NUTS: a Distributed Object-Oriented Platform with High-Level Communication Functions

Enn Tyugu
Tallinn University of Technology
90 PUBLICATIONS **574** CITATIONS
SEE PROFILE

Vladimir Vlassov
KTH Royal Institute of Technology
122 PUBLICATIONS **772** CITATIONS
SEE PROFILE

NUTS: a Distributed Object-Oriented Platform with High-Level Communication Functions

Enn Tyugu, Vladimir Vlassov and Mattin Addibpour

Department of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, Sweden

E-mail: {tyugu, vlad, mattin}@it.kth.se

Contact person:

Prof. Enn Tyugu, Teleinformatics, KTH Electrum 240, S-164 40 KISTA (Stockholm)

Voice: +46 8 752 1371

Fax: +46 8 751 1793

Abstract

An extensible object-oriented platform NUTS for distributed computing is described which is based on an object-oriented programming environment NUT, is built on top of the Parallel Virtual Machine (PVM), and hides all low-level features of the latter. The language of NUTS is a concurrent object-oriented programming language with coarse-grained parallelism and distributed shared memory communication model implemented on a distributed memory architecture. It differs from other languages of concurrent programming in the following: concurrent processes are represented by packages which are semantically richer entities than objects, inter-process communication is performed in terms of classes, objects, scripts and packages, using the EDA communication model; processes can be arranged into structured collections: grids which enable one to program data-parallel computations on a high level; sequential segments of programs can be synthesized automatically from specifications represented as classes using the program synthesis features of NUT. Examples of usage of generic parallel computing control structures PARDIF and PARARR are given.

1.0 Introduction

Distributed computing is gaining popularity in scientific problem solving in the same way as in other computing applications. Distributed memory computers like IBM SP2 provide sufficient computation power to tackle computationally hard scientific problems using the technique of parallel computing. However, applicability of the latter is restricted due to complexity of the program development for distributed computing platforms. Higher level programming tools take almost no advantage of distributed computing facilities, at the best, providing access to them by means of a separate set of low level functions. The basic level of concurrent object-oriented programming has essentially remained unchanged since publication of the book [14] in 1989. From the other side, the growing popularity of user-friendly tools like MATLAB among researchers demonstrate that it is a time for developing high level user-friendly scientific computing software which could enable an easy access also to distributed computer architectures. Development of this kind of software has been a goal of the work presented in this paper. More precisely, our goal has been to develop tools for declarative programming of distributed processes and, in this way, to extend the limits of programmability of sophisticated problems by users with restricted program development resources. Our starting point was a declarative programming technique supported by automatic synthesis of sequential programs - a field where we have been working for quite a long time.

This paper provides an overview of a distributed computing platform NUTS developed as an extension of the object-oriented programming environment NUT which includes program synthesis features and has been used for implementing a number of declarative problem-oriented languages. The language of NUTS is a concurrent object-oriented programming language with coarse-grained parallelism implemented on a distributed memory architecture. It differs from other languages of concurrent object-oriented and distributed programming in the following:

- Concurrent processes are represented by NUT packages which are semantically richer entities than objects.
- Inter-process communication supports transmission of classes, objects, scripts and packages, and hides all low-level communication details.
- Processes can be arranged into structured collections: grids which enable one to program parallel computations on high level.
- Segments of programs can be synthesized automatically from specifications represented as classes.

The last aspect is a distinguished feature of the NUT programming environment enabling users easily to implement declarative problem-oriented languages.

Developing the NUTS system, we have used two existing software tools: an object-oriented programming environment NUT [9, 11] and the Parallel Virtual Machine, PVM [6]. The NUT system will be briefly described in section 2. Here we give some hints about the PVM.

The Parallel Virtual Machine, PVM, is a widely used integrated framework for heterogeneous computing. A multitasked PVM program runs on a set of networked computers which are composed into a virtual multiprocessor. PVM supports parallel computation at the level of Unix processes, using message passing for communication. PVM is composed of the PVM daemon, `pvmd`, and the PVM library, `libpvm`. An instance of the `pvmd` runs on each host of the virtual machine and provides host-process interactions. The `libpvm` constitutes an interface between PVM applications and the run-time system (PVM daemons). A PVM task can be spawned from another active task. PVM tasks communicate using tagged messages. A task can send messages asynchronously, and is responsible for receiving messages directed to it using blocking or non-blocking receive procedures.

Because the PVM is still a low-level tool, a number of programming environments, such as: PADE, CODE 2.0, HeNCE, were implemented to simplify developing and debugging of parallel programs that use PVM.

PADE, the Parallel Applications Development Environment, supports all phases of development of PVM application: editing, compilation, execution, debugging and performance measurement. The PADE package has graphical user interface and it allows the user to develop a parallel application which consists of a number of components located on the file systems of different computes with different operating systems [5].

HeNCE (Heterogeneous Network Computing Environment) and CODE 2.0, are visual programming environments each of which is based on visual parallel programming language using directed graphs as a natural mechanism for presentation of the behaviour of parallel programs [3]. Nodes of a graph denote conventional Fortran or C subroutines and arcs represent data and control dependencies.

The NUTS system also contains visual programming tools which are a part of the NUT environment. We are not considering here the visual tools, but concentrate on the interprocess communication model, communication functions and extensibility of the language. After brief introduction of the NUT programming environment, we describe the control and communication functions of NUTS in section 3. Our goal has been to provide an easily extensible set of control structures for distributed parallel computing in the form of classes which can be used by a program synthesizer. We consider to be important the possibility to develop distributed programs in a compositional style. The declarative style of program development applicable in NUT is directly extended to programming in NUTS, as we can see from the examples presented in section 4.

The NUTS system consists of the following three components:

- NUT programming environment.
- Library of distributed control and communication functions, `librnut`.
- NUTS daemon, `nutd`.

The NUT programming environment is a tool for development of large programs in the object-oriented style; it enables the user to specify classes graphically by drawing their schemas; it combines object-oriented programming, visual programming and automatic program construction paradigms.

`Librnut` is a library of NUT routines for building a system of collaborative NUT processes running on PVM, for exchanging classes, scripts and objects between NUT processes.

The daemon `nutd` is an executable code which is used to manage NUT process control and display connections. It is responsible for halting PVM when all NUT processes exit or are killed.

2.0 The NUT programming environment

The NUT programming environment supports development of large programs in the object-oriented way. NUT language has features of an object-oriented language with multiple inheritance, parametric polymorphism and a flexible mechanism for message passing [11]. The NUT system has good interoperability with programs written in C and, in general, with other software components running under Unix. Each software package used in NUT is represented by an object of a specific class designed for this package. We use MATLAB software in this way in an example in section 4.

