New features for advanced dynamic parallel communication routines in OpenPALM: Algorithms and documentation

Martin Wlotzka, Thierry Morel, Andrea Piacentini, Vincent Heuveline

Preprint No. 2017-04

$\frac{\partial u}{\partial t} - \nu\Delta u + (u\cdot\nabla)u + \frac{1}{\rho}\nabla p$
$\frac{\partial u}{\partial t}$

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
Preprint No. 2017-04

Affiliation of the Authors

Martin Wlotzka[a,1], Thierry Morel[b], Andrea Piacentini[b] Vincent Heuveline[a]

[a] *Engineering Mathematics and Computing Lab (EMCL), Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany*
[b] *Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique (CERFACS), Toulouse, France*
[1] *Corresponding Author: Martin Wlotzka, martin.wlotzka@uni-heidelberg.de*

Impressum

Heidelberg University
Interdisciplinary Center for Scientific Computing (IWR)
Engineering Mathematics and Computing Lab (EMCL)

Speyerer Str. 6,
69115 Heidelberg
Germany

New features for advanced dynamic parallel communication routines in OpenPALM: Algorithms and documentation

Martin Wlotzka, Thierry Morel, Andrea Piacentini, Vincent Heuveline

June 13, 2017

Abstract

We present an algorithmic sketch and documentation of the new dynamic distributors feature in the parallel communication routines of the OpenPALM software coupler tool. OpenPALM controls the execution of coupled applications and provides communication routines for data exchange between the units. A typical use case of OpenPALM is the coupling of numerical models for data assimilation or multiphysics simulations. The communication routines allow to transfer data between units exhibiting internal parallelization and data distribution. OpenPALM is able to match the distributed object parts between parallel units by means of its internal routing table. However, in the legacy OpenPALM version 4.1.4, the parallel configuration of the units and their data distribution needs to be determined through offline tests before the coupling application can run. Moreover, the data distribution cannot change anymore during runtime of the application. This may result in substantial overhead for the setup of applications, and it prevents from common practices in numerical models which may change data sizes and distributions during runtime like coarsening, refinement or load balancing. Our new developments allow to determine and change the data distribution among coupled units at any time during the execution of the coupling application, while at the same time keeping OpenPALM's internal routing table consistent. This greatly eases the setup of applications and supports coarsening, refinement and load balancing during runtime.

1 Introduction

OpenPALM [1, 2, 5] is a software coupler tool with advanced features like dynamic and concurrent execution models, the ability to couple parallel codes, and a flexible communication scheme. It is a general purpose coupling tool, although its main focus lies on scientific computing and numerical simulation. The fundamental concept of OpenPALM is to consider applications as a composition of units which can be coupled by means of a data transfer mechanism. One may in principal regard anything which can be implemented and executed as a computer program as a unit in terms of OpenPALM. Units may represent a vast variety of computational tasks. Typical examples include reading or writing files, performing algebraic operations, solving systems of equations, up to large applications such as complete climate codes or ocean models. OpenPALM's goal is to ease the coupling of new and of existing codes written in Fortran, C or C++ with minimal effort, even if they were not meant to be coupled in the first place. However, OpenPALM's communication mechanism is restricted to the case where data sizes and data distributions among the components are known a priori. In the following paragraphs, we briefly recapitulate the basic terms and concepts of OpenPALM. Section 2 presents an algorithmic sketch of the communication mechanism of the legacy OpenPALM version 4.1.4. Section 3 presents the main result of this work, an algorithmic sketch of the advanced communication mechanism including our developments towards the new

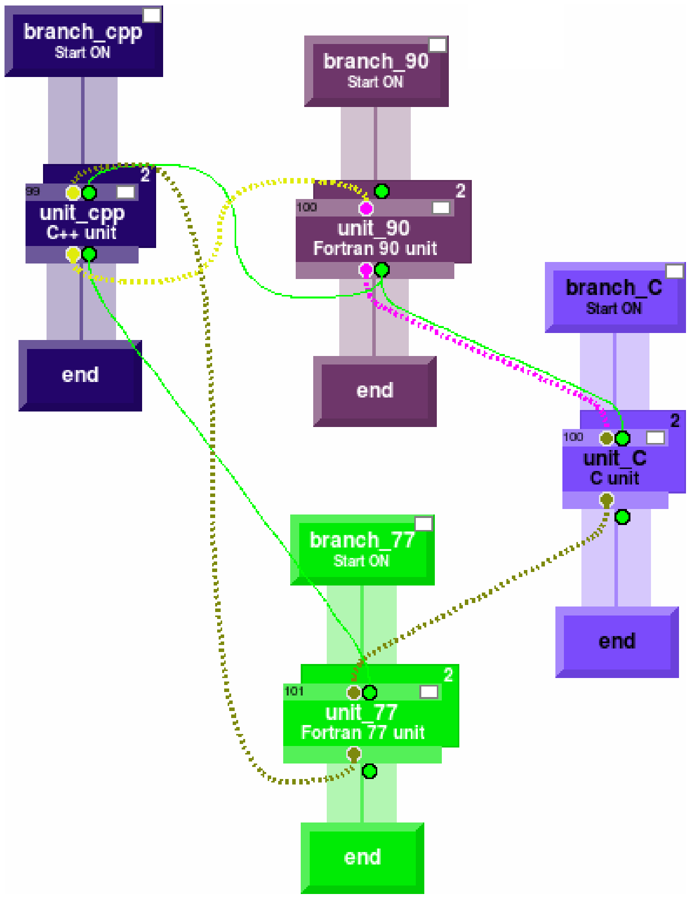

Figure 1: The canvas of the PrePALM graphical user interface. The coupling algorithm in this example has four independent execution branches, depicted as the vertical bars. Each of the branches has one unit, represented by the boxes in the middle, scheduled to it. Data transfer is defined through the connections between the units.

features of dynamic distributors and dynamic sub-objects. The presented features have been implemented in the current OpenPALM version 4.2.3, which is available as open source[1].

1.1 Outline of OpenPALM terms and concepts

OpenPALM consists of three main components: a graphical user interface (GUI) named *PrePALM*, the *driver* and the *library*. The user can compose a coupled application in a pre-processing step with the help of the PrePALM graphical user interface. Its main feature is a canvas where the user can describe the coupling algorithm in a graphical form. This is done by defining execution paths, named *branches* in OpenPALM, scheduling the units by arranging them on the branches, and by connecting the units to indicate data transfer. Figure 1 shows an example of four independent execution branches, with one unit scheduled to each of them.

OpenPALM allows dynamic control flows in the coupling algorithm. This includes the conditional execution of units where it is not known a priori if and when conditions are fulfilled, repeated execution of units in loops where it is not known a priori if and how often the loop will be executed, or execution switches with multiple alternative paths. Complex control flows can be defined using an arbitrary number of branches.

