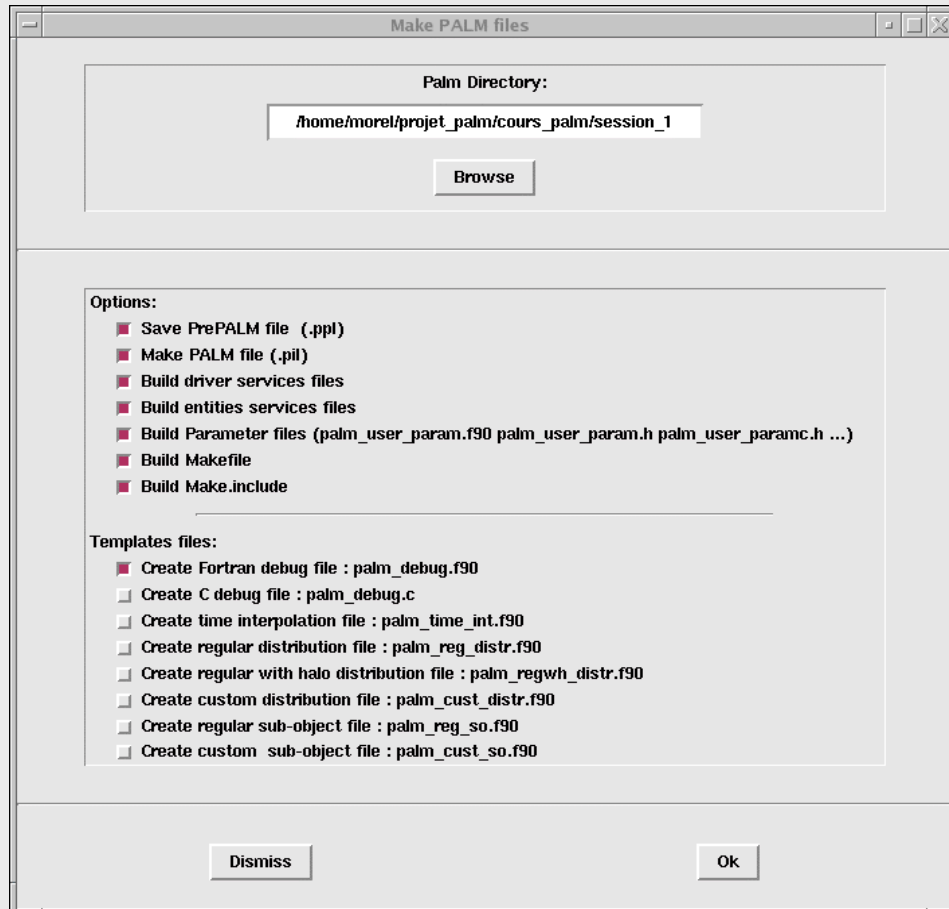
1.6 Generating the PALM service files

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 `cours_palm/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 work on another machine, it will be necessary 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 utility will be detailed later on.
- session_1.pil is a coupler input file: it contains information of the ppl file compiled to be readable by the coupler.

1.7 Compiling the application and executing

The graphic user interface does not launch the PALM application. It has to be compiled and executed on the parallel (or not) computer of your choice. Here (in this tutorial context), we are going to use the same workstation (Linux) to run the graphic user interface and to execute the coupled applications.

Parallel computing and message passing in PALM are based on the MPI2 standard. The implementation of this standard which we will use for the tutorials is LAM MPI. It is necessary, before launching a parallel application with LAM, to launch a daemon. Throughout the sessions you will need some LAM commands. Please note these three commands. They will be useful for you throughout the training:

- > **lamboot** starts the LAMMPI demon
- > **lamhalt** stops the LAMMPI demon
- > **lamclean** reinitialises LAMMPI, useful if the application crashed

Let's try!

- make
- lamboot
- ./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 a condition inside the loop in order to have:

Bonjour 1
 Bonjour 2
 Third bonjour
 Bonjour 4
 Bonjour 5

Note: 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”.

2 Session 2: Launching units

2.1 Introduction

A PALM *unit* is a chunk of user code that can be invoked by calling a function (C, C++) or a FORTRAN subroutine without arguments. 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 communication. The goal is to show how to interface an existing computer code to make a PALM unit.

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 made is thus to replace `PROGRAM` by `SUBROUTINE` (or `main` by another function name). The only constraint is to have the source code of the code to be coupled or at least the main program source code. 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 sources.

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 library include and the writing in 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 graphic 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.

Description of the various fields in our example:

-name `unit77`: gives the unit a generic name. Usually, the names given inside PALM and PREPALM must not contain spaces or 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, 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, this library should be given in the `object_files` field.


See the differences!

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

2.5 Loading the units ID cards

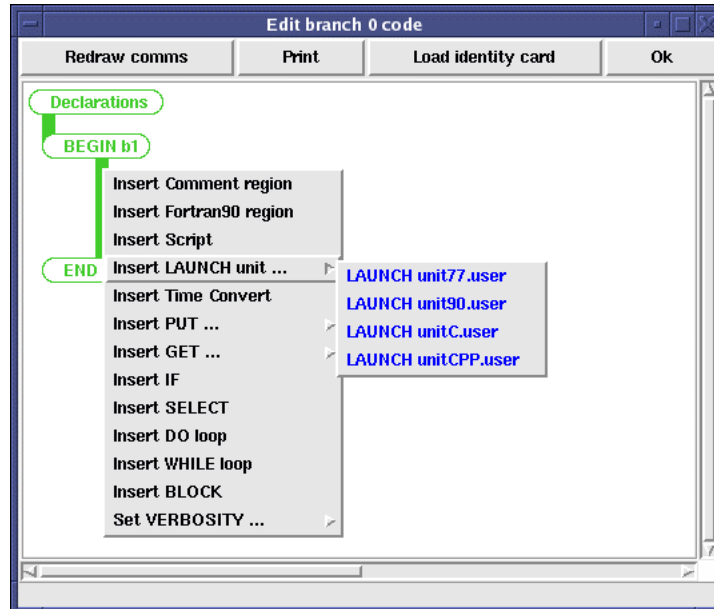
Before launching the units inside PrePALM, it is necessary to load their ID cards.

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 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 => Number of proc: **1**
- Compile and execute **./palm_main**

The result of your "calculation", that is to say the writings in the PL_OUT file, is in the file palm00_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 6 executable files:

- main_unit77
- main_unit90
- main_unitC
- main_unitCPP
- palm_main
- dtm.exe

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

extern int PALM_Init();
extern int PALM_Finalize();
extern void fname(pl_debug) (int*,char*,int*,char*,int*,int*,int*,int*,void*);

int C_MAIN_FOR_FORTRAN(int argc, char **argv, char **envp) {
    int il_err;
    char *cl_exec_name;
    char *cl_wd;
    char cl_wd_envvar[PL_LNAME];
    pvf_debug=&fname(pl_debug);

    if (strchr(argv[0], '/'))
        cl_exec_name = strchr(argv[0], '/');

```

```

else
    cl_exec_name = strdup(argv[0]);
    strncpy(cl_wd_envvar, cl_exec_name, PL_LNAME);
    strcat(cl_wd_envvar, "_WD");
    cl_wd = getenv(cl_wd_envvar);
    if (cl_wd) chdir(cl_wd);