We are going to use the following concepts related to the NUT system: scripts, objects, classes and packages. **Script** is a top level program for immediate execution. Writing scripts in a NUT environment has the role similar to programming in a Unix shell. All data and programs except scripts are **objects**. Objects created from scripts are global objects. Each object has a **class**. Classes in NUT serve, besides the ordinary object-oriented usage, as specifications which can be used for constructing programs automatically. In particular, a method `compute` is available by default in every user-defined class, and it can be sent to any object `x` as a request to compute some of its components, let us say, `y`. The expression `x.compute(y)` produces a value of `y` if the class of `x` contains information for computing this value. The largest modular unit in NUT is **package**. It is a collection of programs, specifications and data, related to a computation. More precisely, it is a collection of scripts, classes, global objects and graphic views (icons, images and schemes) intended for usage in the particular computation or in a whole problem domain. A package can be visualised in a NUT main window, Figure 1. It shows a list of classes (`DirMatrix`, `Dirichlet`, etc.), a list of global objects (`M`, `dim`, etc.), and a script in a workspace for immediate execution. A package constitutes an environment where a process described by a script in the workspace is executed.

The NUT language has some specific features which are essential for our presentation:

- Equations as specifications of computations;
- Encapsulation border different from ordinary encapsulation of objects;
- Automatic program synthesis;
- Runtime compilation of modified/new generated classes.

The first feature permits to specify arithmetic programs with single assignment property simply as sets of equations. The order of computing variables from equations (solving equations) can be determined automatically. This, together with automatic type inference, allows us to define classes in the equational programming style. For example, the following class specifies a surface in the three-dimensional space of `x`, `y`, `z` given by one fixed parameter `ca`:

```
class screw
rel
   alpha=ca*z;
   x=r*cos(alpha);
   y=r*sin(alpha);
```

NUT uses weaker encapsulation of objects, allowing direct naming of immediate components outside of objects in order to bind them in a specification. The following class uses this feature and specifies a producer/consumer pair compositionally in a shared memory style:

```
class prc
super parexec;
var
```

```
   A : producer;
   B : consumer in=A.out;
```

The lower level implementation in NUTS will be a distributed implementation, but this is hidden inside the class `parexec`.

The program synthesis feature of NUT can be explained briefly as follows. All methods specified explicitly by programs are supplied with external views, which are automatically translated into the form of logical axioms. Also equations, equalities and overall structure of classes are translated into axioms. After unfolding all specifications relevant to a program synthesis task, a set of axioms is obtained which describes a theory about the particular synthesis task. The method of structural synthesis of programs (SSP) is used in order to get (completely automatically) a program for solving the task. This method is complete in the following sense. It produces a program for any solvable task and produces an answer about the unsolvability otherwise. Solvability of a task is defined with respect to a specification, i.e. a theory obtained from a class. The SSP has a sound logical explanation in terms of intuitionistic propositional logic and simple types, as well as in terms of higher-order constraint networks [9, 10]. Due to the simplicity of the logical language of SSP, its performance is quite satisfactory which can be seen from the Table 1.

Table 1:

Maximum number of axioms in a theory	10000
Maximum number of propositions in a theory	16000
Average synthesis time	< 1 sec
Average number of lines of source specification	< 20
Maximum expansion rate at unfolding	exponential

A program synthesis task may appear in several forms. First, it can be the message `compute` sent to an object as described in the beginning of this section. Second, it may appear as the keyword `spec` in the body of a method which has to be synthesized according to its external specification as in the following example:

```
r: x -> y{spec};
```

The keyword `spec` here shows that a program for computing `y` from `x` has to be synthesized using as a specification the class where this is written. Third, an external specification of a relation may contain one or more constructions of the following form

```
[a, ..., b -> c, ..., d]
```

where `a,...,d` are names of objects. This construction is called a **subtask** of the method where it appears and its meaning is a request to synthesize a program for computing `c,...,d` from `a,...,b`. The synthesized program becomes an input parameter of the method.

Let us have now a problem of finding a total moment applied to the piece of the surface described above in the class `screw` and restricted by the inequalities 0 < `z` < `zmax` and 0 < `r` < `rmax`, `cr` > `cmax*dr`, where `rmax=cr-z*dr`, `zmax`, `cr`, `dr` are given constants. The moment `T=f*y` is caused by a linear field influencing each point of the surface with the force `f=cf*(cx - x)` in the direction of x-axis, where `cf` and `cx` are given constants. Obviously, in order to solve the problem, we have to integrate along `z` from 0 to `cz` and for each `z` along `r` from 0 to `cr`. This is a typical problem solvable in NUT from a declarative specification. Let us assume that we already have a class describing the concept of integral, and this class has variables `res` for calculated value of integral, `arg` for integration variable and `fun` for value of integrated function. We can write the following specification in NUT:

```
class integration
var
   S: screw;
   I1: integral from=0, to=rmax, arg=S.r, val=T;
   I2: integral from=0, to=zmax, arg=S.z, val=I1.res;
rel
   rmax=cr-S.z*dr;
   T=f*S.y;
   f=cf*(cx-S.x);
   r: zmax,cr,dr,cf,cx -> I2.res{spec};
init
   cr:=3;
   dr:=1;
   zmax:=150;
```

In this specification we have used, first, weak encapsulation for binding values of components of different objects, second, equations for specifying required computations, and third, the keyword `spec` for requesting automatic synthesis of the program we need. It is sufficient in NUT to specify the class `integral` as follows:

```
class integral
var
   from, to, arg, val, res: num;
rel
   [arg->val] from, to, step -> res
    {< an arbitrary integration method >};
```

Later we shall continue this example, redefining the class `integration` for distributed computing by giving some additional specifications.

3.0 Distributed control and communication functions

We present a concise description of control and communication functions of NUTS in this section, showing how they hide details of PVM from the user. These functions provide unified approach to communication and synchronisation, and enable the user to program exclusively in terms of objects, classes and scripts. The NUTS library of distributed control and communication functions, `librnut`, includes the following groups of functions to provide PVM configuration, process control and communication.

- Process control functions are used to start and halt the PVM, to spawn and control NUT processes, to get information about running NUT processes.
- Script passing functions are used for passing and performing scripts of workspaces of NUT processes.
- Class passing functions are used for passing and compiling texts of classes.
- Object passing functions are used for sending and receiving objects through the space of shared objects.

Passing scrips and classes is not used for process synchronization, rather it allows to distribute and control computation in collaborative NUT processes. Synchronization and communication mechanism of NUTS is based on object passing according to the EDA multiprocessing model [8].

The NUTS library `librnut` is implemented on top of PVM library `libpvm` which provides low level inter-process communication based on explicit message passing.

3.1 Process Control

The distributed NUTS program is structured as a number of collaborative NUT processes running on PVM. Each process runs in a separate NUT environment consisting of a NUT package. Each process can have an open NUT window where the lists of classes and objects are visible together with the script. The environment of a process can be changed dynamically by the process itself or by other processes.