OpenPALM features two levels of parallelism. On the one hand, units can run concurrently on separate

[1] `http://www.cerfacs.fr/globc/PALM_WEB/user.html` retrieved on June 13, 2017

sets of processors when they are scheduled to separate execution branches. On the other hand, OpenPALM is able to couple units which are internally parallelized supporting both shared and distributed memory parallel units, which may internally use message passing, multi-threading and/or accelerators.
A key feature is the communication mechanism. Owing to OpenPALM's philosophy of coupling individual units, it is necessary to facilitate data transfer between units and at the same time keeping them general and independent from any particular application. Models are viewed as entities which produce and/or consume data and perform certain computational tasks, so that generality and reusability for any purpose is maintained. Therefore, units cannot know about communication partners. Instead, they need a way to request for input data or to announce the availability of output data without information about source or target of the communication. OpenPALM offers communication routines for sending and receiving data, which fulfill these requirements. These routines, among others, are implemented in the *library*. Developers can use them in the unit's source code and link against the library. The communication routines are independent from the specific application at hand by using an abstract description of the data to be exchanged.
These library routines used in the units are complemented by the OpenPALM *driver*. The driver is a special entity which is automatically adjoined to any coupled application. It has two main purposes: to orchestrate the execution of the branches and units, and to act as a broker for the data transfer between the units. The driver starts, stops, and monitors the execution of the branches, and controls the units' access to resources such as files, memory, or processors. It also forms the counterpart to the communication routines used in the units. Since units do in general not know their communication partners, they announce data transfer requests to the driver. The driver then deduces the correct matching of source and target, and arranges a connection between the corresponding units.
For the algorithmic sketch of the communication mechanism and our new developments, we briefly introduce some OpenPALM terminology. We refer to the OpenPALM documentation and user guide [4] for further details. An *object* identifies the data which is transferred between units. A *space* is the definition of an object's data type. There are predefined spaces for the usual character, integer and floating point types, and arrays of them. One can also define custom spaces of composed types. A *distributor* describes the distribution of an object in a parallel unit. Distributors are complemented with *localizations*, which are ordered tupels of process ranks indicating which processes of a parallel unit take part in a distribution. A *sub-object* identifies a sub-set of an object. This can be used to extract specific parts if not the whole object is relevant for a communication. A *dynamic space* is a space who's size can be changed during runtime. In the legacy OpenPALM version 4.1.4, dynamic spaces can only be used for non-distributed objects. The goal of our developments is to enable dynamic spaces also for distributed objects, and to enable dynamic distributors and sub-objects.

The next section outlines the technical details of the legacy `PALM_Put` and `PALM_Get` routines by means of algorithmic sketches. These routines are the main facility to exchange data between coupled units. They trigger driver reactions to arrange the communication routes.

2 Internal data transfer mechanism in the legacy OpenPALM version 4.1.4

In the following, we present the legacy data transfer mechanism as it was implemented in OpenPALM version 4.1.4. We state the legacy `PALM_Put` and `PALM_Get` routines in Algs. 1 and 2, respectively, and the legacy implementation of the driver's reactions to these communication routines in Algs. 3 to 6. Some steps of the algorithms are marked with an asterisk indicating that an explanatory comment is given below.

Algorithm 1 Legacy `PALM_Put`

```
1:  Input: space name, object name, time, tag, pointer to local memory where the local object is
    stored.
2:  Determine the comids which are relevant for this call to PALM_Put.(*)
3:  Set $flag = 0$.
4:  for each comid do
5:      if this is a non-optimized communication then
6:          Check local mail buffer flag.(*)
7:          if $flag == 0$ then
8:              Notify the PALM_Put to the driver.(*)
9:              Receive the driver's answer.(*)
10:             Set $flag = 1$.(*)
11:         end if
12:         Determine rank and current shape of the space.
13:         if the space is dynamic then
14:             Check that the distributor is SINGLE_ON_FIRST_PROC, otherwise abort.(*)
15:             Compute the DOR according to the distributor intersection from the source side.(*)
16:         end if
17:         Allocate temporary memory for the DOR parts and disassemble the local object.
18:         Determine which target processes shall receive a part of the disassembled local object
            from this source process.
19:         for each process of the target's distributor do
20:             if that target process shall receive a part then
21:                 if the object shall be written to a file then
22:                     Write the object part to the file.
23:                 else
24:                     Send a connection request to the driver.(*)
25:                     Receive a connection authorization from the driver.(*)
26:                     Extract the communication type, the receiver entity type, and the receiver
                        MPI process rank from the drivers answer.
27:                     if the receiver is a unit or a memory slave then
28:                         Create an MPI intercommunicator to the receiver process.
29:                     end if
30:                     Send the object part to the receiver process.(*)
31:                     if the receiver is a unit or a memory slave then
32:                         Delete the intercommunicator to the receiver process.
33:                     end if
34:                 end if
35:             end if
36:             Notify the completion of the transfer of this object part to the driver.
37:         end for
```

```
38:         if the space is dynamic then
39:             Delete the DOR which was just created in step 15.
40:         else
41:             Deallocate the temporary memory for the DOR parts.
42:         end if
43:     else this is an optimized communication
44:         Get the MPI intracommunicator for this object.
45:         Determine rank and current shape of the space.
46:         if the space is dynamic then
47:             Check that the source distributor is SINGLE_ON_FIRST_PROC, otherwise abort.(*)
48:             Compute the DOR according to the distributor intersection from the source side.(*)
49:         end if
50:         Allocate temporary memory for the DOR parts and disassemble the local object.
51:         for each process of the target distributor do
52:             if that target process shall receive a part then
53:                 Create an MPI intercommunicator to the receiver process.
54:                 Send the object part to the receiver process.(*)
55:                 Delete the intercommunicator to the receiver process.
56:                 Call MPI_Barrier on the local process group of this object's MPI intra-
                    communicator.(*)
57:             end if
58:         end for
59:         if the space is dynamic then
60:             Delete the DOR which was just created in step 48.
61:         else
62:             Deallocate the temporary memory for the DOR parts.
63:         end if
64:     end if
65: end for
```

Explanatory comments on Alg. 1:

Step 2: Note that the calling EOS might be the source in several tubes. For each affected tube, the time-tag and localization matching is done, and the corresponding comids are determined. The rank of the calling process with respect to the source distributor is derived from the localization.
Step 6: Units which are integrated into the same block can use local mail buffer memory to exchange data instead of using the driver's mail buffer.
Step 8: The driver's reaction to this notification is stated in Alg. 3.
Step 9: The driver's answer is given in step 15 of Alg. 3.
Step 10: The flag is used to ensure that the driver is notified only once, even if there are multiple comids. The driver can itself determine all relevant comids.
Steps 14 and 47: This is the reason why the legacy PALM_Put algorithm can only handle dynamic spaces for non-distributed objects.
Steps 15 and 48: Since dynamic spaces are only possible with non-distributed objects in the legacy PALM_Put, the resulting DOR just represents the new space shape.
Step 24: The driver's reaction to this connection request is stated in Alg. 4.
Step 25: The driver's answer is given in steps 3, 8 or 24 of Alg. 4.
Step 30: The object part is received in step 26 of Alg. 2 in case of a direct communication, or in step

18 or 20 of Alg. 4 in case of an indirect communication, or in step 12 or 14 of Alg. 4 in case of a buffer communication.
Step 54: The object part is received in step 49 of Alg. 2.
Step 56: The MPI barrier is called on the local process group, i.e. only on the source side. This is necessary to avoid race conditions where several source processes could send object parts to the same target process at the same time.