    fname(palm_time_conv_init)();
    il_err = PALM_Init();
    fname(unit90)();
    il_err += PALM_Finalize();
    return (il_err);
}

```

The `unit90` FORTRAN subroutine call was encapsulated in this main program where all initialisations necessary for PALM are made before calling the `unit90` routine. This avoids the user having to add these calls in the units' program source. The `fname()` is a precompilation macro (defined in PALM for the preprocessor) intended to convert automatically the function names to ensure the compatibility between the FORTRAN and C languages.

A PALM application is a 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 have been inserted on only one branch, 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 it is not done, the output of the former executions is kept, and the new outputs will be appended to the former 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 and 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 parallelism which can be simply defined in the graphic user interface by creating several branches. The second is an internal parallelism, i.e. within the units. We will come back later on this second level of parallelism.

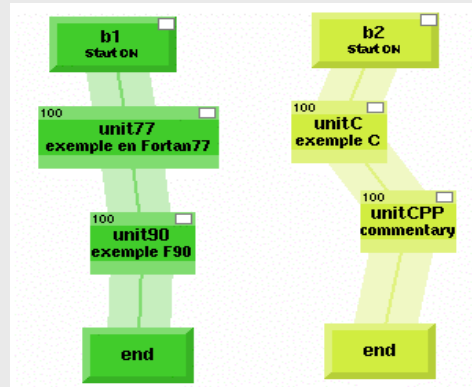
Parallelize it!

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 => Number of proc : 2

Save, service files, make clean, ./palm_main

Bravo!

You have made your first parallel computing almost without realizing it, without anything to know about MPI, only by drawing! It is one of the PALM characteristics: one can make parallel computing without any further specific knowledge. The results are now in two distinct files: the file palm00_000.log contains the branch 1 units output and the file palm01_000.log the branch 2 units output. Open these files to see the results.

2.8 The performances 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!

➤ Settings menu => Palm execution setting, check this box:

Trace execution for animation (file palmperf.log)

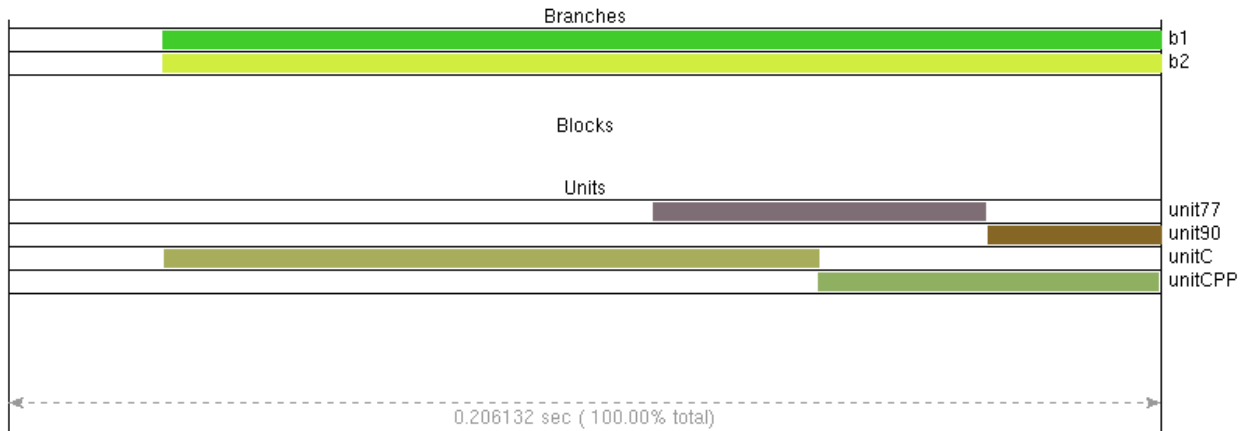
➤ Service files; make clean; make ; ./palm_main

➤ Menu analyze run => Load file: read palmperf.log

➤ A summary report gives you the execution time per unit

➤ Menu analyze run => Play: it allows you to see the launching of the units graphically (find by yourself how to do it!)

➤ Menu analyze run => Performances analyser: produces the following picture

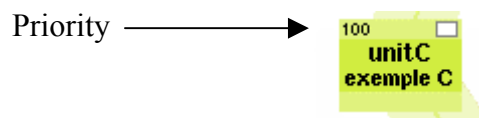


On this example, we can see that unit77 runs at the same time with unitC and unitCPP.

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 with only one process, play with the priorities in order to run these units in this order:

- unit77
- unitC
- unit90
- unitCPP

Check it with the “Play” and with the “the performances analyzer”

3 Session 3: the blocks

For the moment we have seen that each PALM unit generates an independent executable file. For two main reasons (memory usage and launching time optimisation) 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 parameterise the number of launchings of this unit, we will use the PrePALM constants. The constants correspond to the FORTRAN `PARAMETER` declarations, or the C `#define`. Those values are known at compile time and cannot be modified during the execution.

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

The use of constants presents some interest in the graphic user interface itself, but also 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:

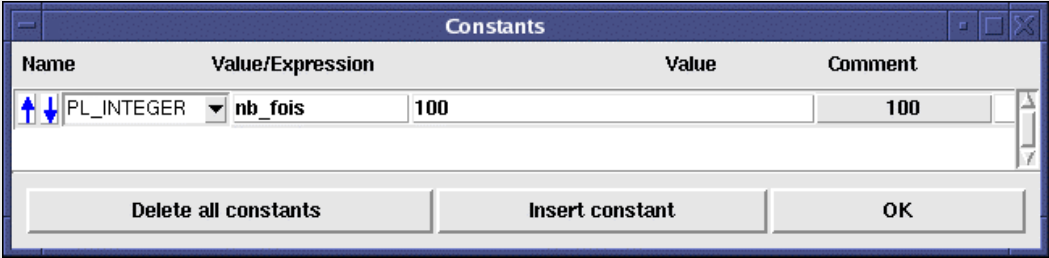
- `palm_user_param.f90` F90 `use palm_user_param`
- `palm_user_param.h` F77 `include 'palm_user_param.h'`
- `palm_user_paramc.h` C,C++ `#include "palm_user_paramc.h"`

These files contain all the same constants. They can be used for example for dimensioning the static arrays.

In our application, we will define only one constant which will be used only in the graphic user interface for the control of the upper bound of the DO-loop.

Be constant!

- Run PrePALM in the directory `session_3`
- Constants menu => constants editor :



- Add an integer constant, give it a name and a value (100)

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}
```

```

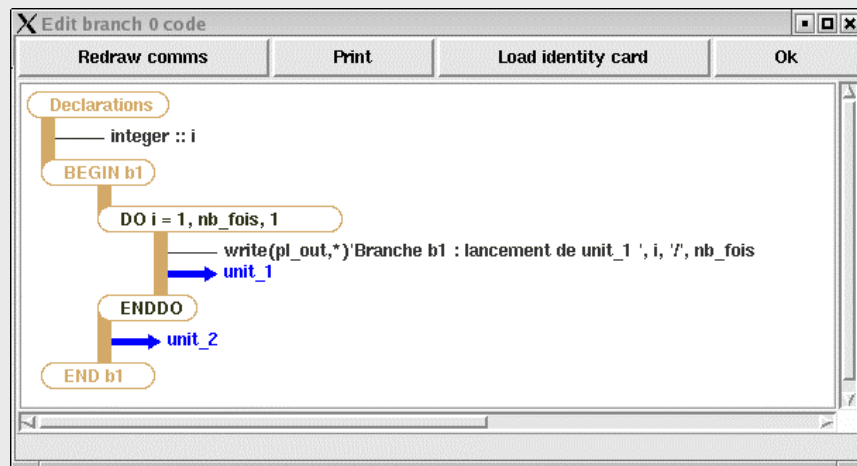
!           -comment {exemple F90}
SUBROUTINE unit_1
  USE palmlib      !*I The PALM interface
  IMPLICIT NONE
  INTEGER :: iglobal
  COMMON /partage/ iglobal
  iglobal = iglobal + 1
  WRITE(PL_OUT,*) ' '
  WRITE(PL_OUT,*) 'UNIT_1 : Bonjour'
  WRITE(PL_OUT,*) 'UNIT_1 : valeur de iglobal: ', iglobal
END SUBROUTINE unit_1

```

This unit prints out the value of a variable (iglobal) which is incremented in 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.

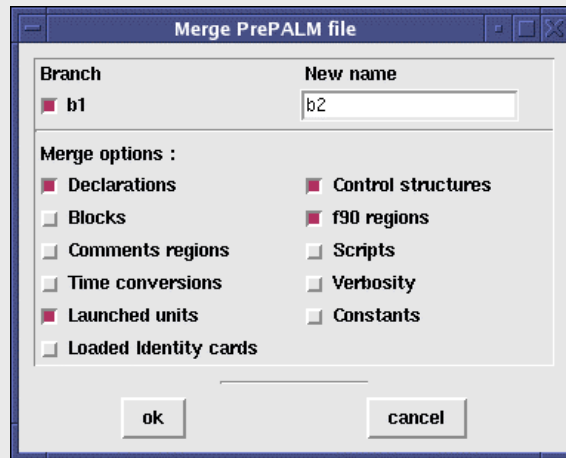


- Save your file.

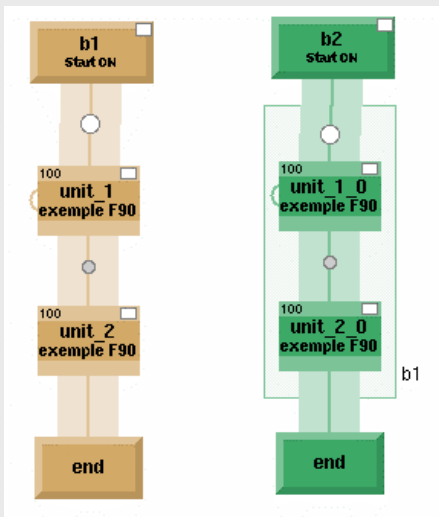
We now will build a second branch. As it will be very similar to the first one, we will use the PrePALM Merge function which makes it possible to transfer a part of an existing application to the current one. In our case, we will use the file which we have just saved.

Copy yourself!

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



- In the canvas duplicate the two 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.

Consequences of the block construct:

1) Sharing the memory

If we look at the file palm00_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.

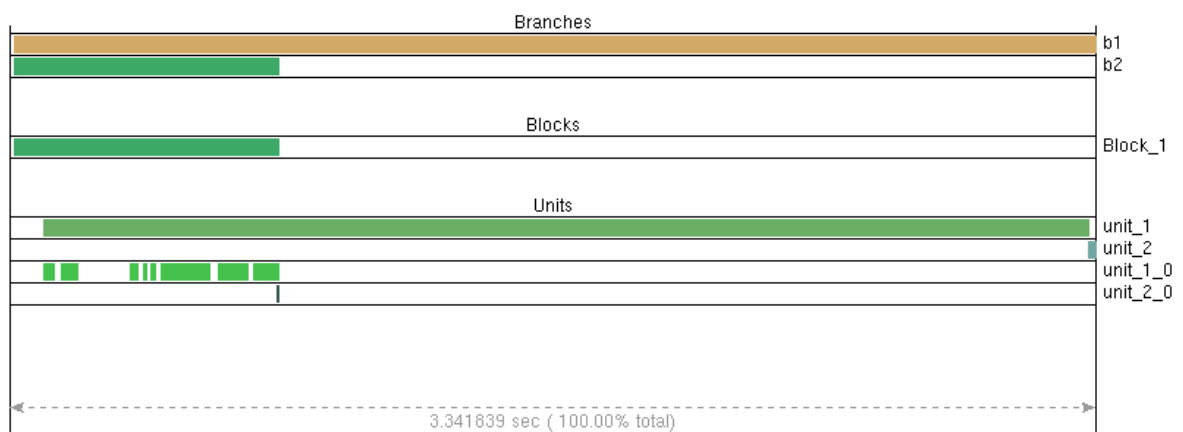
On the other hand in the file palm01_000.log, output file of the branch b2, the value of iglobal is preserved between each launch of unit_1_0. This is normal because now, the launch of a unit corresponds to the call of a subroutine in the main program executing the loop. For the same reason, now the 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. This can be a problem, since the hardware memory size is always limited.

2) Optimising the CPU time

If you open the PrePALM performance analyser for this application, you will see 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 graphic user interface is flexible enough to test easily several configurations. One can see here all the interest to have defined the units like subroutines and not as programs. This leaves all the flexibility to encapsulate the code in blocks at the the graphic user interface level.