The library `librnut` includes routines for control of NUT processes and configuration of PVM. There are routines to spawn a number of child NUT processes (`rnut_spawn`), to close and to open a window interface of the remote NUT (`rnut_chmod`), to kill (`rnut_kill`) or to send an exit request to set of NUT processes (`rnut_exit`), to load a package (`rnut_pack`).

The NUT process started first is called a root NUT process. Most of routines from `librnut` enrol the root process into PVM. Each of them starts PVM if it is not running already, and spawns daemon `nutd` on the host which will be used to display NUT windows. The library `librnut` contains two routines `rnut_addhosts` and `rnut_delhosts` which are used to change the configuration of the virtual machine. These routines also notify the daemon `nutd` about hosts which are added or deleted from PVM. Respectively, `nutd` adds or removes given hosts from the list of hosts allowed to connect to the X server to open a display connection.

Any number of child NUT processes can be spawned from any active NUT using `rnut_spawn(n, mode, where, package)` routine. This routine creates `n` processes running in a mode defined by `mode` on a host named `where` and loads a package named `package` into all spawned processes. A child process can be spawned with open or closed window interface. If argument `where` is not defined or `nil` then PVM is responsible for choosing the convenient set of hosts to start up new NUT processes. Each spawned child obtains a unique PVM task identifier, tid, which is used to specify the destination address for data passing and remote control. Each NUT process running in PVM can get its task identifier using `rnut_mytid` routine, which returns its tid.

In the following example two NUT processes start with open window interface on the host called `‘walrus’` with package `‘eda.mem’`. An array `tids` contains task identifiers of children.

Example 1.

```
mytid := rnut_mytid();
tids := rnut_spawn(2,2,‘walrus’,‘eda.mem’);
```

During spawning, the parent NUT sends the task identifiers of its children to the daemon `nutd` and also to each newly spawned child in order to notify them about their siblings. Each child can get this information if needed using `rnut_parent` and `rnut_mygrid` routines. The first routine returns tid of the parent, the second one returns an array of tids of all siblings including the calling NUT process.

Example 2.

```
parent:= rnut_parent();
stids := rnut_mygrid();
```

A parent and all its child NUT processes spawned by one and the same action `rnut_spawn` constitute a complete graph, which we call **grid**, from the accessibility point of view, i.e. each processes of a grid can directly communicate with any other process of the grid. The spawn routine guarantees the same structure of copies of the array of tids which are distributed among all members of the grid. Each member of the grid can spawn their children and form a new grid, being itself a parent. Figure 3 illustrates a process of spawning as a tree, where nodes are NUT processes and arcs represent spawn relations.

Most of librnut routines use tids as destination addresses for communication. Sometimes it is more convenient to use a metrics of integer numbers 0, 1, 2,..., n instead of task identifiers (tids) for addressing NUT processes. The classes grid and mygrid support also relative addressing of processes where 0 is an address of the parent and 1, 2,..., n are addresses of children in a grid. The routine rnut_mynum can be used to get a number of the calling NUT process in the grid.

The remote control of a NUT process from another NUT process is implemented by sending appropriate requests, which can be served when the X-event loop in the receiving NUT process is idle and the process is not in the Intepreter stage.

The PVM termination is implemented using a semaphore in the daemon nutd. During spawning, the parent NUT notifies the daemon about its children, and the semaphore is incremented by the number of spawned NUT processes. Each NUT process can exit independently of other processes. When a task exits or is killed, the PVM daemon pvmd notifies the daemon nutd, and the semaphore is decremented. When the semaphore becomes zero, nutd halts PVM. The asynchronous exit of NUT processes can be the reason of deadlock while accessing shared objects, classes or workspaces. But interactive nature of the NUT programming technology, as well as remote process control routines give a possibility to handle these deadlocks and to solve the termination problem.

3.2 Passing and Performing Scripts

The following routines are intended for passing a script in the form of a textual object for a workspace of NUT processes:

```
rnut_putws*( tids, nnut, ws );
rnut_getws*( tid );
```

and the following routines combine passing and performing a script into a single action:

```
rnut_pputws*( tids, nnut, ws );
rnut_pgetws*( tid );
```

Where the asterisk ‘*’ should be one of two characters: `a` (append) or `r` (replace).

Routines `rnut_putws*`, `rnut_pputws*` allow to multicast a script of workspace `ws` to `nnut` processes specified by `tids`. Routines `rnut_getws*`, `rnut_pgetws*` are used to get a current script of workspace from remote NUT process specified by `tid`. The received script is appended to the current workspace of the receiving process, if a routine has a suffix `a`. The received script replaces the current workspace of the receiving process, if a routine has a suffix `r`.

The routine `rnut_perform` multicasts a request to perform current scripts of requested processes. The user is responsible for providing all classes and objects, which are necessary to perform a script.

3.3 Passing Classes

The library `librnut` contains routines that enable a NUT process to send a copy of a class to a set of NUT processes (`rnut_clcpy`), and to get a copy of a class from another NUT process (`rnut_clget`). Both routines allow to rename the copy of the class in the destination NUT processes. In the following example, the first statement sends a copy of the class named `‘grid’` with a new name `‘gr’` to processes specified by an array `tids`. The next statement sends a request to get a copy of the class named `‘mygrid’` with the same name `‘mygrid’` from parent process:

```
rnut_clcpy(tids, length(tids), ‘gr’, ‘grid’);
```

```
rnut_clget(rnut_parent(), ‘mygrid, ‘mygrid’);
```

Being received, the text of the class is automatically compiled. If a NUT process already contains a class named as the received class then the new class overwrites the old one. It is safer, however, to use package passing (`rnut_pack`) instead of passing classes separately, because care must be taken of sending classes in a proper order.

3.4 Passing Objects

The communication model of NUTS is object passing which is based on communication aspects of the Extended Dataflow Actor model, EDA. A formal description and software implementation of the EDA model can be found in [2, 8, 12].

3.4.1 The EDA Communication Operations

EDA is a model of object-oriented multithreaded computation. An EDA object contains local and shared variables and a thread of control. All shared variables form a space of shared memory which is accessible from each object. In EDA, passing data between local and shared memory is represented by a set of special store and fetch operations on a spaces of local and shared variables. An EDA object can store local data into shared memory and fetch a value of a shared variable into its local memory. A shared variable may be in one of two states: full (containing data) or empty. The EDA model [8] recognizes three types of shared variables: **x**, **i** and **s**, each with special synchronization requirements. Semantics of typed shared variables of EDA was extended and expressed in terms of special shared memory operations carried out on untyped shared variables [2, 13]. This modification was done in order to make the EDA model more convenient and flexible. The modified EDA model specifies eight kinds of shared memory operations:

- **x-fetch** is a blocking extract operation. It is used for extracting the data from a full shared variable to a local variable of an object. If the shared variable is full, its value is extracted and its state becomes empty. If the shared variable is empty, then x-fetch request is enqueued on this shared variable, and the executing thread is suspended until the variable becomes full by a store operation.
- **s-fetch** is a non-blocking extract operation. It is used for extracting the data from a shared variable to a local variable of an object. If the shared variable is full, its value is extracted and its state becomes empty. If the shared variable is empty, then s-fetch returns an empty value.
- **i-fetch** is a blocking copy operation. It is used for copying the data from a shared variable to a local variable of an object. If the shared variable is full, its value is copied and its state remains full. If the shared variable is empty, then i-fetch request is enqueued on this shared variable, and the executing thread is suspended until the variable becomes full by a store operation.
- **u-fetch** is a non-blocking copy operation. It is used for coping the data from a shared variable to a local variable of an object. If the shared variable is full, its value is copied and its state remains full. If the shared variable is empty, then i-fetch returns an empty value.
- **x-store** is a blocking store operation. It is used for storing the data to a shared variable from a local variable of an object. If the shared variable is empty, a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full, then x-store request is enqueued on this shared variable, and the executing thread is suspended until the variable is emptied by a fetch operation.
- **s-store** is a non-blocking buffering store operation. It is used for storing the data to a

shared variable from a local variable of an object without suspension. If the shared variable is empty, a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full, then computation in the executing thread resumes as soon as the local value is buffered. The buffer with the local data is enqueued on this shared variable until the variable is emptied a the fetch operation.

- **i-store** is a non-blocking store operation, which can be called as write-once. It is used for storing the data to a shared variable from a local variable of an object without suspension. If the shared variable is empty, a local value is simply copied into the shared variable and its state becomes full. If the shared variable is full, then i-store operation is ignored.
- **u-store** is a non-blocking unconditional update operation. It is used for updating a value of a share variable with a value of a local variable of an object. u-store always copies a local value into the shared variable independently of its state.

3.4.2 Object Passing Functions in NUTS

An EDA object is represented in NUTS by a NUT process running in a separate NUT environment consisting of a NUT package; EDA shared and local variables are represented by NUT objects called **shared** and **local** respectively. A shared object is considered empty, if it has `nil` value or does not exist, otherwise it is full. In communication routines a shared object is addressed by means of two items **tid** and **objname**:

- **tid** is a task identifier of a NUT process in which the object is located,
- **objname** is a name of the object in NUT.

The `librnut` contains eight communication functions for object passing which realise the semantics of EDA store and fetch operations on a space of shared and local

NUT objects (see Table 2).

Table 2: Communication Functions

Name `rnut_`	Action	full	empty
`xfetch`	extract a meaning value from a shared object to local	extract	suspend
`sfetch`	extract any value from a shared object to local	extract	return `nil`
`ifetch`	copy a meaning value from a shared object to local	copy	suspend
`ufetch`	copy any value from a shared object to local	copy	return `nil`
`xstore`	store a value from a local object to an empty shared object	suspend	store
`sstore`	store a value from a local object to an empty shared object	buffering	store
`istore`	store a value from a local object to an empty shared object	skip	store
`ustore`	store a value from a local object to a shared object	update	store

A storing routine multicasts a request to store a value of a local object into shared objects located in remote NUT processes. A fetching routine sends a request to extract or to copy a value from a remote object into a local one. All fetching routines, as well as `rnut_xstore` routine, block the requesting process until a value or acknowledgements arrive. A requested remote process can serve the request only if it is not in the Interpreter stage, i.e. it does not perform anything.

Object passing routines have the following form where the asterisk `'*'` should be one of four characters `x`, `s`, `i`, or `u`:

```
rnut_*store(tids, nnut, objname, obj)
obj := rnut_*fetch(tid, objname);
```

Storing routines, `rnut_*store`, are used to store a value of the object `obj` into a shared object, named `objname`, of each NUT process, specified in the array `tids`. A storing routine sends a value `obj` to NUT processes defined by `tids` with a request to store a value of `obj` into shared objects named `objname`. Three routines, `rnut_istore`, `rnut_sstore` and `rnut_ustore` are non-blocking. It means that computation on the requesting process resumes as soon as all requests are sent. The routine `rnut_xstore` blocks the requesting process until `nnut` acknowledgments have arrived from requested processes. The received value can be consumed in the receiving process by different ways according to an access type: x, i, s, or u, and a state of the object `objname`. To provide an arbitrary access to shared objects NUTS uses `nil` value to define an empty state of an object. The user is responsible for avoiding possible deadlocks, because if the object named `objname` is not empty, then the x- and s-store request are enqueued until the `objname` will be emptied by some fetching routine. To provide a correct synchronization of processes, and to avoid possible deadlocks, it is recommended to consume values stored into shared objects by extracting them into local objects using matching fetching procedures: `rnut_xfetch` or `rnut_sfetch`. Each of these routines extracts a value from a full shared object and serves the first store request suspended in a store request queue of this object. If the waiting request is `rnut_xstore`, then fetching procedure sends an acknowledgement to the requesting NUT process.

Fetching routines, `rnut_*fetch`, can be used to extract or to copy a value into the local object `obj` from the shared objects, named `objname` and located in the NUT process, specified by `tid`. A fetching routine sends the name `objname` to NUT process with a request to extract or to copy a value from an object, named `objname`. Two routines `rnut_xfetch` and `rnut_sfetch` generate an extract request, two other routines `rnut_ifetch` and `rnut_ufetch` generate a copy request. A requesting process becomes suspended until a value has arrived from a requested process. The received value is assigned to the local object `obj`. Fetching of the value is visible in the same manner as creation of an object `objname` by means of `new` expression in the consuming process. After that, a user can use the object `objname` in an ordinary way.

Two routines, `rnut_sfetch` and `rnut_ufetch`, are non-blocking. Two other fetching routines, `rnut_xfetch` and `rnut_ifetch`, are blocking. It means that the requested process can execute fetch request only if the object named `objname` is full, i.e. has a value. Otherwise the fetch request is queued until the `objname` will be filled by the matching storing routine.

Communication functions are most suitable for using in matching pairs:

- `rnut_xstore` and `rnut_xfetch`, for supporting mutually exclusive interprocess communication.
- `rnut_sstore` and `rnut_sfetch`, for supporting object streams between NUT processes.
- `rnut_istore` and `rnut_ifetch`, for supporting write-once shared object for synchronizing single writer-multiple readers and OR-parallelism.
- `rnut_ustore` and `rnut_ufetch`, for unconditional updating of shared objects.

The following example illustrates using of the pair `rnut_istore` and `rnut_ifetch` routines to realise the OR-parallelism. The process `Boss` chooses the first object from several objects sent to it by its child processes `Worker1,...,` `Workern`. Each child process sends a value of its local object `work_out` into the shared object named `‘boss_in’` located in the process `Boss` specified by a task identifier `tid_b`. The process `Boss` consumes only the earliest sent data from the object `‘boss_in’` using `rnut_ifetch` routine.