Algorithm 2 Legacy `PALM_Get`

```
1: Input: space name, object name, time, tag, pointer to local memory where the local object
   shall be stored after reception.
2: Check if there is a comid for this call to PALM_Get.
3: if there is a comid then
4:     if this is a non-optimized communication then
5:         Notify the PALM_Get to the driver.(*)
6:         Receive the driver's answer.(*)
7:         Check local mail buffer flag.(*)
8:         Determine rank and current shape of the space.
9:         if the space is dynamic then
10:            Check that the target distributor is SINGLE_ON_FIRST_PROC, otherwise abort.(*)
11:            Compute the DOR according to the distributor intersection from the target side.(*)
12:        end if
13:        Get the number of source object parts for this target process.
14:        if the object shall be read from a file then
15:            Read local object parts from file.
16:        end if
17:        Receive information from the driver where the object parts are located.(*)
18:        Allocate temporary memory for the DOR parts.
19:        for each source process do
20:            if that source process contributes an object part then
21:                Send a connection request to the driver.(*)
22:                Check if an MPI intercommunicator is needed.(*)
23:                if an MPI intercommunicator is needed then
24:                    Create an MPI intercommunicator to the sender process.
25:                end if
26:                Receive the object part from the sender process.(*)
27:                if an MPI intercommunicator was need then
28:                    Delete the MPI intercommunicator to the sender process.
29:                end if
30:            end if
31:        end for
32:        Assemble the DOR parts to build the local object.
```

```
33:         if the space is dynamic then
34:             Delete the DOR which was just created in step 11.
35:         else
36:             Deallocate the temporary memory for the DOR parts.
37:         end if
38:     else this is an optimized communication
39:         Determine rank and current shape of the space.
40:         if the space is dynamic then
41:             Check that the target distributor is SINGLE_ON_FIRST_PROC, otherwise abort.(*)
42:             Compute the DOR according to the distributor intersection from the target side.(*)
43:         end if
44:         Get the number of source DOR parts for this target process.
45:         Allocate temporary memory for the DOR parts.
46:         for each source process do
47:             if that source process contributes a DOR part then
48:                 Create an MPI intercommunicator to the sender process.
49:                 Receive the object part from the sender process.(*)
50:                 Delete the MPI intercommunicator to the sender process.
51:             end if
52:         end for
53:         if the space is dynamic then
54:             Delete the DOR which was just created in step 42.
55:         else
56:             Deallocate the temporary memory for the DOR parts.
57:         end if
58:     end if
59: end if
```

Explanatory comments on Alg. 2:

Step 5: The driver's reaction to this notification is stated in Alg. 5.
Step 6: The driver's answer is given in steps 13 or 22 of Alg. 5.
Step 7: Units which are integrated into the same block can use local mail buffer memory to exchange data instead of using the driver's mail buffer.
Steps 10 and 41: This is the reason why the legacy `PALM_Get` algorithm can only handle dynamic spaces for non-distributed objects.
Steps 11 and 42: Since dynamic spaces are only possible with non-distributed objects in the legacy `PALM_Get`, the resulting DOR just represents the new space shape.
Step 17: The object part locations are sent by the driver in step 12 of Alg. 3 in case of a direct communication, or in step 23 of Alg. 5 in case of an indirect or buffer communication.
Step 21: The driver's reaction to this connection request is stated in Alg. 6.
Step 22: An MPI intercommunicator needs to be created if the sender is not the driver, to which an intercommunicator exists anyway, and when the local mail buffer is not used.
Step 26: The object part is sent in step 30 of Alg. 1 in case of a direct communication, or in step 10 or 12 of Alg. 6 in case of an indirect communication, or in step 3 or 5 of Alg. 6 in case of a buffer communication.
Steps 49: The object part is sent in step 54 of Alg. 1.

Algorithm 3 Driver reaction on the `PALM_Put` notification in step 8 of Alg. 1

```
1: Determine the comids which are relevant for this call to PALM_Put.
2: for each comid do
3:     Insert this PALM_Put into the commstate table.
4:     Search the commstate table for a waiting PALM_Get matching this comid.
5:     if there is a waiting PALM_Get then
6:         Update the commstate table that the PALM_Get will be served by this PALM_Put.
7:         if the source is the buffer then
8:             Determine if the object is located at the driver or a memory slave.
9:         end if
10:        for each target process do
11:            Send answer to target process.(*)
12:            Send object part locations to the target process.(*)
13:        end for
14:    end if
15:    Send answer to the source process.(*)
16: end for
```

Explanatory comments on Alg. 3:

Step 11: This answer is received in step 6 of Alg. 2.
Step 12: The object part locations are received in step 17 of Alg. 2.
Step 15: This answer is received in step 9 of Alg. 1.

Algorithm 4 Driver reaction on the `PALM_Put` connection request in step 24 of Alg. 1

```
1: Query the commstate table if this PALM_Put matches a waiting PALM_Get.
2: if a PALM_Get is waiting for the connection then
3:     Send a connection authorization to the source process telling that the communication
       type is direct, the target is a unit, and the MPI rank of the receiver process.(*)
4: else the target is the buffer, or the object must be temporarily stored in the mailbuf
5:     Determine if the target is the buffer, or if the mailbuf shall take the object part.
6:     Determine if the driver or a memory slave shall receive the object.
7:     if the driver receives the object part then
8:         Send a connection authorization to the source process telling whether the communi-
           cation type is buffer or indirect, the receiver is the driver, and the MPI rank of the
           driver.(*)
9:     end if
10:    if the target is the buffer then
11:        if the driver receives the object part then
12:            Receive the object part from the source process and put it in the buffer.(*)
13:        else
14:            Order a memory slave to receive the object part and to put it in the buffer.(*)
15:        end if
```

```
16:     else the target is not the buffer
17:         if the driver receives the object part then
18:             Receive the object part from the source process and put it in the mailbuf.(*)
19:         else
20:             Order a memory slave to receive the object part and to put it in the mailbuf.(*)
21:         end if
22:     end if
23:     if a memory slave receives the object then
24:         Send a connection authorization to the source process telling whether the communi-
            cation type is buffer or indirect, the receiver is a memory slave, and the MPI rank of
            the memory slave.(*)
25:     end if
26: end if
```

Explanatory comments on Alg. 4:
Steps 3, 8 and 24: This answer is received in step 25 of Alg. 1.
Steps 12, 14, 18 and 20: The object part is sent in step 30 of Alg. 1.

Algorithm 5 Driver reaction on the `PALM_Get` notification in step 5 of Alg. 2

```
 1: Determine the comids which are relevant for this call to PALM_Get.
 2: Query the commstate table to choose a comid.
 3: if this PALM_Get does not read from a file then
 4:     if not all processes of the target entity have announced this PALM_Get yet then
 5:         Return.(*)
 6:     end if
 7:     if there is neither a matching PALM_Put currently active nor is the object available in the
        mailbuf or buffer then
 8:         Memorize the PALM_Get to be wakened later in step 5 of Alg. 3 when a matching
            PALM_Put occurs.
 9:         Return.(*)
10:     end if
11: else the object is read from a file
12:     for each target process do
13:         Send a dummy answer to the target process.(*)
14:     end for
15: end if
16: for each target process do
17:     for each source process do
18:         if the object part is in the mailbuf then
19:             Determine whether the object part is stored on the driver or a memory slave.
20:         end if
21:     end for
```

```
22:     Send an answer to the target process telling the chosen comid for this PALM_Get.(*)
23:     Send the object part locations to the target process.(*)
24: end for
```

Explanatory comments on Alg. 5:
Step 5: The driver only continues to serve this `PALM_Get` when all processes of the target entity have announced it. Since the target processes wait for the answer in step 6 of Alg. 2, this implies a synchronization among the target processes.
Step 9: Since the target processes are waiting for the answer in step 6 of Alg. 2, this makes the target wait until a matching `PALM_Put` occurs.
Step 13: This is necessary to let the target entity continue since it is waiting for the answer in step 6 of Alg. 2.
Step 22: This answer is received in step 6 in Alg. 2.
Step 23: The object part locations are received in step 17 in Alg. 2.