Note:

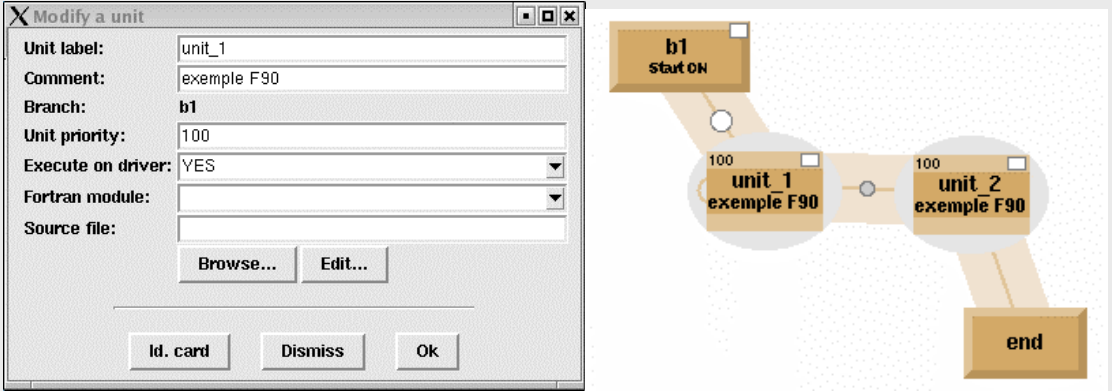
Some computer codes (and they are many, according to our past experience) have real difficulties to run in a loop inside a block, simply because they were not designed for this usage: files are not closed, variables are not deallocated, initialisations are carried out only once... In this case, it is simpler to avoid using the blocks, and this does not prevent us from executing such a code in a loop.

3.1 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 last .ppl
- Delete the second branch
- Edit the 2 units and choose yes for “execute on driver”



The image shows a screenshot of a graphical user interface window titled "Modify a unit". The window contains the following fields and controls:

- Unit label: unit_1
- Comment: exemple F90
- Branch: b1
- Unit priority: 100
- Execute on driver: YES (dropdown menu)
- Fortran module: (empty dropdown menu)
- Source file: (empty text field)
- Buttons: Browse..., Edit..., Id. card, Dismiss, OK

To the right of the dialog box is a diagram illustrating a program flow. It starts with a box labeled "b1 stat ON". A line leads to a box labeled "unit_1 exemple F90" with a greyed ring around it. This is followed by a box labeled "unit_2 exemple F90" also with a greyed ring. The flow ends with a box labeled "end".

- In Setting => palm execution settings, put 0 for the max. process number
- Generate the service files then test the application

Notice that the units which are executed in the PALM driver appear with a greyed ring in the PrePALM canvas. The execution result (writings in file PL_OUT) is 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.

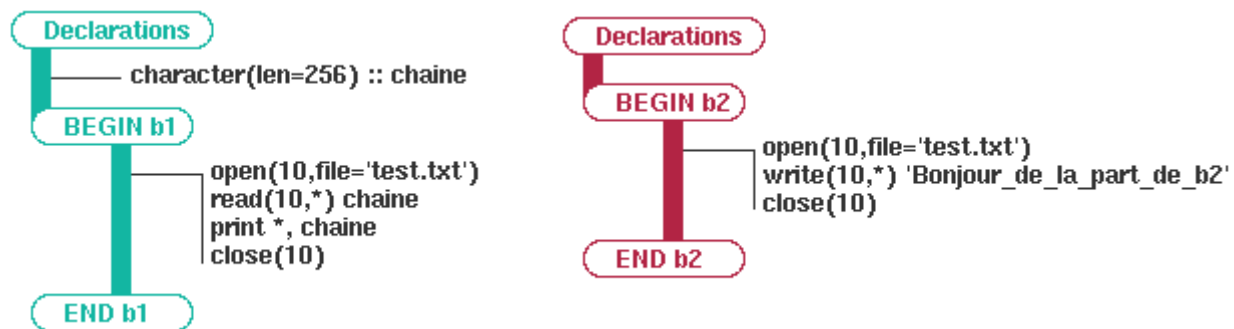
This functionality is also very interesting for parallel applications. It can prevent the need of an additional process. But it can sometimes lead to deadlocks if the application is not correctly synchronized. Indeed, the PALM driver provides two essential functions: launching the units according to the coupling algorithm described in the graphic 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 functions. Therefore launching a unit in the PALM driver may have large 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 execute in the PALM driver. In this case you have to specify “--without-mpi” during the configuration phase of the PALM installation.

4 Session 4: More about the branches

4.1 Launching by another branch

Until now, all the branches we have defined, started 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. These barriers are associated to the PrePALM *steps*. To use a step it is necessary to create it in the graphic user interface. A step may or may not have the attribute BARRIER. 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" 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. In general computer codes read their data in a file having a specific 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, each of our data files in the same file `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 graphic 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 from 1 to 5
- Launch a unit code inside the DO-loop
- Before the launch, insert this script:

```
➤ echo $ib_do  
➤ cp -f fic$ib_do.in don.in
```

- Test your application

The result should be like that:

```
1 premier_jeu_de_donnees  
2  
3 second  
4 troisieme  
5  
6 quatrieme  
7  
8 cinquieme
```

Exercise 5:

Try to make the DO-loop run in a block

What is happening?

Change the code : `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!

- Copy the file `fpi.f` in `pi_mpi.f` and edit it
- Ad 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” by “subroutine `pi_mpi()`”
- Add include file: `'palmlib.h'`
- Comment 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 launch the unit `pi_mpi`
- Set the number of processes 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 best 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 do not know this, 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.

Note:

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 it is possible to run the code under the form of a PALM unit.

5 Session 5: the 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, the first question to ask is: what needs to be exchanged, what are the useful coupling data to be identified in a coupled application? 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 (See 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 have the same space. In the graphic 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 graphic 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 the exercise of code coupling, iterative processes are quite often presents. The objects exchanged in this context reflect for example the temporal evolution of a physical 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 with. For PALM, this time field is simply an integer variable. The user is free to associate this integer to a physical date (if it is your case, please ask the PALM trainers about the calendar conversion tools) or to neglect this attribute by using a preset 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 and a vector. 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`). `IP_SIZE` is a constant which has to be defined in the graphic user interface `PrePALM`. 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 performance 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 producteurSUBROUTINE 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 graphic 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 “palmli” module (PALM library): 35. In the Fortran source code we have to do 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 send. 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.

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?

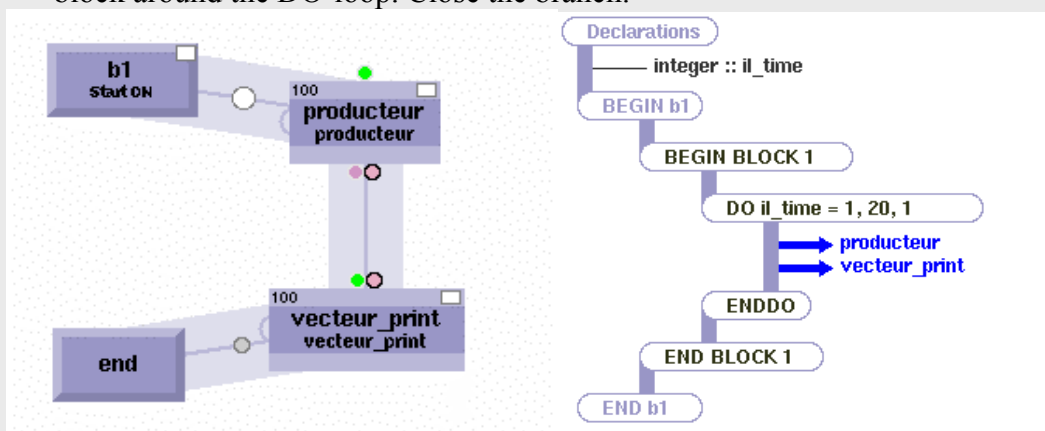
Complete the ID card and save the file!

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)
- Change the ID card of producteur.f90 and vecteur_print.f90
- Insert a branch b1 (IP_START_ON)
- Edit the branch code. Leave “producteur” and “vecteur_print” in this order
- Ad a DO-loop (1 to 20) with **il_time** as index, around the launch of the 2 units 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 management : | MEMORY |
| <input type="button" value="Cancel"/> <input type="button" value="Ok"/> | |

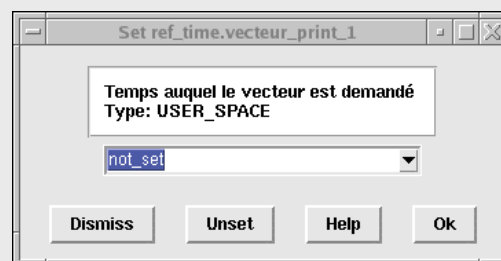
- In the field “Time list” instead of PL_NO_TIME, enter 1:20

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 ask for the vector at all of these times.

We don't have a unit producing object for a seta 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 graphic 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 25 30 | 20 25 30 |
| 4 404 | 4 | 404 |
| 18 118 ;1 1 :3 | 18 1 1 1 | 118 1 2 3 |
| 1 :3 :2 1 | 1 3 | 1 1 |
| 1 :3 4 :6 | 1 2 3 | 4 5 6 |
| 1 :3 4 :8 | Incorrect 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 branches **variables** are **prohibited**, 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.

Description of the field tag

If one of the two plugs has its **tag** field activated (-tag ON), the dialog window requires to enter the field : **tag list**. We 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**.

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 Hardwiring 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 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 (hardcode) the vector with the following Fortran90 expression

```
(/(i,i=1,IP_SIZE)/)*ib_do
```

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

Test

Note: 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 make it easier if you want to develop a complex application.