Process `Worker`$_I$ (I = 1,..., N):

```
tid_b := rnut_parent();
% compute worker_out
worker_out := rnut_mytid();
% send the value of worker_out to the Boss process
rnut_istore([tid_b], 1, ‘boss_in’, worker_out);
```

Process `Boss`:

```
% consume the first answer from children
my_in := rnut_ifetch(rnut_mytid(), ‘boss_in’);
```

The next example illustrates using of `rnut_sstore` routine for sending a set of objects through the one and the same shared object located in a remote process. The process `A` sends five objects of different classes into the shared object named `‘sh_buf’` and located in the remote NUT process specified by `tid_b`. The receiving process `B` extracts this data from the object `sh_buf` using `rnut_xfetch` routine and builds an array of objects of different classes which is allowed in NUT.

Process `A`:

```
dest := [tid_b];
buf  := ‘sh_buf’;
rnut_sstore(dest, 1, buf, 5);
rnut_sstore(dest, 1, buf, -1.3 );
rnut_sstore(dest, 1, buf, ‘Hello World’);
rnut_sstore(dest, 1, buf, [7, 3.1415]);
rnut_sstore(dest, 1, buf, [‘radio’, ‘-ga’, ‘-ga’]);
```

Process `B`:

```
x := new array of any;
me:= rnut_mytid();
```

```
for i := 1 to 5 do
   x[i]:= rnut_xfetch(me, ‘sh_buf’);
od;
```

Figure 4 illustrates changing a value of the object named `‘sh_buf’` and the resulting value of the local object `x` which consumes a stream of values from the process `A`.

4.0 Programming Experience with NUTS

The flexibility of NUTS high level communication functions combined with the open programming environment NUT can be exploited in order to make different distributed programming environments. The intention is to construct and run parallel branches of computations automatically, using a set of predefined control structures. These control structures are implemented in the form of classes in NUT. Once preprogrammed, the control structures are expected to constitute a domain-specific parallel programming toolkit [1].

4.1 Control Structures for Parallel Programming Using NUTS

The task of developing parallel programs consists of a number of sub-tasks: parallel algorithm design, problem partitioning, mapping, communication and synchronization management and resource management. A domain-specific parallel programming environment in the form of a hierarchical set of NUTS classes has been realised to facilitate design and implementation of parallel applications in NUTS, and to achieve their efficient processing on PVM. A set of control structures supports automatic mapping, communication and synchronization management as well as resource management in the development of parallel applications. Control structures are divided into two levels: administrative and problem-oriented. These control structures are then used by classes at the third (application) level, in order to construct a parallel application (Fig.5).

There is one control structure, called `PAR_ENGINE`, on the administrative level. It is written in an application-independent style and is used for managing the configuration of PVM, spawning and administrating a pool of NUT processes needed by objects of the underlying classes.

The problem-oriented level contains generic control structures, each of which differs in its distributed computation and communication patterns. The problem-oriented control structures are used for parallel solving of specific problems on a pool of NUT processes provided by `PAR_ENGINE`. They hide all aspects of process creation, communication and synchronisation from the application level. For example, class `PARSEARCH` is for solving the breadth-first search problems, `PARDIF` is intended for the class of problems which can be parallelized using data partitioning and `PARARR` supports parallel processing of elements of arrays.

The bottom level contains application classes, which specify solutions of specific tasks. For example, `Dirichlet` is a class which specifies the Dirichlet problem, `Integration` solves the screw problem and the `15-puzzle` class solves the well known 15-puzzle problem.

NUT classes of the second and third level are expected to constitute a domain-specific class library for parallel programming.

Control structures support all phases of execution from starting PVM and spawning tasks until collecting the result and killing remote processes. Most of the PVM initialisation, process creation and control is preprogrammed in `PAR_ENGINE`. Therefore, `PAR_ENGINE` can be used as a basis for developing new control structures. In our context, the main functionality of a control structure is to support automated instantiation of computational fragments. Each control structure supports also a communication pattern related to its operating domain. In order to support communication between computational fragments, a set of names are defined for each control structure (e.g. `par_in, par_out` in the example below). These predefined names of communication components are then used inside classes at the application level and act as bindings between remote processes.

4.2 Solving the Screw Example Using PARARR

The problem-oriented control structure PARARR is used for parallel processing of elements in an array (each element can be object of an arbitrary class). The communication pattern of the PARRAR is illustrated in Fig.6. A NUT root process executes method `Exec` of an object of the PARARR class. The root spawns p workers and creates p objects of an application class each of which is initialized by an element of an input array. The root process distributes these objects to workers with a request to compute an element of an output array. Each worker asks for all necessary classes from the root, synthesizes an algorithm for computation, executes it and sends the result to the root process. The PARARR structure uses `PAR_ENGINE` as an interface to PVM, hence all functions related to the PVM initialization and process control are hidden in `PAR_ENGINE`.

A specification of the PARARR class defines two communication components, `par_in` and `par_out` of arbitrary classes, which constitute the object passing interface between a group of workers and the root. An element of the input array is passed by the PARRAR object to the `par_in` component of each application object created for a worker. The result computed by the worker is passed to the `par_out` component of the object as an element of the output array collected by PARRAR in the root process. It is convenient to use an alias specification included to an application level class for a worker. In our example, it specifies an input element in the form of a vector (`I2.from`, `I2.to`) and an output element as `I2.res` produced by the worker:

```
alias
   par_in  = (I2.from,I2.to);
   par_out = I2.res;
```

PARARR supports also a set of reduction methods to perform reduce operations (such as sum, max) across all collected results.

We shall compute the double integral in the screw example (see section 2.0) in parallel using the control structure PARARR. The integration interval `I2` is divided into `p` sub-intervals each of which will be processed by a separate worker. The following NUT script is executed by the root process to calculate a value of the double integral represented as `result`, using 4 workers:

```
pararr := new PARARR;
in_arr :=[[0,4],[4,8],[8,12],[12,16]];
pararr.Exec('integration',in_arr);
result := pararr.ReduceAdd();
```

The first statement of the script creates an instance of the PARARR class. The input array `in_arr` specifies four integration sub-intervals to be computed in parallel. The method `Exec` in the PARARR class spawns workers and distributes four objects of class named `integration` (each of which is initiated by an element of `in_arr`*)*:

```
class integration
var
   S: screw;
   I1: integral from=0, to=rmax, arg=S.r, val=T;
   I2: integral arg=S.z, val=I1.res;
alias
   par_in  = (I2.from,I2.to);
   par_out = I2.res;
rel
```

```
   rmax=cr-S.z*dr;
   T=f*S.y;
   f=cf*(cx-S.x);
   r: zmax,cr,dr,cf,cx -> I2.res{spec};
init
   cr:=3;
   dr:=1;
   zmax:=16;
```

Objects of the `integration` class are sent to workers. Each worker calculates and sends the result `I2.res` (alias `par_out`) to the root process which collects them and computes the final result using method `ReduceAdd`.