Algorithm 6 Driver reaction on the `PALM_Get` connection request in step 21 of Alg. 2

```
1: if the source is the buffer then
2:     if the object is stored in the driver then
3:         Send the object part to the target process.(*)
4:     else
5:         Determine which memory slave has the object and order it to send the object part to
           the target process.(*)
6:     end if
7: else the source is not the buffer
8:     if the object part is served from the mailbuf then
9:         if the object is stored on the driver then
10:            Send the object part to the target process.(*)
11:        else
12:            Determine which memory slave has the object and order it to send the object part
               to the target process.(*)
13:        end if
14:    end if
15: end if
```

Explanatory comments on Alg. 6:
Steps 3, 5, 10 and 12: The object part is received in step 26 of Alg. 2.

3 New developments to enable dynamic spaces, dynamic distributors and dynamic sub-objects

As indicated above, the legacy OpenPALM version 4.1.4 exhibited several restrictions in the use of spaces, distributors and sub-objects. The goal of our work on OpenPALM is to overcome these restrictions. We have developed a mechanism which allows to use dynamic spaces not only for non-distributed objects, but also for distributed or replicated objects. Moreover, we have developed a means for changing the distributor and sub-object definitions during runtime of OpenPALM applications, while maintaining the consistency between sources and targets of communications without the need for additional synchronizations. Using a similar denomination as for the dynamic spaces, we call them *dynamic distributors* and *dynamic sub-objects*. In the following, we describe the implementation of the new features, and how they are used by means of the new API functions `PALM_Distributor_set` and `PALM_Subobject_set`.
In the legacy OpenPALM version 4.1.4, the dynamic spaces feature already existed, but with the restriction to be used only for non-distributed objects. Moreover, all distributors and sub-objects must have been defined at compile time, and could not anymore be changed at runtime. The new features overcome these restrictions through several modifications of the relevant mechanisms. In particular, it is not anymore necessary to define all distributors and sub-objects at compile time. All spaces, distributors and sub-objects which are already known at compile time of the OpenPALM application can be defined in the ID cards of the units, in PrePALM, or through dedicated functions which are implemented for the purpose of returning a distributor or sub-object definition, as it was already the case in the legacy version. This makes the new developments compatible with existing applications. But with the new version, it is also possible to only declare distributors and sub-objects without defining them, i.e. to only signalize their existence without providing a concrete distribution or a concrete sub-object definition. The definition can be given later at runtime through the API functions `PALM_Distributor_set` and `PALM_Subobject_set`, respectively. This new feature is particularly useful in the case when a distribution or sub-object cannot be known at compile time, e.g. because a parallel unit uses a domain decomposition which is determined only during runtime. In the legacy OpenPALM version, one needed to start the unit, memorize its data distribution and sub-objects, stop the application, provide the distribution and sub-object definitions, recompile the OpenPALM application, and finally run it. Even worse, as soon as any distribution or sub-object changed, it was necessary to repeat these steps. This could happen frequently, e.g. when using a different number of processes or a different computational grid. With the new version, one can save this effort and provide the distributions and sub-objects only when they are known during runtime. Of course, all affected distributors and sub-objects must be defined before a communication can be done. But this is no restriction in general, because it would be meaningless for a unit to communicate without knowing its data distribution and sub-objects. The necessary information would naturally be available in the units before they send or receive data.
Another modification of the mechanism is the introduction of a *dynamicity flag*. Each distributor and sub-object has an individual flag which can either hold the value `static` or `dynamic`. The value `static` means that the corresponding distributor or sub-object cannot be changed anymore after it has been defined for the first time, which might be at compile time or later at runtime. The value `dynamic` indicates that the corresponding distributor or sub-object is allowed to be changed an arbitrary number of times. We introduced this flag for a performance reason. It allows to skip the request for possible updates on distributors or sub-objects if they are `static`. All dynamicity flags must be set at compile time, and they cannot be altered during runtime. This is no restriction, because the user would usually know whether a unit needs to change a distributor or sub-object several times, or if it will stay fixed once it has been defined. Even if this is not known at compile time, one can set the flag to `dynamic` to not restrict the unit.
The driver as well as all processes of all entities hold lists of the spaces, of the distributors and of the sub-objects. These lists are used to keep track of all space, distributor and sub-object definitions. Whenever a space, a distributor or a sub-object changes, the new definition is appended as a new version in the corresponding list. Units can only set new versions for their own spaces, distributors or sub-objects, and all changes are announced to the driver. This ensures that units cannot disorder each others' lists, and that the driver is up to date with the newest versions of all lists at all times. Another consequence is that unnecessary synchronization between entities is avoided. Only when a communication event involves a dynamic space, distributor or sub-object on the remote side, the unit requests the driver for possible

updates.

Complementing the space, distributor and sub-object lists, we have enhanced the driver's commstate table. It keeps track of the space, distributor and sub-object versions which were used in communications. This is necessary to check space, distributor and sub-object compliance between source and target sides. It is also necessary to detect a mismatch situation where the source and the target side use different versions of a space, of a distributor or of a sub-object. To cope with that, we have implemented a redistribution functionality which is used in target units. In a mismatch situation, when the object is sent with an outdated version of a space, a distributor or a sub-object, the target entity switches back to that outdated version to receive the object. It aborts the transfer if the spaces have changed, since there is no way to transform an object from one space to another. However, if the distributor or the sub-object has changed, the object itself is still intact. Only its distribution or the sub-object selection may have changed. The target unit computes all necessary information so that it can internally redistribute the object to comply with the new distributor and sub-object.

3.1 New API routine PALM_Distributor_set

We have developed a new API routine which allows to define distributors in an OpenPALM application during runtime. Like the `PALM_Put` and `PALM_Get` routines, this new API routine can be used in the source code of units. It takes as input the distributor name and an encoded definition of the object distribution. It installs the new distributor version in the unit where the routine is called, and it announces it to the driver. Units can only set new versions for their own distributors which are marked with the `dynamic` flag, and for `static` distributors which have only been declared but not yet defined. The routine is described in Alg. 7.