6 Session 6: Predefined units

6.1 Introduction

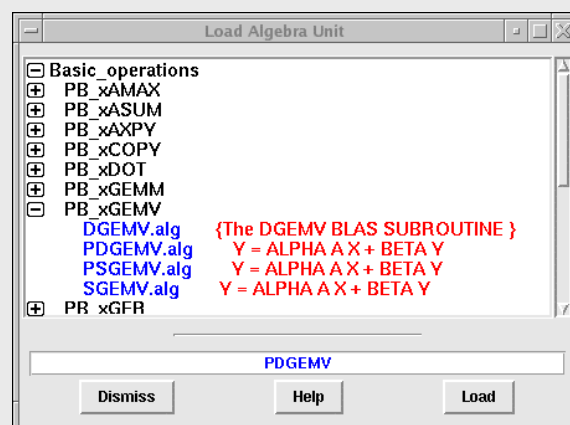
When you are building a coupled application, you may need to perform linear algebra operations on PALM objects before exchanging them between two units. For example a matrix vector product. These "generic" operations, from the simplest to the most complex, are directly available in the graphic user interface PrePALM. When they exist, you are even strongly advised 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 preset 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 producer unit (the same unit as in session 5)

Use the preset 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: File menu => Load Algebra unit

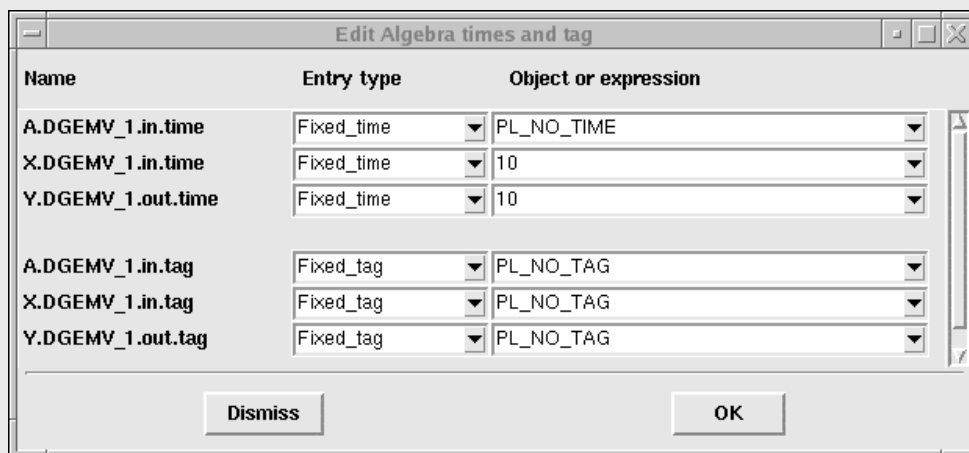


- Open Basic_operations => PB_xGEMV
- Selection DGEMV then Load

After the loading of 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 preset 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 communication PL_NO_TIME 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.
- Hardcode 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 requested at this same time in DGEMV.
- Set 10 for .DGEMV_1.in.time and Y.DGEMV_1.out.time
- Test your application.

Note:

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:

The simplest is to preserve the second branch as a START_ON branch, and to launch DGEMV in a block, 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.