4.3 Solving the Dirichlet Problem Using PARDIF

As depicted in Fig.6, the control structure PARARR provides communications only between the root and the workers. The more general control structure `PARDIF` creates a grid (one, two or three dimensional) of NUT processes with a communication pattern illustrated in Fig.7.

Each worker from the grid can communicate with its neighbours. `PARDIF` can be used specially for problems which can be parallelized using data partitioning in one, two and three dimensions. `PARDIF` object in the root process distributes objects of an application class to workers as well as a script for communication control. Each worker can communicate with the root process using `par_in` and `par_out` components as in the PARARR communication pattern. To communicate with neighbours, a worker can use up to twelve communication components in an object of the application class (see Fig. 8):

```
set_up, get_up, set_down, get_down,
set_left, get_left, set_right, get_right,
```

`set_back, get_back, set_forth and get_forth`

The first four components are used to communicate in one dimensional grid, the first eight components in two dimensional grid and all the components are needed in three dimensional grid of workers.

Each object of the application class which is sent to a worker, is initialized by `PARDIF` in the same way as in `PARARR`, by using `par_in`. Workers can perform computations iteratively. The number of iterations and the grid dimension are specified in the `PARDIF` and application objects during their initialization. During each iteration, a worker can communicate with its neighbours through the communication components mentioned above (`set_up`, `get_up`, etc.). After each iteration, all workers send the results to the root through the `par_out` component.

To demonstrate a usage of the `PARDIF` structure, we have implemented a two-dimensional Laplace equation solver with Dirichlet boundary conditions, using the simultaneous displacement method of Jacobi [4, 7]. This problem called the Dirichlet problem is formulated as:

$$\nabla^2 F = \frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = 0 \qquad \text{(EQ 1)}$$

To solve this equation, the surface *F* is partitioned into an *n* by *n* grid whose elements represent the initial state of *F*. The value of *F* at the surface boundaries is a known constant. In each iteration, a new value of *F* is calculated as:

$$F_{x,y} = (F_{x-1,y} + F_{x,y-1} + F_{x+1,y} + F_{x,y+1}) / 4 \qquad \text{(EQ 2)}$$

In the NUT application, the grid of the surface is represented by matrix which contains $n \times n$ elements. The initial matrix is divided into partitions, each of which is processed by a separate worker. On each iteration, the workers exchange their boundary rows and send results of the iteration to the root process to display. Figure 9 illustrates a scheme of main data communication between workers and the root process.

The following NUT script is executed by the root process to calculate and display values of the matrix 100 by 100 during 1000 iterations, using four workers connected into one dimensional process grid:

```
% Creating objects:
pardif := new PARDIF;
M      := new DirMatrix;
matlab := new MATLAB;

% Initializations:
partitions  :=  M.BuildPart(4);%partitioning  the  initial
matrix.
max_iter   := 1000;
dim        := 1;

% Starting remote processes:
pardif.Exec(`Dirichlet', partitions, max_iter, dim);

% collecting results and sending to MATLAB for drawing
for i to max_iter do
   m := pardif.result;
   matlab.HotImage(M.BuildMatrix(m));
od;
```

The root process creates an object `pardif` of the `PARDIF` class, initializes the object `M` of the `DirMatrix` class and divides it into 4 partitions constituting an array `partitions`. These partitions are passed to the object `pardif` as the second parameter. The root performs the method `Exec` in the object `pardif` in order to spawn four workers, create and initialize four objects from the application class `Dirichlet`, and distribute them to workers for computing. Compared to the `PARARR`, the method `Exec` in `PARDIF` has two more arguments: the number of iterations and the dimension of partitioning. The result of each iteration is collected in the component `par-dif.result` and passed to display using the object `matlab`. (It is interesting to notice how easy it is to build an interface to existing software like MATLAB in NUT. One just has to define an inerface class `MATLAB` with methods for using the software. It is so due to the fact that NUT is interoperable with C and Unix -- one can even execute shell commands from NUT programs.)

Each object of the application class `Dirichlet` has the following components for communication with its neighbours and the root:

- `par_in` is used for fetching a partition of the initial matrix from the root
- `par_out` is used for passing the calculated value of the partition to the root after each iteration
- `set_down`, `set_up` (the output boundaries) are used for passing newly computed boundaries of the partition to the neighbour workers
- `get_down`, `get_up` (the input boundaries) are used for fetching boundaries of partitions from a neighbour workers.

The Dirichlet class is specified as follows:

```
class Dirichlet
var
   my_part, par_in, par_out: DirMatrix;
   get_down, get_up,
   set_down, set_up: Array of num;
rel
   par_out = my_part;

   Init: par_in -> my_part, set_down, set_up
```

```
{
    my_part  := par_in;
    set_up   := my_part.GetRow(1);
    set_down := my_part.GetRow(my_part.m);
};

Iteration: my_part, get_down, get_up-> set_down, set_up
{
    my_part.Iter(get_down, get_up);
    set_up   := my_part.GetRow(1);
    set_down := my_part.GetRow(my_part.m);
};
```

The equation (EQ 2) is implemented as a method `Iter` in the class `DirMatrix`.

5.0 Concluding remarks

A distributed object-oriented computing platform, NUTS, provides a dynamic set of collaborative NUT processes running on PVM. Each NUT process runs in a user-friendly interactive programming environment which contains a NUT package. The NUT environment is an integrated system which is based on object-oriented NUT language and supports automatic program systhesis and runtime compilation of classes. A NUT package is the largest modular unit in NUTS. It encapsulates sets of scripts, classes, objects and graphics views. Running NUT processes can be controlled from one host by one user as well as from a number of hosts by multiple users.

NUTS exploits all possibilities provided by the PVM system for parallel machine configuration and process control. However a PVM communication model based on explicit message passing would require great efforts to construct a flexible and complex schemes of synchronous and asynchronous process communication for an *interac-*

tive object-oriented distributed programming environment such as NUTS. Therefore the NUTS library `librnut` contains two sets of specialized communication routines: (1) Functions for passing classes, scripts and loading packages, and (2) Functions for object passing. Functions from the first set are not used for process synchronization, they are non-blocking, asynchronous and do not need acknowledgments from a destination.On the other hand the object passing functions based on the EDA multiprocessing model provide a flexible and unified approach to inter-process communication and synchronization.

As programming environment NUTS smoothly merges declarative programming with distributed programming, allowing specification of problems on a high level. We have demonstrated using PARARR and PARDIF classes as examples that NUTS allows the user to construct different domain-specific parallel programming environments in the form of hierarchical libraries of classes representing control structures. High-level structures written in a domain-independent style can provide a set of scalable communication patterns predefined on a dynamic grid of NUT processes. PVM configuration, process control and explicit object passing are encapsulated into control structures. An unified application interface with a NUTS control structure can be specified and used in the same way as in global class declaration and usage.