Algorithm 7 `PALM_Distributor_set`

```
1: Input: distributor name, encoded definition of the new distribution.
2: Check if the calling entity uses the distributor for at least one of its objects, otherwise return.
3: Check if the calling process belongs to the localization of the distributor, otherwise return.
4: Check if the distributor is dynamic, or if it is static but still undefined, otherwise abort.
5: if this process has rank 0 in the associated localization then(*)
6:     Send a notification to the driver telling the size of the encoded definition of the new
       distribution.(*)
7:     Send the encoded definition to the driver.(*)
8: end if
9: Append the new version in the local distributor list.
10: Call MPI_Barrier on the localization.(*)
```

Explanatory comments on Alg. 7:
Step 5: Only the first process in the localization communicates with the driver.
Step 6: The driver's reaction to this notification is stated in Alg. 8.
Step 7: The driver receives the encoded distributor definition in step 1 in Alg. 8.
Step 10: Without this barrier, it could happen that a process other than the first process finishes setting the new distributor version and issues a communication before the first process has announced the new distributor version to the driver. This would result in a wrong mapping of the object parts between source and target. The barrier is necessary to prevent from that.

Algorithm 8 Driver reaction on the `PALM_Distributor_set` notification in step 6 of Alg. 7

```
1: Receive the encoded definition from the entity.(*)
2: Append the new version in the driver's distributor list.
```

Explanatory comments on Alg. 8:
Step 1: The encoded distributor definition is sent in step 7 of Alg. 7.

3.2 New API routine PALM_Subobject_set

Similar to the `PALM_Distributor_set` routine, we have developed another new API routine which allows to define sub-objects in an OpenPALM application during runtime. Like the `PALM_Put` and `PALM_Get` routines, this new API routine can be used in the source code of units. It takes as input the sub-object name and an encoded definition of the sub-object filter. It installs the new version in the unit where the routine is called, and it announces it to the driver. Units can only set new versions for their own sub-objects which are marked with the `dynamic` flag, or for `static` sub-objects which have only been declared but not yet defined. The routine is described in Alg. 9.

Algorithm 9 `PALM_Subobject_set`

```
1: Input: sub-object name, encoded definition of the new sub-object filter.
2: Check if the calling entity owns the sub-object, otherwise return.
3: Check if the calling process belongs to the localization of the object using this sub-object,
   otherwise return.
4: Check if the sub-object is dynamic, or if it is static but still undefined, otherwise abort.
5: if this process has rank 0 in the associated localization then(*)
6:     Send a notification to the driver telling the size of the encoded definition of the new
       sub-object.(*)
7:     Send the encoded definition to the driver.(*)
8: end if
9: Append the new version in the local sub-object list.
10: Call MPI_Barrier on the localization.(*)
```

Explanatory comments on Alg. 9:
Step 5: Only the first process in the localization communicates with the driver.
Step 6: The driver's reaction to this notification is stated in Alg. 10.
Step 7: The driver receives the encoded sub-object definition in step 1 in Alg. 10.
Step 10: Without this barrier, it could happen that a process finishes setting the new sub-object version and issues a communication before the first process has announced the new sub-object version to the driver. This would result in a wrong mapping of the object parts between source and target. The barrier is necessary to prevent from that.

Algorithm 10 Driver reaction on the `PALM_Subobject_set` notification in step 6 of Alg. 9

```
1: Receive the encoded definition from the entity.(*)
2: Append the new version in the driver's sub-object list.
```

Explanatory comments on Alg. 10:
Step 1: The encoded sub-object definition is sent in step 7 of Alg. 9.

3.3 Enhanced API routines `PALM_Put` and `PALM_Get`

In this subsection, we present the enhanced data transfer mechanism which is able to use dynamic spaces, dynamic distributors and dynamic sub-objects. We state the enhanced `PALM_Put` and `PALM_Get` routines in Algs. 11 and 12, respectively, and the enhanced implementations of the driver's reactions to these communication routines in Algs. 14 to 17. Some steps of the algorithms are marked with an asterisk indicating that an explanatory comment is given below.

Algorithm 11 Enhanced `PALM_Put`

```
1: Input: space name, object name, time, tag, pointer to local memory where the local object is
   stored.
2: Determine the comids which are relevant for this call to PALM_Put.(*)
3: if at least one of the affected spaces, distributors or sub-objects is dynamic, or static but
   still undefined then
4:     Send an update notification to the driver.(*)
5:     Send the last known versions of the spaces, distributors and sub-objects to the driver.(*)
6:     Receive the list of available updates from the driver.(*)
7:     for each new space, distributor or sub-object version do
8:         Receive the definition of the new space, distributor or sub-object version from the
           driver.(*)
9:         Append the new version in the corresponding local space, distributor or sub-object
           list.
10:    end for
11: end if
12: Set $flag = 0$.
13: for each comid do
14:     if this is a non-optimized communication then
15:         if $flag == 0$ then
16:             Notify the PALM_Put to the driver.(*)
17:             for each relevant comid do
18:                 Receive information from the driver whether this is a new object version.(*)
19:                 if this is the first process which contributes a part to the object then
20:                     Determine the newest available distributor and sub-object versions for the
                        target entity.
21:                     if source and newest target sub-object do not comply then
22:                         Determine the sub-object which this source process has last used for the
                            target.
23:                         if source and last used target sub-object do not comply then
24:                             Use the IDENTITY sub-object for the target.
25:                         end if
26:                     end if
27:                     if source and newest target distributor do not comply then
28:                         Determine the distributor which this process has last used for the target.
```

```
29:                 **if** source and last used target distributor do not comply **then**
30:                     Use the SINGLE_PROC distributor for the target.
31:                 **end if**
32:             **end if**
33:             Send the chosen distributor and sub-object versions to the driver.$^{(*)}$
34:         **else**
35:             Receive the chosen distributor and sub-object versions from the driver.$^{(*)}$
36:         **end if**
37:     **end for**
38:     Receive the driver's answer.$^{(*)}$
39:     Set $flag = 1$.$^{(*)}$
40: **end if**
41: Determine rank and current shape of the space.
42: **if** the source distributor or sub-object have changed since the last communication
    event **then**
43:     Compute the intersection of the source distributor and sub-object.
44: **end if**
45: **if** the target distributor or sub-object have changed since the last communication
    event **then**
46:     Compute the intersection of the target distributor and sub-object.
47: **end if**
48: **if** there is a new source or target distributor or sub-object version, or the source space
    has changed since the last communication event **then**
49:     **if** there is an old DOR **then**
50:         Delete the old DOR.
51:     **end if**
52:     Compute the new DOR.
53: **end if**
54: Allocate temporary memory for the DOR parts and disassemble the local object.
55: Determine which target processes shall receive a part of the disassembled local object
    from this source process.
56: **for** each process of the target's distributor **do**
57:     **if** that target process shall receive a part **then**
58:         **if** the object shall be written to a file **then**
59:             Write the object part to the file.
60:         **else**
61:             Send a connection request to the driver.$^{(*)}$
62:             Receive a connection authorization from the driver.$^{(*)}$
63:             Extract the communication type, the receiver entity type, and the receiver
                MPI process rank from the drivers answer.
64:             **if** the receiver is a unit or a memory slave **then**
65:                 Create an MPI intercommunicator to the receiver process.
66:             **end if**
```

```
                Send the object part to the receiver process.$^{(*)}$
                if the receiver is a unit or a memory slave then
                    Delete the intercommunicator to the receiver process.
                end if
            end if
        end if
        Notify the completion of the transfer of this object part to the driver.
    end for
    Deallocate the temporary memory for the DOR parts.
else this is an optimized communication
    Get the MPI intracommunicator for this object.
    Determine rank and current shape of the space.
    if the source distributor or sub-object have changed since the last communication
    event then
        Compute the intersection of the source distributor and sub-object.
    end if
    if the target distributor or sub-object have changed since the last communication
    event then
        Compute the intersection of the target distributor and sub-object.
    end if
    if there is a new source or target distributor or sub-object version, or the source space
    has changed since the last communication event then
        if there is an old DOR then
            Delete the old DOR.
        end if
        Compute the new DOR.
    end if
    Allocate temporary memory for the DOR parts and disassemble the local object.
    for each process of the target distributor do
        if that target process shall receive a part then
            Create an MPI intercommunicator to the receiver process.
            Send the object part to the receiver process.$^{(*)}$
            Delete the intercommunicator to the receiver process.
            Call MPI_Barrier on the local process group of this object's MPI intra-
            communicator.$^{(*)}$
        end if
    end for
    Deallocate the temporary memory for the DOR parts.
end if
end for
```