7 Session 7: Derived data type objects

7.1 Introduction

Until now, the exchanged objects type was always a pre-defined 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 (=name)
  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, is better 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 have:

```

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.

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 key word “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 parameterised 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 be 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

Note:

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.

8 Session 8: Time interpolation

8.1 Introduction

When two computer codes are coupled, most of the time these programs do not have the same time step. In order to circumvent this problem we may interpolate in time the physical fields, which in turn forces us to store in memory several temporal instances of the same object. The time interpolation is readily offered by 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*. 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 producer (= “producteur”) 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!

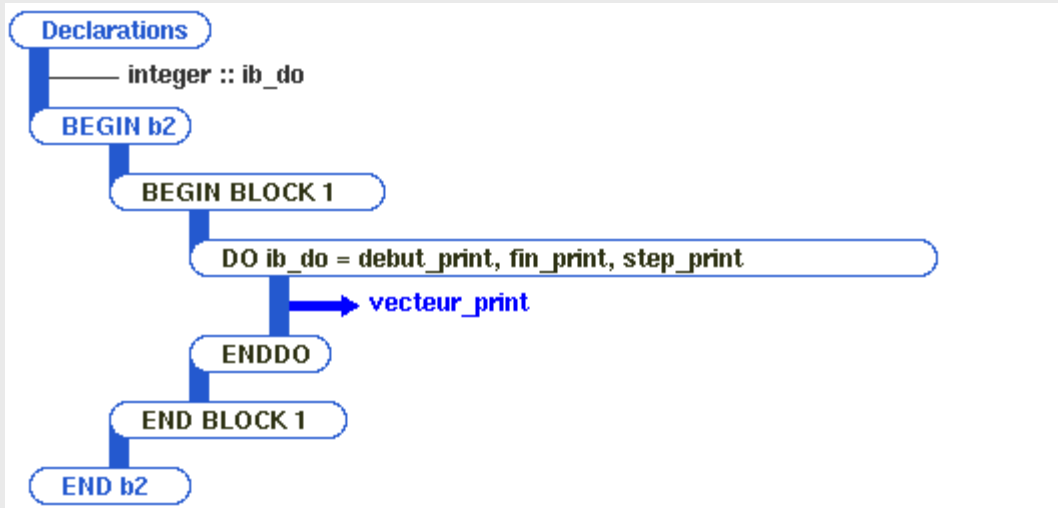
- Open a new PrePALM in the directory session_8
- Define four constants: `IP_SIZE = 100000` (vector size), `debut_prod = 0`, `fin_prod = 1000` and `step_prod = 10` (= start, end and frequency stride of the inner loop)
- Insert a branch b1 which launches the producer. For the producer input, in order 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

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:

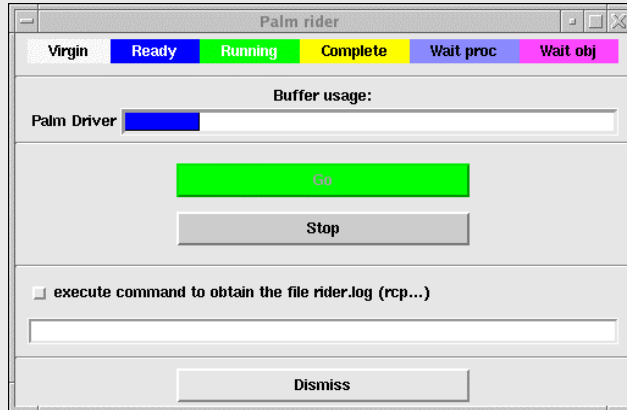


- Hardwire the loop index in the `ref_time` input plug of the `vecteur_print` unit
- Double-click on the vector input plug
- “time list” field: `debut_print:fin_print:step_print`
- Choose **PL_GET_LINEAR** for the interpolation field
- In the settings menu: palm execution settings: check all boxes
- Still in the settings menu: palm memory settings: set 100 for the **memory per slaves** field, this is equivalent to the maximum size of the PALM BUFFER.
- Test the application

8.3 Monitoring the application in real time

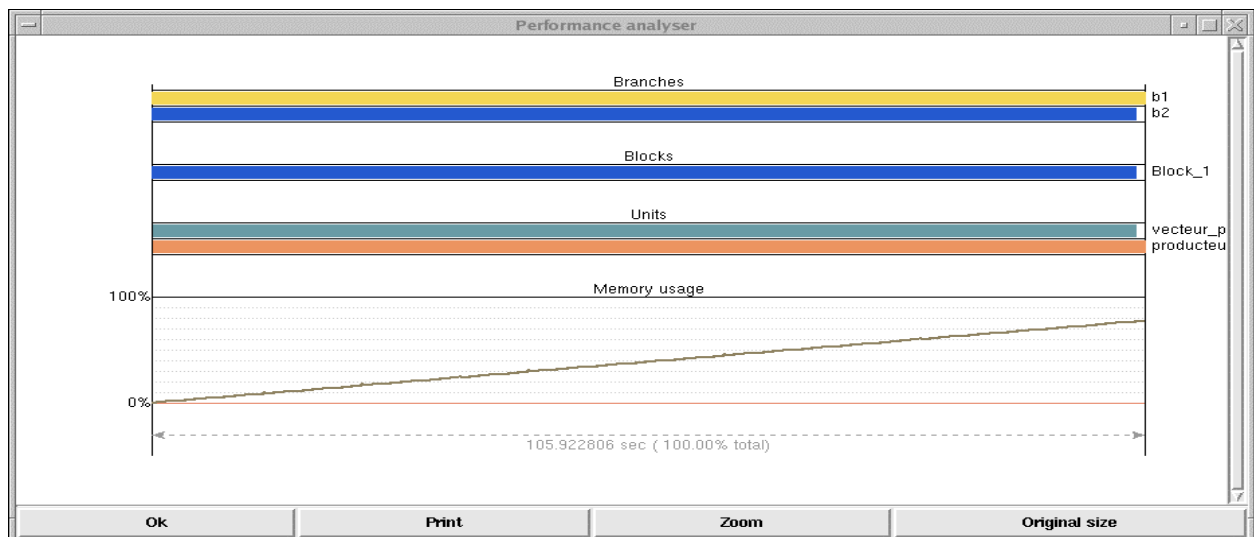
It is normal that the execution takes some time to run. In the loop producing the vector (unit producer) a `sleep(1)` call artificially slows down the execution. We need it to learn how to monitor in real time the execution of a coupled simulation.

Monitor the application!
Analyse run Menu => connect to run



Click on go!

In the PrePALM canvas, the units colour is green when they are running and yellow when they have finished. On the right of the units you may notice a red number which shows how much times each unit has already run. The Buffer usage progress bar allows us to see the size of the PALM driver memory really used. You notice that this size is increasing steadily 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 circumvent this problem for our application. But before that, check that in the file palm01_000.log you have 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 seen already 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 menu => Help on steplang grammar, you may find the syntax of the steplang language. You should read carefully this help .

Did you understand anything? 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 times 10 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).

For the memory usage optimisation, it would be nice to make the following action:

At all times after time 15, when a communication is made from the BUFFER to the vecteur_print, we have to delete from the BUFFER the objects "older" than the lower 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 translate easily relatively complex actions in order to manage the objects in the buffer.

Cleaning time!

- Step-actions menu => 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.

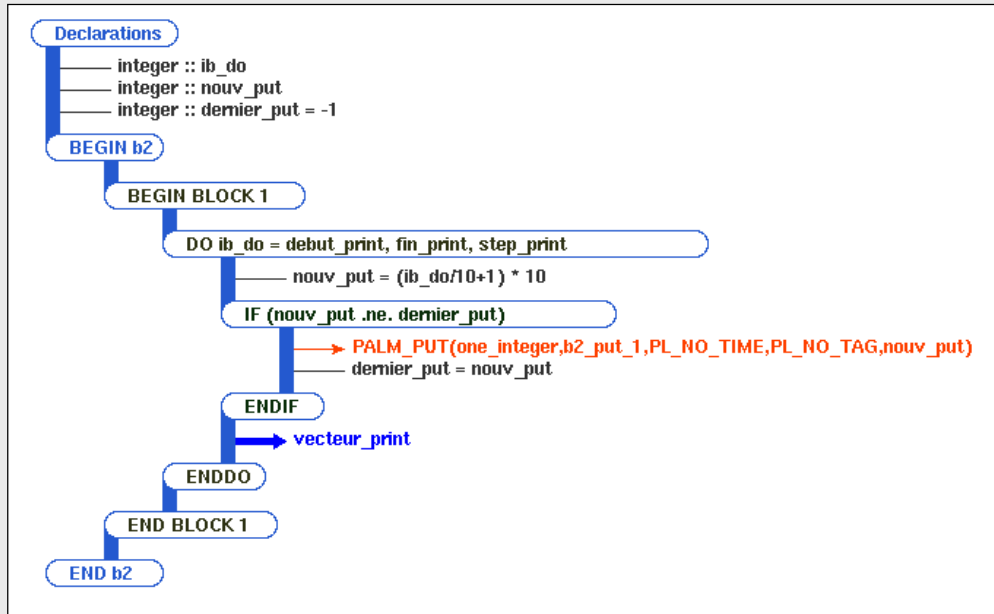
Slow down the producer!

- 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.

Synchronize!

- Change the vecteur_print branch as:



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

Now, the producteur unit is slowed down 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

When there is no more place in the PALM BUFFER, there is still another alternative. New processes can be associated with the PALM driver. These additional processes are devoted only to the management of the BUFFER memory. When these processes are launched on other processors, we may use their memory as an extension of the BUFFER, and thus we can go beyond the limits of the driver alone. Notice that this option of interest only on distributed memory machines. On shared memory machines, each processor can access the full memory, thus making the "memory slaves" useless.

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 help!

- 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 BUFFER?

What do we do on step1?

Is there an interest in making both units run in parallel?

What can we do to save a process?

Launch the application with one less process

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 parameterised via the constants of PrePALM, however it was static, i.e. it could not be changed during the run. 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 us see now how we can deal with this type of objects.

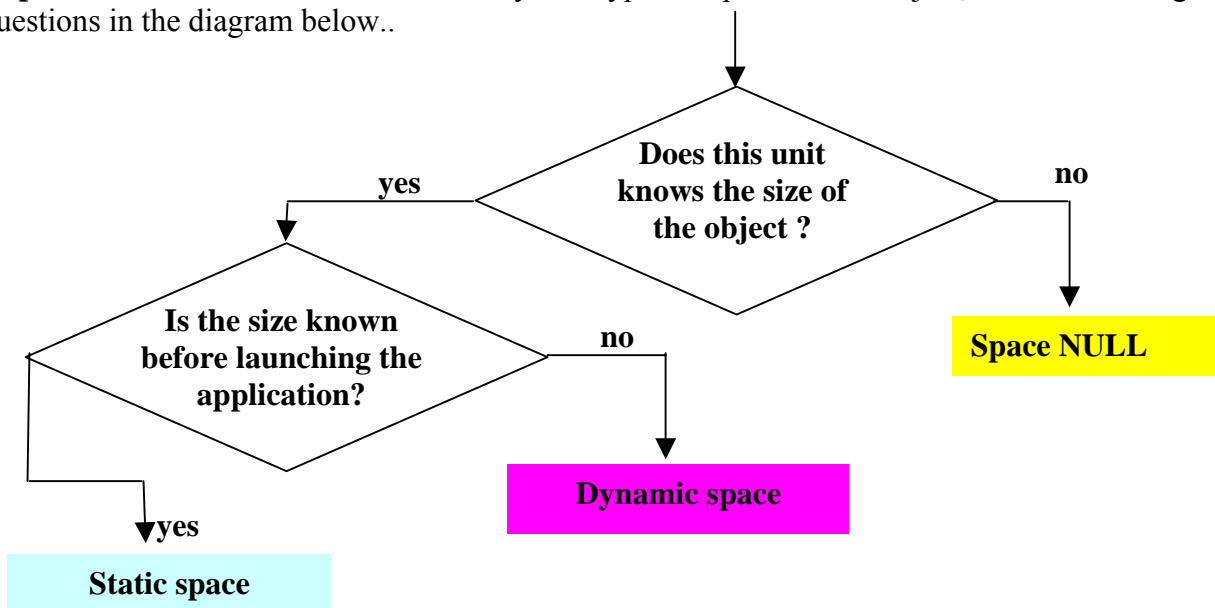
A first observation can be made: the size of a dynamic object must, of course, be known before the exchange, and this information can only be found in the unit which produces the object. Only the sending unit may know its current size (and certainly not the receiving unit, do you agree?). Thus, the source 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 can simply use the same space for the exchange. Since the units are supposed to be independent (spaces are private to each unit), a NULL space can be specified for the object in this case. Again, the graphic user interface PrePALM will be used to specify that the space of the receiving unit inherits the properties from the source object. This can be done quite simply by defining a new communication: NULL spaces inherit the characteristics from the received spaces.

In PrePALM the dynamic objects have a pink colour which differentiates them from the static objects. Objects with an indefinite space (NULL) have a yellow colour.

Moreover, you should notice that NULL spaces can as well 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 stay 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.

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



Exercise 12:

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 make the product of the matrix by the vector and will give 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, how many spaces (static, dynamic, NULL)?

How many objects for produit_mv, how many spaces (static, dynamic, NULL)?

How many objects for vecteur_print, how many spaces (static, dynamic, NULL)?

Edit the unit producteur.f90 of 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.

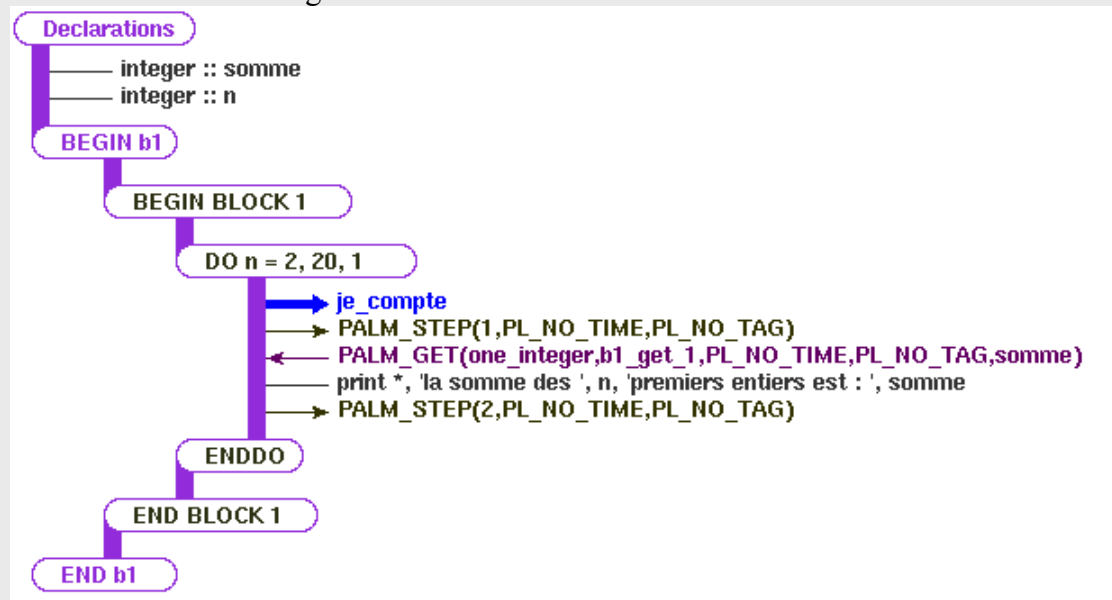
10 Session 10: Assembling objects in the BUFFER

We have seen the advantage of using the PALM BUFFER for keeping objects in a permanent memory and thus to be able to interpolate them or recover them repeatedly. Another role of the BUFFER is the object assembling, for example to calculate averages or sums of objects.

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 typed as input of `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 with 0. This will illustrate how to restore to some value (0) an object in the BUFFER, using `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 goes to the BUFFER with an object name “`somme`” (=sum). Check the field “`add`” of “`Palm algebra`” and put 1 and 1 for the coefficients.
 - The branch `PALM_Get` recovers 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 a script which sets the object state as “`ready`” on `step1` and which reset its value at 0 on `step2`.
- Test

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.

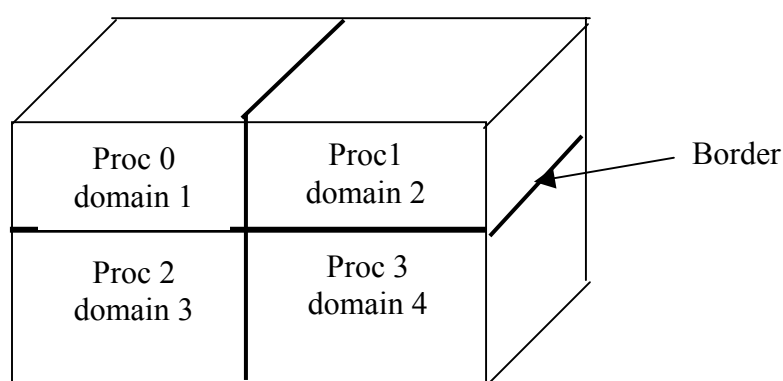
11 Session 11: Parallel communications

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 distribution. 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 treat a 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 handles in memory 3d arrays (corresponding to a physical domain discretization) which cover the whole Earth. The code is parallelized in order to shorten 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 physically it is distributed in the memory of several processors.



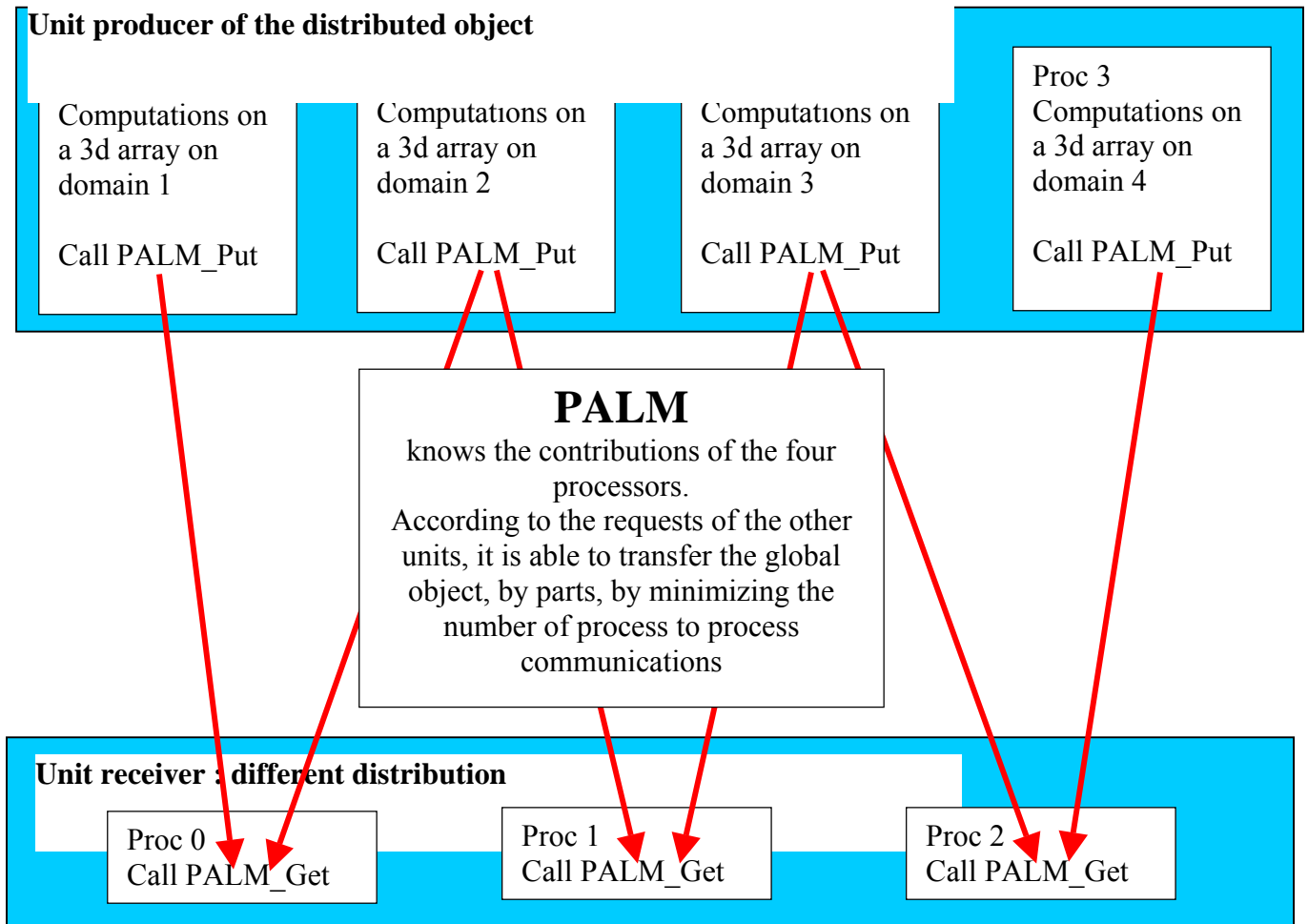
Example of distributing a 3d field on 4 processors

Each processor holds only one part of the 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.

Executable // MPI



PALM launching the unit on 4 proc.
the program is replicated on 4 processors running in parallel



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, or therefore how the code has been parallelized. The relevant information is: "what is the part of the global object managed by every single process". In PALM we have introduced a syntax for this

information. To grant enough flexibility, it is possible to describe the distributors in two different ways more or less in accordance with to the case to be treated

The distributors of type **regular** 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 type **custom** are less concise but they allow to describe any type of distribution.

The distributions are described in the units ID cards. In the same ID card, the distributed objects must specify which distribution they use.

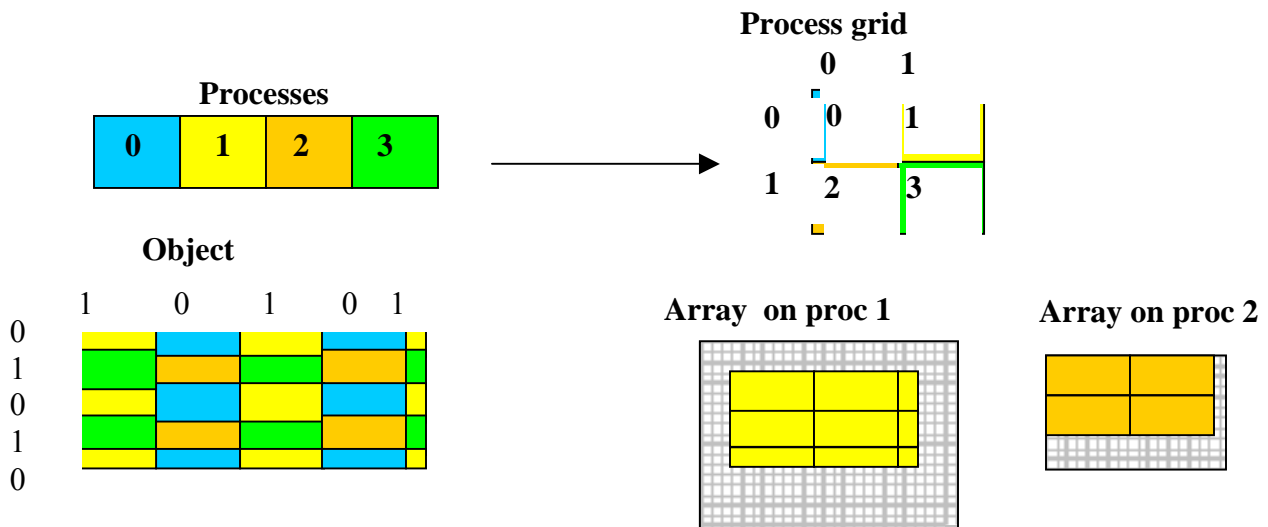
11.3 Block cyclic distributors

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 is arbitrary.

In our example, we chose (0,1), in yellow.

The blocks are stored in the local object in a contiguous way. The local object can be larger than the total size of the blocks.

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

1. The global object shape (i.e. its size in all dimensions): (2,2) in the previous example
2. The elementary block size: (30,40) for example
3. The coordinates in the process grid which will contain the first global object block: (0,1) in the previous example
4. For each process:
 1. the local object form
 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.

Exercise 13:

Distribute a global object of size 1000x1000 in 4 equals parts

Like this:

| | | | |
|----------|----------|----------|----------|
| 0 | 1 | 2 | 3 |
|----------|----------|----------|----------|

Then like that:

| |
|----------|
| 2 |
| 3 |
| 0 |
| 1 |

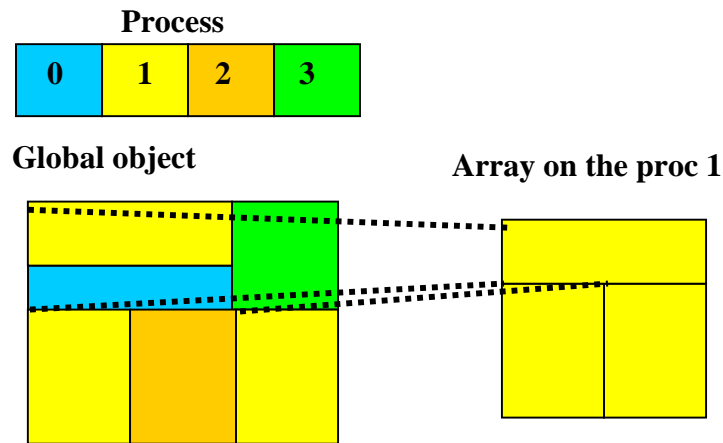
And then like this:

| | |
|----------|----------|
| 0 | 1 |
| 2 | 3 |

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 The distribution functions

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:

- Create regular distribution file : palm_reg_distr.f90
- Create regular with halo distribution file : palm_regwh_distr.f90
- Create custom distribution file : palm_cust_distr.f90

11.6 An example of a distributed object

In the directory session_11, 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 process 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 reduces to a copy/paste of the part which parallelizes the code (the definition of the domains decomposition) and to manage these data in such a way that PALM can understand it.

Distribute!

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

| | |
|------------------------------|------------------|
| <code>ip_nlon_ocean</code> | <code>182</code> |
| <code>ip_nlat_ocean</code> | <code>149</code> |
| <code>ip_nbproc_ocean</code> | <code>8</code> |

- 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`
- Test

Exercise 14:

Make the model work on just one time step, and modify the `toy_ocean` source code such 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 -1 for the refresh object of the unit `plot_tcl`, in order to keep the drawing on the screen. In this case, also add the character “&” (asynchronous 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 2_4_0 version, be dynamic: This feature is under development.

11.7 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 copied on all the processes. All of these characteristics must be communicated to PALM. This is done again via the ID cards and the graphic user interface.

If you look at the unit `toy_ocean` ID card, you will see that there are two fields “-localisation” in the object description. 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.

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 manage the local parts of the object described in the distributor.

An object is said to be replicated if several unit processes to which it belongs manage independently a copy of this object. In this case, the localisation describes the list of the unit processes which manage an instance of this object.

A PALM localisation is defined with the keyword `PALM_LOCALISATION`.

Three predefined localisations are available for the user:

- `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.

If the localisation of your object is not one of the pre-defined localisations, it will be necessary to define your own localisation. For example let us suppose that your object is distributed on the proc 0, 4, 3 and 2 whereas your unit executes on 5 processors, in this case you should use a localisation of the type:

```

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

```

Thus, the localisations are attached to the objects and are defined in the ID card.

Since the objects may also be replicated, it is necessary, at the time of the communication definition in PrePALM, to accurately describe the way in which the instances of the objects are transferred between the source and target units.

For 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 the proc 0 to the 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 rebuilt 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 |

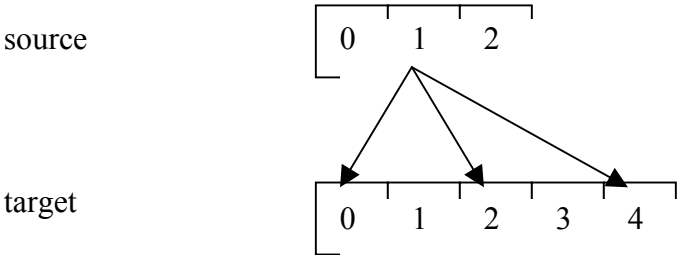
| | |
|--|---|
| | process with the same rank, on the target side. Assoc : 0 :nbproc_src-1 |
|--|---|

For the more complex cases, it is necessary to define the associations between the "base processors" of the instances of the objects on the source and target sides of the communication. This definition is made with the same syntax as the one used before to establish the correspondence of the time lists:

```
start1[:end1[:stp1]] [ start2[:end2[:stp2]]] [ ; ...]
```

Example:

Let us consider two units exchanging an object. This object has only one instance in the source unit which is localized on process 1 of this unit. In the target unit, three instances of the object are present, on processes 0, 2 and 4. We want to send the source object to each instance of the target object. The association of the instances for this communication is then **1|0:4:2**



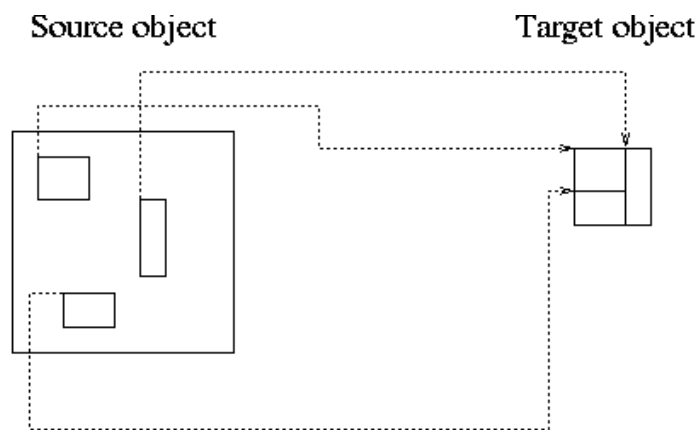
You should not worry; even if PALM offers this mechanism to be able to treat all kinds of parallel communications, it is quite unusual to have to describe a localisation which differs from the pre-defined ones.

12 Session 12: Sub-objects

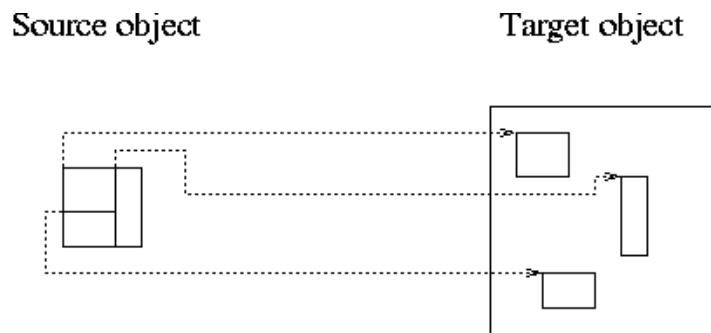
The sub-objects have been introduced in PALM_MP to grant an even higher level of independence between the units. It may be useful for example, to recover only a part of an object in a target unit without having to modify the code of the source unit. For this purpose we may use sub-object descriptors.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

This feature allows:

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

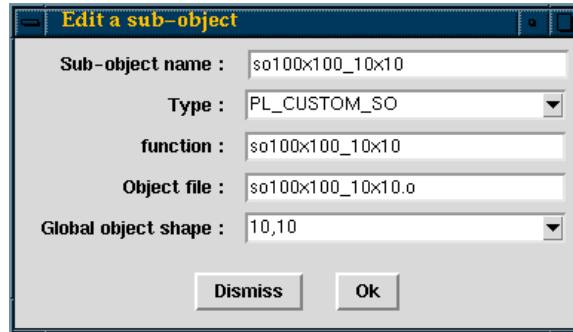


- to update in a target unit only part of an object by a PALM_Get,



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.

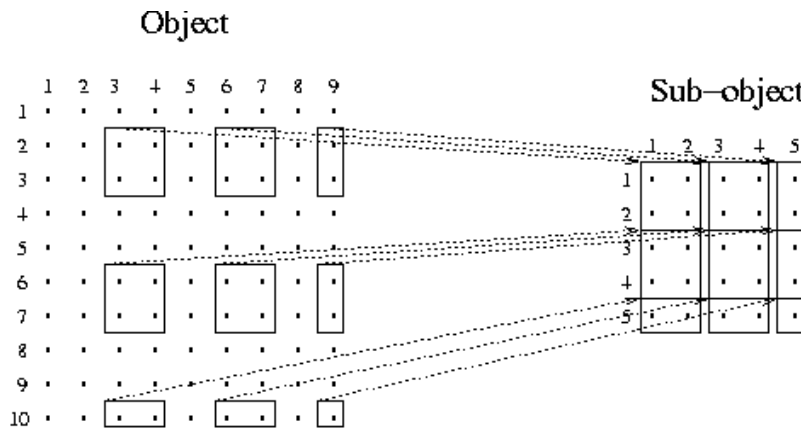
To define a sub-object, the user must check the box “**Sub-object descriptor**” in the Selector then click on the **Insert** button. A window pops up. The user must enter the name of the sub-object, its type, the name of the function which describes it, the name of the compiled file which contains the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



The different types of sub-objects correspond to the different models which describe them. A sub-object can have a type : **PL_CUSTOM_SO** or a 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 data 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:

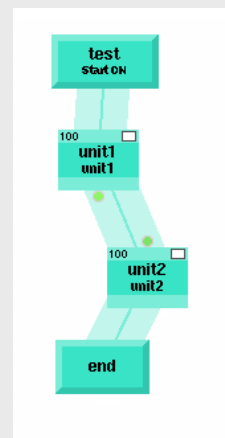
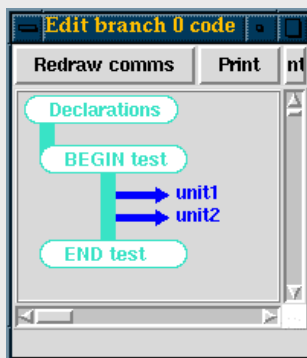
- the sub-object rank,
- its shape,
- the number of blocks of the sub-object,
- the description of each block (shape, coordinates of the top left corner in the global object and in the sub-object).

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 makes a PALM_Get of a 10x10 real array and print 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
- Create the following application:



- menu File => Make Palm files, check the case “create custom sub object file” and create the file “palm_cust_so.f90”
- Rename this file as: so100x100_10x10.f90
- 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:

The screenshot shows a dialog box titled "Insert a sub-object". It contains the following fields and values:

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

Buttons: Dismiss, Ok

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

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/write 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 session 11. At each time step, the model `toy_ocean` produces 2d fields on a relatively complex space discretization grid (orca grid from the famous 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 data compared to the spaces definition in units. Beside that, one finds the same information as in the ID cards of the PALM units.
- With PrePALM, open the application `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 made with a pre-defined unit that you can find directly in PrePALM (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.

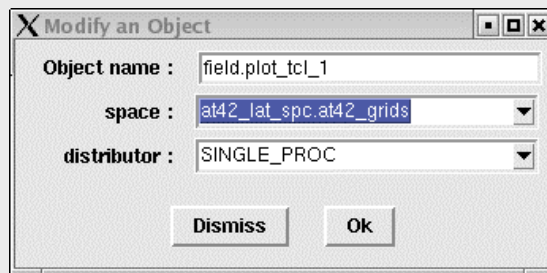
The interpolation unit works like this:

- 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

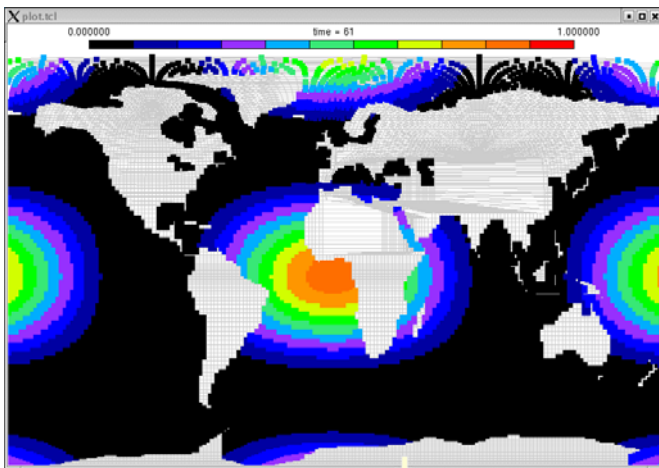
options given by the user. In the subsequent runs, the unit just reads the needed data from this file.

Interpolate!

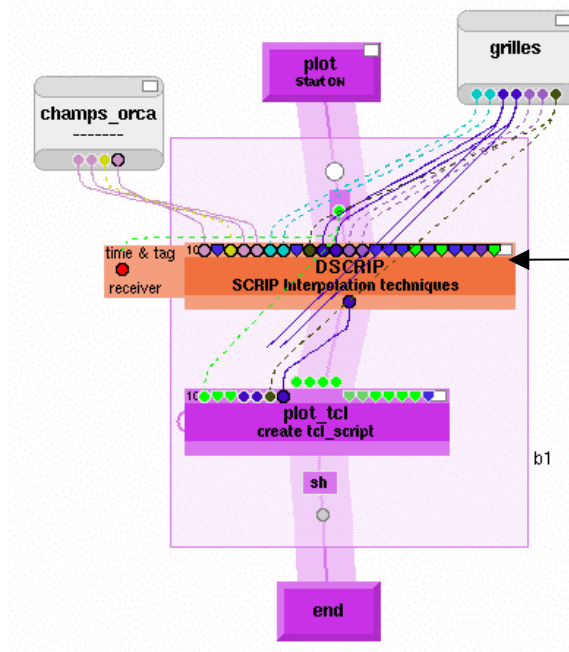
- In PrePALM open the file `session_13.ppl` which contains a branch calling the interpolation unit and the fields visualization utility. The file containing the fields on the orca grid is already connected in PrePALM. Notice that one will interpolate only a few time instances in this file (DO-loop).
- 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), give the right file name for 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.

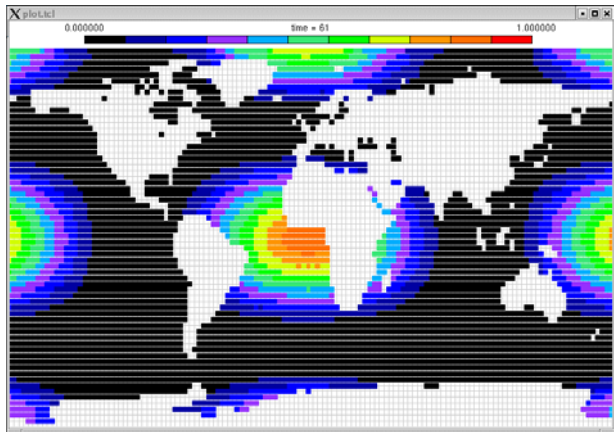


Fields calculated on the ocean grid (orca 2)



OASIS spatial interpolation

Interpolated field on the ARPEGE AT42 grid



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 an application process.

As an example, we will minimise a cost function of the form: $J(\mathbf{x}) = \mathbf{x}^T \mathbf{B}\mathbf{x}$. The gradient of this function is $\mathbf{grad} J(\mathbf{x}) = 2*\mathbf{B}\mathbf{x}$. We will use the CGPLUS minimiser implemented as 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} (the elements of the matrix \mathbf{B} are 1, 2, 3, ..., ip_vectsize) and it will return the $\mathbf{B}\mathbf{x}$ result. The third unit (“result”) will just print the result.

The gradient $2*\mathbf{B}\mathbf{x}$ can be easily calculated just by multiplying $\mathbf{B}\mathbf{x}$ by 2 with the algebra unit DSCAL. The expression $\mathbf{x}^T \mathbf{B}\mathbf{x}$ 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 calculated to make a new iteration. For each iteration, the minimiser raises a flag telling if the minimization process is 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.

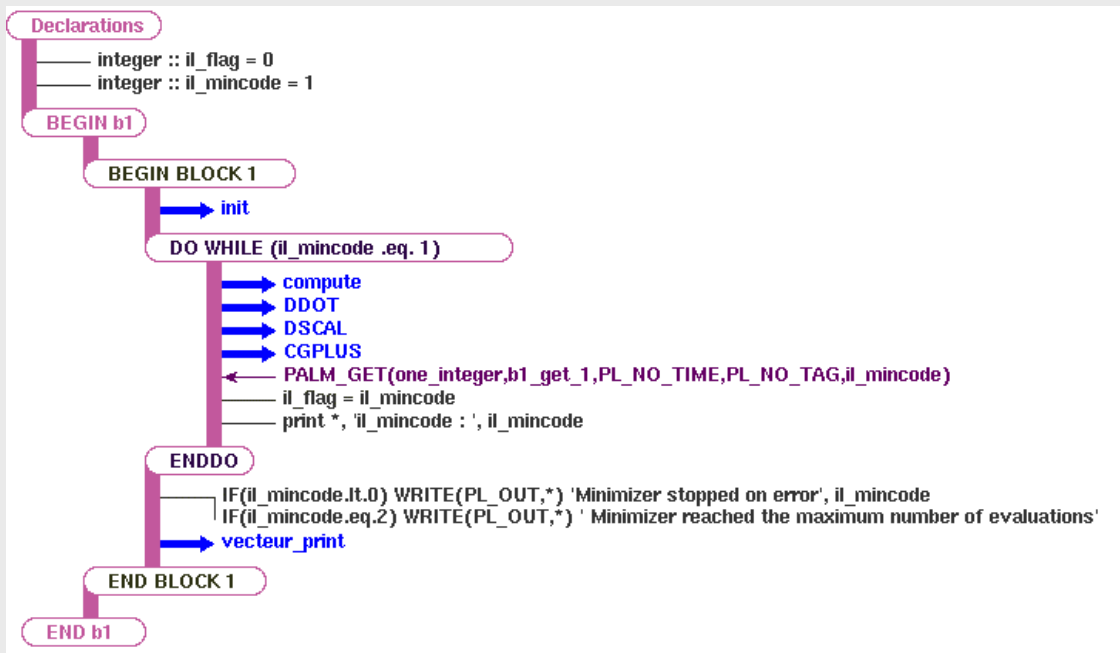
Minimize!

- 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 (“first_guess” of “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 |

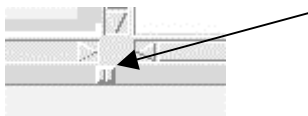
- From the CGPLUS flag to the PALM_Get branch