The NUT programming environment has flexible graphical user interface and graphical tools, such as Scheme Editor with a possibility of generation of class, script and any other texts specified by user. This feature of NUT allows one to simplify development of both sequential and distributed NUT applications using visual object-oriented programming. Graphical representation of classes including control structures developed by the Scheme Editor constitute a graphical language. Program synthesis capabilities of NUT provide a possibility to implement rich semantics of graphical descriptions, including description of parallel programs.

The NUT system together with its documentation is available by anonymous ftp from *ftp.it.kth.se* where it is in the directory *Software/CSlab/Software-Engineering/NUT*. Papers related to he NUT system can be found in www under *http://www.it.kth.se/CSlab/Software-Engineering/Projectpage.htm*l.

Acknowledgments

This work could not been done in the present form without the ideas developed in the parallel architectures group of the Teleinformatics Department and we wish to thank especially Lars-Erik Thorelli and Hallo Ahmed for the discussions on the EDA model. We have a pleasure to thank Tarmo Uustalu for many useful discussions on the design of rNUT Toolkit. This research is supported by the Swedish National Board for Industrial Development, NUTEK (grant No. 9303405-2). Vladimir Vlassov is supported by a scholarship from the Wenner-Gren Centre Foundation for Scientific Research.

References

[1] M. Addibpour, *Control Structures for Parallel Computing in NUT.* MSc thesis TRITA - IT-R 95:18, CSLab, Dept. of Teleinformatics, Royal Institute of Technology (KTH), Stockholm, 1995.

[2] H. Ahmed, L-E Thorelli, and V. Vlassov, “mEDA: A Parallel Programming Environment”, in *Proceedings of the Euromicro’95 Conference on Design of Hardware/Software Systems, Como, Italy, September 1995.*

[3] J.C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton, *Visual Programming and Debugging for Parallel Computing*, Technical Report TR94-229, Dept. of Computer Sciences, Univ. of Texas at Austin, 1994.

[4] K.M. Chandy and S Taylor, *An Introduction to Parallel Programming*. Jones and Bartlett

Publishers, Boston, 1992, pp. 103-122.

[5] J.E. Devaney, R. Lipman, M. Lo, W.M. Mitchell, M. Edwards, and C.W. Clark, *The Parallel Application Development Environment (PADE). A User's Manual.* National Institute of Standards and Technology, 1995.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM3: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[7] R.H. Perrot, *Parallel Programming.* Addison-Wesley Publishing Company, G.B., 1987, pp. 150-153.

[8] L.-E. Thorelli, *The EDA Multiprocessing Model.* Technical Report TRITA-IT-R 94:28, CSLab, Dept. of Teleinformatics, Royal Institute of Technology (KTH), Stockholm, 1994.

[9] E. Tyugu, "Using classes as specifications for automatic construction of programs in the NUT system," *Journal of Automated Software Engineering*, vol. 1, pp. 315 - 334, 1994.

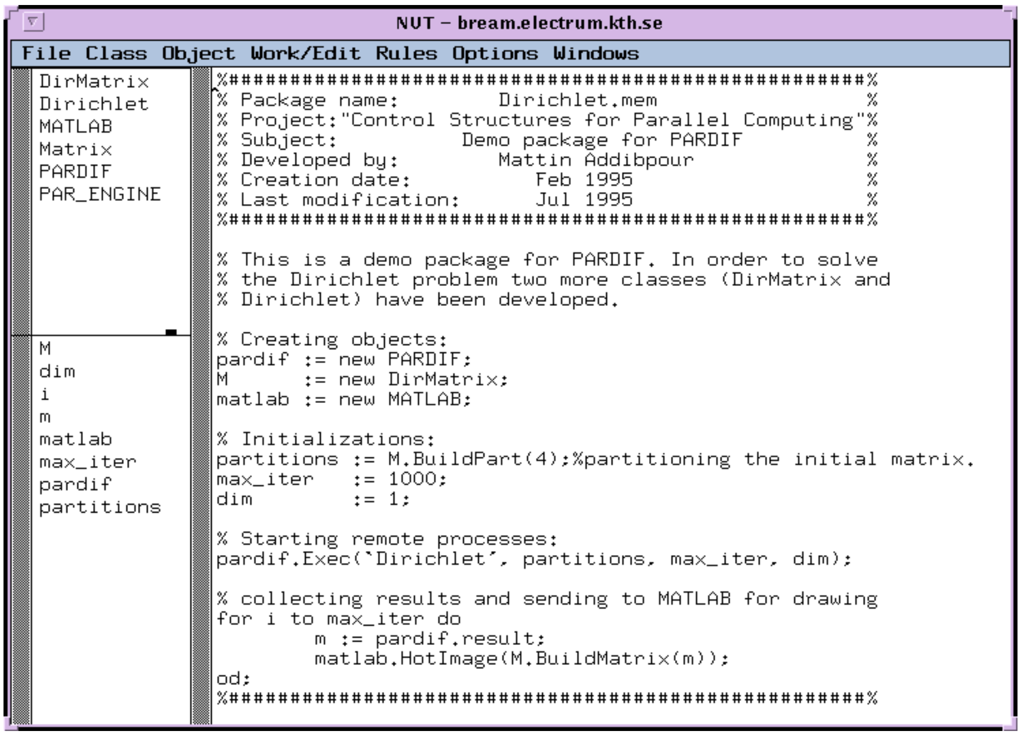
[10] E. Tyugu and T. Uustalu, "Higher-order functional constraint networks," *Constraint programming. NATO ASI Series F: Computer and System Sciences*, vol. 131, pp. 116 - 139, 1994.

[11] T. Uustalu, U. Kopra, V. Kotkas, M. Matskin, and E Tyugu, *The NUT Language Report.* Technical Report TRITA-IT-R 94:14, CSLab, Dept. of Teleinformatics, Royal Institute of Technology (KTH), Stockholm, 1994.

[12] V. Vlassov, H. Ahmed, and L-E Thorelli, "mEDA2: An Extension of PVM", in *Proceedings of the 3rd International Conference on Parallel Computing Technologies, St.-Petersburg, Russia, September 1995*. pp. 288-293.

[13] V. Vlassov, E. Tyugu, and M. Addibpour, *Distributed programming toolkit for NUT.* Technical Report TRITA-IT-R 94:34, CSLab, Dept. of Teleinformatics, Royal Institute of Technology (KTH), Stockholm, 1994.

[14] A. Yonezawa and M. Tokoro (eds.), *Object-oriented concurrent programming,* The MIT Press, 1989.