Explanatory comments on Alg. 11:

Step 2: Note that the calling EOS might be the source in several tubes. For each affected tube, the

time-tag and localization matching is done, and the corresponding comids are determined. The rank of the calling process with respect to the source distributor is derived from the localization.
Step 4: The driver's reaction to this notification is stated in Alg. 13.
Step 5: The driver receives the last known versions in step 1 of Alg. 13.
Step 6: The driver sends the list of available updates in step 3 of Alg. 13.
Step 8: The driver sends the new versions in step 5 of Alg. 13.
Step 16: The driver's reaction to this notification is stated in Alg. 14.
Step 18: This information is sent by the driver in step 4 of Alg. 14.
Step 33: The chosen versions are received by the driver in step 6 of Alg. 14.
Step 35: The chosen versions are sent by the driver in step 8 of Alg. 14.
Step 38: The driver's answer is given in step 27 of Alg. 14.
Step 39: The flag is used to ensure that the driver is notified only once, even if there are multiple comids. The driver can itself determine all relevant comids.
Step 61: The driver's reaction to this connection request is stated in Alg. 15.
Step 62: The driver's answer is given in steps 3, 8 or 24 of Alg. 15.
Step 67: The object part is received in step 48 of Alg. 12 in case of a direct communication, or in step 18 or 20 of Alg. 15 in case of an indirect communication, or in step 12 or 14 of Alg. 15 in case of a buffer communication.
Step 95: The object part is received in step 96 of Alg. 12.
Step 97: The MPI barrier is called on the local process group, i.e. only on the source side. This is necessary to avoid race conditions where several source processes could send object parts to the same target process for the same communication event.

Algorithm 12 Enhanced `PALM_Get`

```
1:  Input: space name, object name, time, tag, pointer to local memory where the local object
    shall be stored after reception.
2:  Check if there is a comid for this call to PALM_Get.
3:  if there is a comid then
4:      if this is a non-optimized communication then
5:          Notify the PALM_Get to the driver.(*)
6:          Receive the driver's answer.(*)
7:          Send the last known versions of the spaces, distributors and sub-objects to the
            driver.(*)
8:          Receive the list of available updates from the driver.(*)
9:          for each new space, distributor or sub-object version do
10:             Receive the definition of the new space, distributor or sub-object version from the
                driver.(*)
11:             Append the new version in the corresponding local space, distributor or sub-object
                list.
12:         end for
13:         Receive the versions of the source and target distributors and sub-objects which the
            sender has used for this object.(*)
14:         Check local mail buffer flag (*)
15:         Determine rank and current shape of the space.
16:         if the source distributor or sub-object have changed since the last communication
            event then
17:             Compute the intersection of the source distributor and sub-object.
18:         end if
```

```
19:     **if** there is a new target distributor or sub-object version since the last communication
        event, or the source entity did not use the current target distributor and
        sub-object **then**
20:         Compute the intersection of the target distributor and sub-object.
21:     **end if**
22:     **if** there is a new source or target distributor or sub-object version, or the source space
        has changed since the last communication event **then**
23:         **if** there is an old DOR **then**
24:             Delete the old DOR.
25:         **end if**
26:         Compute a new DOR as the intersection of the source distributor and sub-object
            and of target distributor and sub-object which the source entity has used.
27:     **end if**
28:     Get the number of source object parts for this target process.
29:     **if** the object shall be read from a file **then**
30:         Read local object parts from file.
31:     **end if**
32:     Receive information from the driver where the object parts are located.$^{(*)}$
33:     Determine if the object must be redistributed within the target entity after reception.
34:     **if** the object must be redistributed after reception **then**
35:         Allocate temporary memory for the local object redistribution.
36:     **end if**
37:     Allocate memory for the DOR parts.
38:     **for** each source process **do**
39:         **if** that source process contributes an object part **then**
40:             Send a connection request to the driver.$^{(*)}$
41:             Check if an MPI intercommunicator is needed.$^{(*)}$
42:             **if** an MPI intercommunicator is needed **then**
43:                 Create an MPI intercommunicator to the sender process.
44:             **end if**
45:             **if** the object must be redistributed after reception **then**
46:                 Receive the object part from the sender process in the temporary memory.$^{(*)}$
47:             **else**
48:                 Receive the object part from the sender process in the local object
                    memory.$^{(*)}$
49:             **end if**
50:             **if** an MPI intercommunicator was needed **then**
51:                 Delete the MPI intercommunicator to the sender process.
52:             **end if**
53:         **end if**
54:     **end for**
```

```
55:     if the object must be redistributed then
56:         Compute a temporary DOR as the intersection between the outdated target dis-
            tributor and sub-object versions which the source entity has used, and the newest
            target distributor and sub-object versions.
57:         Assemble the local object from the DOR which the source entity has used into the
            temporary memory.
58:         Disassemble the local object from the temporary memory into the source side of
            the temporary DOR.
59:         for each process of this entity which shall receive an object part from this process
                in the internal redistribution do
60:             Call the non-blocking MPI_Isend to send the object part to that process.
61:         end for
62:         for each process of this entity which shall send an object part to this process in
                the internal redistribution do
63:             Call the non-blocking MPI_Irecv to receive the object part from that process.
64:         end for
65:         Call MPI_Waitall to wait for the completion of all send and receive operations.
66:         Assemble the local object from the target side of the temporary DOR into the
            local object memory.
67:         Delete the temporary DOR and deallocate the temporary memory.
68:     else
69:         Assemble the local object from the target side of the DOR into the local object
            memory.
70:     end if
71:     Send a notification to the driver telling that this process has finished the PALM_Get.
72: else this is an optimized communication
73:     Determine whether any affected space, distributor or sub-object might require an
        update.
74:     if an update is required then
75:         Send an update notification to the driver.(*)
76:         Send the last known space, distributor and sub-object versions to the driver.(*)
77:         Receive the list of available updates from the driver.(*)
78:         for each available update do
79:             Receive the new version from the driver and append it to the corresponding
                space, distributor or sub-object list.(*)
80:         end for
81:         Unpack all updates into the corresponding version lists of the affected spaces,
            distributors and sub-objects.
82:     end if
83:     Determine rank and current shape of the space.
84:     Determine whether a new DOR must be computed.
```

```
85:         if a new DOR must be computed then
86:             if there exists already an old DOR then
87:                 Delete the old DOR.
88:             end if
89:             Compute the new DOR.
90:         end if
91:         Get the number of source DOR parts for this target process.
92:         Allocate temporary memory for the DOR parts if necessary.
93:         for each source process do
94:             if that source process contributes a DOR part then
95:                 Create an MPI intercommunicator to the sender process.
96:                 Receive the object part from the sender process.(*)
97:                 Delete the MPI intercommunicator to the sender process.
98:             end if
99:         end for
100:        Assemble the local object from the DOR.
101:    end if
102: end if
```