- For the other plugs, hardwire the following values:
- 2.0 for ALPHA and DSCAL
 - 1 and 0 for iprint1 and iprint2 of CGPLUS
 - 1.d-12 for eps of CGPLUS
 - il_flag for iflag of CGPLUS
 - 0 for irst, 2 for method
 - ip_iter for nbmaxiter
 - ip_eval for nbmaxeval
 - .false. for finish
- Test

Let's take advantage from the 11 communications we have (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 graphic user interface.

Select the communication category of PrePALM:

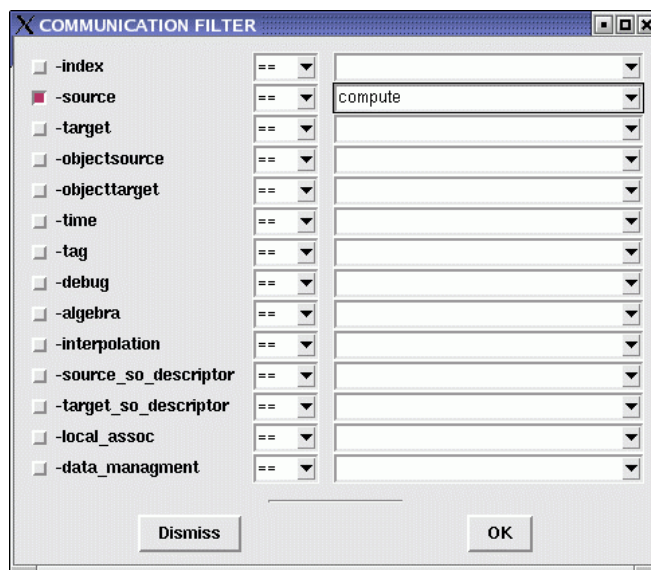


Make the window "attributes" larger with the slider:



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

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 "computed". 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.

You can finally customize the graphic look of the canvas in the menu settings => canvas settings: plugs size, width of the branches...etc, etc.

{ TC "Palm Glossary" \1 1}Palm Glossary

Action: steplang language instruction to be executed on an event; for example: destruction of an object, or definition of an attribute for this object.

Algebra: pre-defined unit for algebraic algorithms. The algebraic operations (linear combinations, linear systems solving, eigenvalues and eigenvectors computing, minimisers, ...) can be treated as special units belonging to branches and available from PrePALM.

Application: a PALM application is the collection of elementary units plus the main driver in order 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.

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

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.

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.

Category: used in Prepalm: choice of a keyword defining the verbosity of the outputs. Each keyword refers to a set of attributes of the elements

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.

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 with an include file in the source code (language dependent).

Control Structures: control structures like loops or conditions can be used in the construction of the branches in PrePALM.

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 uses the lamd daemon, started by the lamboot command.

Derived Data Type: composite data type defined with elementary data types.

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.

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.

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 a dynamic launch of the units or blocks or other branches, and 2) answer the requests from the executable files for all communications.

Dynamic Object: an object having a dynamic space.

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.

Event: an event is linked to a step: it is the point where an action can be executed in the PALM Buffer. All communications are events like any other STEP defined by the user.

FORTRAN Region: in the PrePALM graphic interface, the FORTRAN regions allow to execute FORTRAN instructions in the application branches. The source code of these regions must be FORTRAN 90.

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 graphic 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).

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

Inheritance: in the PrePALM graphic 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.

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

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

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. This file is automatically generated by the PrePALM graphic 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)

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.

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 subroutine or PALM function: any of the subroutines included in the PALM library, which can be called by the user.

"Palmer": (to palm, or palming): action to modify a computing code in order to use the PALM coupler. In French you can also say: "palmériser". The "palmipèdes" (PALM users' community) prefer "palmer".

Parallel Communication: communication involving at least one distributed object.

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.

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

PrePALM: the PALM graphic 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 hardwired 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 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).

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 deducted from the localisation of the objects.

Processor: electronic component executing the computing code

Replay: function of the graphic interface allowing a visualisation of the sequential execution of all application components.

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

Run Time Animation: function of the graphic interface allowing seeing in real time which unit, block or branch is currently executed.

Script: file containing system commands (written in a shell language like csh, ksh, etc...).

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

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 **Single Program Multiple Data**.

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.

Steplang: programming language, interpreted by the PALM driver, describing the actions to be executed on the BUFFER's objects.

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.

Time: 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".

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.

Verbosity: level to be defined for the number of PALM messages in the output files.

Sub-Object: PALM feature allowing to handle only a part of an object.

{ TC "List of PALM functions"\1 1}List of PALM functions

Programming languages:

Call from C/C++:

```
int il_err ;  
il_err = PALM_Exemple(arg1,arg2,...) ;
```

Call from FORTRAN:

```
integer il_err  
CALL PALM_Exemple (arg1, arg2, ..., il_err)
```

Application control:

```
int PALM_Abort()
```

```
int PALM_Error_explain(int *err_code_to_explain)
```

Communications:

```
int PALM_Put(char *space, char *obj, int *time, int *tag, void *data)
```

```
int PALM_Get(char *space, char *obj, int *time, int *tag, void *data)
```

```
int PALM_Query_get(char *space, char *obj, int *time, int *tag)
```

```
int PALM_Query_put(char *space, char *obj, int *time, int *tag)
```

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_off()

int PALM_Verblevel_overall_get(int *level)

int PALM_Verblevel_overall_set(int *level)

With:

level: integer or PL_VERBLVL_NOHING, PL_VERBLVL_WARNING, PL_VERBLVL_USRLEVEL.

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) ;

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)

{ TC "Identity Cards"\1 1}Identity Cards

!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 |
| -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 graphic 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.