Explanatory comments on Alg. 12:

Step 5: The driver's reaction to this notification is stated in Alg. 16.
Step 6: The driver's answer is given in step 17 of Alg. 14 or in step 13 or 22 of Alg. 16.
Step 7: The driver receives the last known versions in step 18 of Alg. 14 or in step 23 of Alg. 16.
Step 8: The driver sends the list of available updates in step 19 of Alg. 14 or in step 24 of Alg. 16.
Step 10: The driver sends the new version in step 21 of Alg. 14 or in step 26 of Alg. 16.
Step 13: The driver sends the chosen versions in step 23 of Alg. 14 or in step 28 of Alg. 16.
Step 14: Units which are integrated into the same block can use local mail buffer memory to exchange data instead of using the driver's mail buffer.
Step 32: The object part locations are sent by the driver in step 24 of Alg. 14 in case of a direct communication, or in step 29 of Alg. 16 in case of an indirect or buffer communication.
Step 40: The driver's reaction to this connection request is stated in Alg. 17.
Step 41: An MPI intercommunicator needs to be created if the sender is not the driver, to which an intercommunicator exists anyway, and when the local mail buffer is not used.
Step 46: The object part is sent in step 10 or 12 of Alg. 17.
Step 48: The object part is sent in step 67 of Alg. 11 in case of a direct communication, or in step 10 or 12 of Alg. 17 in case of an indirect communication, or in step 3 or 5 of Alg. 17 in case of a buffer communication.
Step 75: The driver's reaction to this notification is stated in Alg. 13.
Step 76: The driver receives the last known versions in step 1 of Alg. 13.
Step 77: The driver sends the list of available updates in step 3 of Alg. 13.
Step 79: The driver sends the new versions in step 5 of Alg. 13.
Steps 96: The object part is sent in step 95 of Alg. 11.

Algorithm 13 Driver reaction on the space, distributor or sub-object update notification in step 4 of Alg. 11 or in step 75 of Alg. 12

```
1: Receive the last known versions of the spaces, distributors and sub-objects from the entity.(*)
2: Compare the last known versions of the entity with the newest available versions to determine
   the necessary updates.
```

```
3: Send the list of available updates to the entity.(*)
4: for each available space, distributor or sub-object update do
5:     Send the definition of the new version to the entity.(*)
6: end for
```

Explanatory comments on Alg. 13:
Step 1: The last known versions are sent in step 5 of Alg. 11 or in step 76 of Alg. 12.
Step 3: The list of available updates is received in step 6 of Alg. 11 or in step 77 of Alg. 12.
Step 5: The definition of the new version is received in step 8 of Alg. 11 or in step 79 of Alg. 12.

Algorithm 14 Enhanced driver reaction on the `PALM_Put` notification in step 16 of Alg. 11

```
1: Determine the comids which are relevant for this call to PALM_Put.
2: for each comid do
3:     Insert this PALM_Put into the commstate table.
4:     Send information to the source process whether this is a new object version.(*)
5:     if if this source process is the first process for this PALM_Put then
6:         Receive the space, distributor and sub-object versions which the source process uses
           for this PALM_Put, and store them in the commstate table.(*)
7:     else
8:         Read the space, distributor and sub-object versions which the first process used for
           this PALM_Put from the commstate table, and send them to the source process.(*)
9:     end if
10:    Search the commstate table for a waiting PALM_Get matching this comid.
11:    if there is a waiting PALM_Get then
12:        Update the commstate table that the PALM_Get will be served by this PALM_Put.
13:        if the source is the buffer then
14:            Determine if the object is located at the driver or a memory slave.
15:        end if
16:        for each target process do
17:            Send answer to target process.(*)
18:            Receive the last known space, distributor and sub-object versions from this target
               process.(*)
19:            Determine if updates are available for this target process, and send the list of
               available updates to the target process.(*)
20:            for each available update do
21:                Send the new version to the target process.(*)
22:            end for
23:            Read the space, distributor and sub-object versions which the source entity uses
               for this PALM_Put from the commstate table, and send them to this target
               process.(*)
24:            Send object part locations to this target process.(*)
25:        end for
26:    end if
```

```
    Send answer to the source process.$^{(*)}$
end for
```

Explanatory comments on Alg. 14:
Step 4: This information is received by the source process in step 18 of Alg. 11.
Step 6: The chosen versions for the spaces, distributors and sub-objects are send in step 33 of Alg. 11.
Step 8: The chosen versions for the spaces, distributors and sub-objects are received in step 35 of Alg. 11.
Step 17: This answer is received in step 6 of Alg. 12.
Step 18: The last known versions are sent in step 7 of Alg. 12.
Step 19: The update list is received in step 8 of Alg. 12.
Step 21: The new version is received in step 10 of Alg. 12.
Step 23: The chosen versions for the space, distributors and sub-objects are received in step 13 of Alg. 12.
Step 24: The object part locations are received in step 32 of Alg. 12.
Step 27: This answer is received in step 38 of Alg. 11.

Algorithm 15 Enhanced driver reaction on the `PALM_Put` connection request in step 61 of Alg. 11

```
Query the commstate table if this PALM_Put matches a waiting PALM_Get.
if a PALM_Get is waiting for the connection then
    Send a connection authorization to the source process telling that the communication type is direct, the target is a unit, and the MPI rank of the receiver process.$^{(*)}$
else the target is the buffer, or the object must be temporarily stored in the mailbuf
    Determine if the target is the buffer, or if the mailbuf shall take the object part.
    Determine if the driver or a memory slave shall receive the object.
    if the driver receives the object part then
        Send a connection authorization to the source process telling whether the communication type is buffer or indirect, the receiver is the driver, and the MPI rank of the driver.$^{(*)}$
    end if
    if the target is the buffer then
        if the driver receives the object part then
            Receive the object part from the source process and put it in the buffer.$^{(*)}$
        else
            Order a memory slave to receive the object part and to put it in the buffer.$^{(*)}$
        end if
    else
        if the driver receives the object part then
            Receive the object part from the source process and put it in the mailbuf.$^{(*)}$
        else
            Order a memory slave to receive the object part and to put it in the mailbuf.$^{(*)}$
        end if
    end if
```

```
23:     if a memory slave receives the object then
24:         Send a connection authorization to the source process telling whether the communi-
            cation type is buffer or indirect, the receiver is a memory slave, and the MPI rank of
            the memory slave.(*)
25:     end if
26: end if
```

Explanatory comments on Alg. 15:
Steps 3, 8 and 24: This answer is received in step 62 of Alg. 11.
Steps 12, 14, 18 and 20: The object part is sent in step 67 of Alg. 11.

Algorithm 16 Enhanced driver reaction on the `PALM_Get` notification in step 5 of Alg. 12

```
1: Determine the comids which are relevant for this call to PALM_Get.
2: Query the commstate table to choose a comid.
3: if this PALM_Get does not read from a file then
4:     if not all processes of the target entity have announced this PALM_Get yet then
5:         Return.(*)
6:     end if
7:     if there is neither a matching PALM_Put currently active nor is the object available in the
       mailbuf or buffer then
8:         Memorize the PALM_Get to be wakened later in step 11 of Alg. 14 when a matching
           PALM_Put occurs.
9:         Return.(*)
10:    end if
11: else the object is read from a file
12:    for each target process do
13:        Send a dummy answer to the target process.(*)
14:    end for
15: end if
16: for each target process do
17:    for each source process do
18:        if the object part is in the mailbuf then
19:            Determine whether the object part is stored on the driver or a memory slave.
20:        end if
21:    end for
22:    Send an answer to the target process telling the chosen comid for this PALM_Get.(*)
23:    Receive the last known space, distributor and sub-object versions from this target
       process.(*)
24:    Determine if updates are available and send the list of available updates to the target
       process.(*)
25:    for each available update do
26:        Send the new version to the target process.(*)
27:    end for
```

```
28:     Read the space, distributor and sub-object versions which the source entity uses for this
        PALM_Put from the commstate table, and send them to this target process.$^{(*)}$
29:     Send the object part locations to the target process.$^{(*)}$
30: end for
```

Explanatory comments on Alg. 16:
Step 5: The driver only continues to serve this `PALM_Get` when all processes of the target entity have announced it. Since the target processes wait for the answer in step 6 of Alg. 12, this implies a synchronization among the target processes.
Step 9: Since the target processes are waiting for the answer in step 6 of Alg. 12, this makes the target wait until a matching `PALM_Put` occurs.
Step 13: This is necessary to let the target entity continue since it is waiting for the answer in step 6 of Alg. 12.
Step 22: This answer is received in step 6 in Alg. 12.
Step 23: The last known versions are sent in step 7 of Alg. 12.
Step 24: The update list is received in step 8 of Alg. 12.
Step 26: The new version is received in step 10 of Alg. 12.
Step 28: The chosen versions are received in step 13 of Alg. 12.
Step 29: The object part locations are received in step 32 in Alg. 12.

Algorithm 17 Enhanced driver reaction on the `PALM_Get` connection request in step 40 of Alg. 12

```
1: if the source is the buffer then
2:     if the object is stored in the driver then
3:         Send the object part to the target process.$^{(*)}$
4:     else
5:         Determine which memory slave has the object and order it to send the object part to
           the target process.$^{(*)}$
6:     end if
7: else the source is not the buffer
8:     if the object part is served from the mailbuf then
9:         if the object is stored on the driver then
10:            Send the object part to the target process.$^{(*)}$
11:        else
12:            Determine which memory slave has the object and order it to send the object part
               to the target process.$^{(*)}$
13:        end if
14:    end if
15: end if
```

Explanatory comments on Alg. 17:
Steps 3, 5, 10 and 12: The object part is received in step 46 or 48 of Alg. 12.

4 Conclusion

We presented algorithmic sketches of the legacy and of the enhanced communication mechanism of Open-PALM. The main new feature is the dynamic distributors feature, which enables to determine and change the distribution of data objects among parallelized OpenPALM units any time during runtime of the coupled application. Furthermore, we enhanced the existing dynamic spaces feature to work not only with non-distributed objects as it was the case in the legacy OpenPALM version, but also with distributed objects in combination with the dynamic distributors feature. In addition, we developed the new dynamic sub-objects feature, which allows to determine and change sub-object definitions any time during runtime of the coupled application. All of these features have been implemented in OpenPALM version 4.2.3, which is available for download as open source[2].

The new features support common practices in numerical simulations such as coarsening, refinement or load balancing, which have previously been difficult to manage or even impossible with OpenPALM. With the legacy version, users needed to determine data sizes, distributions and sub-objects through offline testing before the actual coupled application could run, and were not able to change them anymore during runtime. This yielded substantial overhead for setting up the coupling. Moreover, to change any distributor, any sub-object, or any space of parallel units, one needed to stop the application, repeat the offline testing, and restart the application. With the new dynamic features, one can completely avoid this overhead. Instead, one can adjust OpenPALM's communication mechanism whenever necessary during runtime of the coupled application. Typical use cases for the dynamic features are the determination of matrix and vector sizes, domain decompositions or boundary and interface data during runtime in the setup phase of units, and the adjustment of spaces, distributors and sub-objects after coarsening, refinement or load balancing.

The new features remove the offline testing overhead for setting up coupled applications, and they remove the need to stop, test, reconfigure and restart the application whenever the communication mechanism shall be adapted. Instead, the dynamic spaces, dynamic distributors and dynamic sub-objects features together render OpenPALM's communication mechanism fully dynamic and flexible. They have been adopted in different applications, such as a natural convection scenario using a coupling of fluid and temperature models [6], or in nutrient cycle simulations using a coupling of biogeochemical and hydrology models [3, 7].

Acknowledgements

This work was supported by the German Research Foundation (DFG) under grant no. HE 4760/4-1. The authors greatly thank Prof. Olivier Thual at Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique (CERFACS) in Toulouse for hosting a research stay to support the developments.

References

[1] S. Buis, A. Piacentini, D. Declat, and the PALM Group. Palm: A computational framework for assembling high-performance computing applications. *Concurr. Comput. Pract. Exp.*, 18:231–245, 2006.

[2] T. Lagarde, A. Piacentini, and O. Thual. A new representation of data-assimilation methods: The PALM flow-charting approach. *Q. J. R. Meteorol. Soc.*, 127:189–207, 2001.

[3] S. Klatt D. Kraus E. Haas R. Kiese K. Butterbach-Bahl P. Kraft L. Breuer M. Wlotzka, V. Heuveline. Parallel multiphysics simulations using OpenPALM with application to hydro-biogeochemistry coupling. In *accepted for Proceedings of 6th Int. Conf. on High Perform. Sci. Comput., March 16-20, 2015, Hanoi, Vietnam*, 2016.

[4] T. Morel, F. Duchaine, A. Thevenin, A. Piacentini, M. Kirmse, and E. Quemerais. *Open-PALM coupler version 4.1.4: User guide and training manual*, 2013.

[5] A. Piacentini and the PALM Group. PALM: A Dynamic Parallel Coupler. *Lecture Notes in Computer Science*, 2565:479–492, 2003.

[6] M. Wlotzka and V. Heuveline. A parallel solution scheme for multiphysics evolution problems using OpenPALM. *EMCL Prepr. Ser.*, 1, 2014.

[2] `http://www.cerfacs.fr/globc/PALM_WEB/user.html` retrieved on June 13, 2017

[7] M. Wlotzka, V. Heuveline, S. Klatt, E. Haas, D. Kraus, K. Butterbach-Bahl, P. Kraft, and L. Breuer. *Handbook of Geomathematics*, chapter Simulation of Land Management Effects on Soil N_2O Emissions using a Coupled Hydrology-Biogeochemistry Model on the Landscape Scale. Springer, 2014.