NUT – bream.electrum.kth.se
File Class Object Work/Edit Rules Options Windows

```
DirMatrix
Dirichlet
MATLAB
Matrix
PARDIF
PAR_ENGINE
```

```
M
dim
i
m
matlab
max_iter
pardif
partitions
```

```
%####################################################%
% Package name:         Dirichlet.mem                %
% Project:"Control Structures for Parallel Computing"%
% Subject:          Demo package for PARDIF          %
% Developed by:         Mattin Addibpour             %
% Creation date:            Feb 1995                 %
% Last modification:        Jul 1995                 %
%####################################################%

% This is a demo package for PARDIF. In order to solve
% the Dirichlet problem two more classes (DirMatrix and
% Dirichlet) have been developed.

% Creating objects:
pardif := new PARDIF;
M      := new DirMatrix;
matlab := new MATLAB;

% Initializations:
partitions := M.BuildPart(4);%partitioning the initial matrix.
max_iter   := 1000;
dim        := 1;

% Starting remote processes:
pardif.Exec(`Dirichlet´, partitions, max_iter, dim);

% collecting results and sending to MATLAB for drawing
for i to max_iter do
        m := pardif.result;
        matlab.HotImage(M.BuildMatrix(m));
od;
%####################################################%
```

FIGURE 1 NUT main window

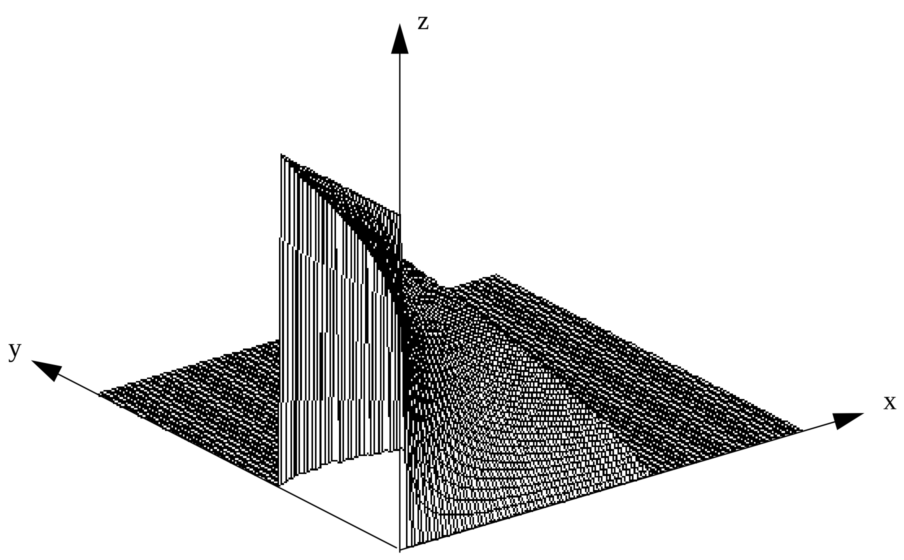


FIGURE 2 The screw example

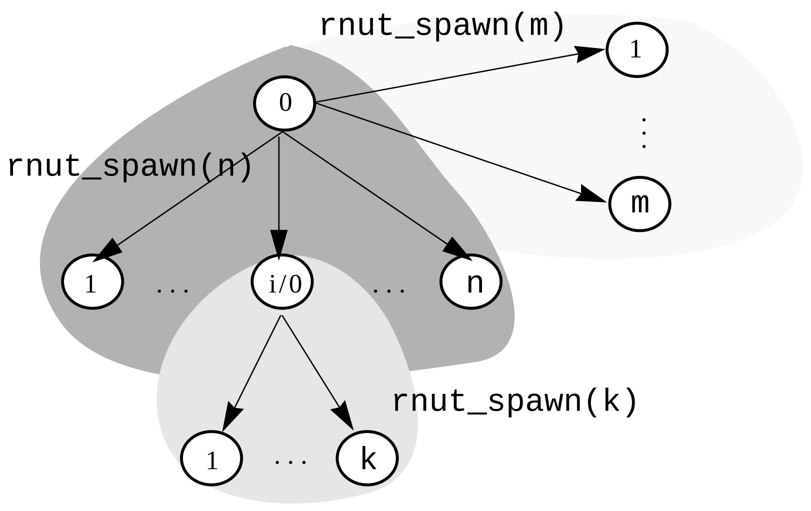
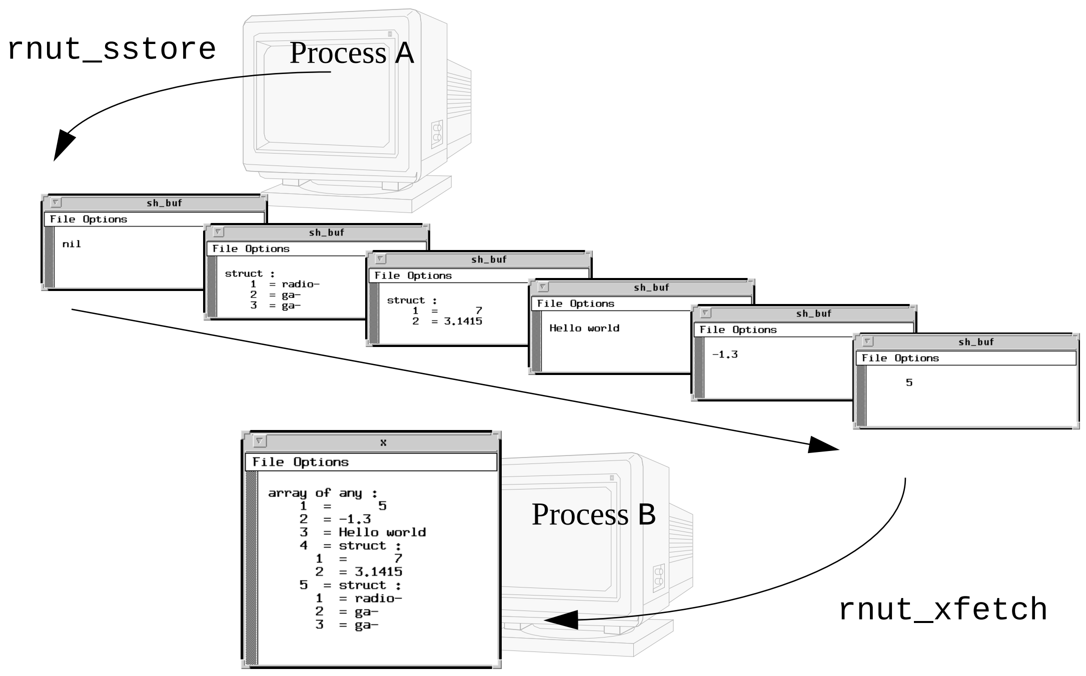


FIGURE 3 A spawn tree with three grids



```
nil
```

```
struct :
    1  = radio-
    2  = ga-
    3  = ga-
```

```
struct :
    1  =       7
    2  = 3.1415
```

```
Hello world
```

```
-1.3
```

```
     5
```

```
array of any :
    1  =       5
    2  = -1.3
    3  = Hello world
    4  = struct :
      1  =       7
      2  = 3.1415
    5  = struct :
      1  = radio-
      2  = ga-
      3  = ga-
```

FIGURE 4 Stream of objects

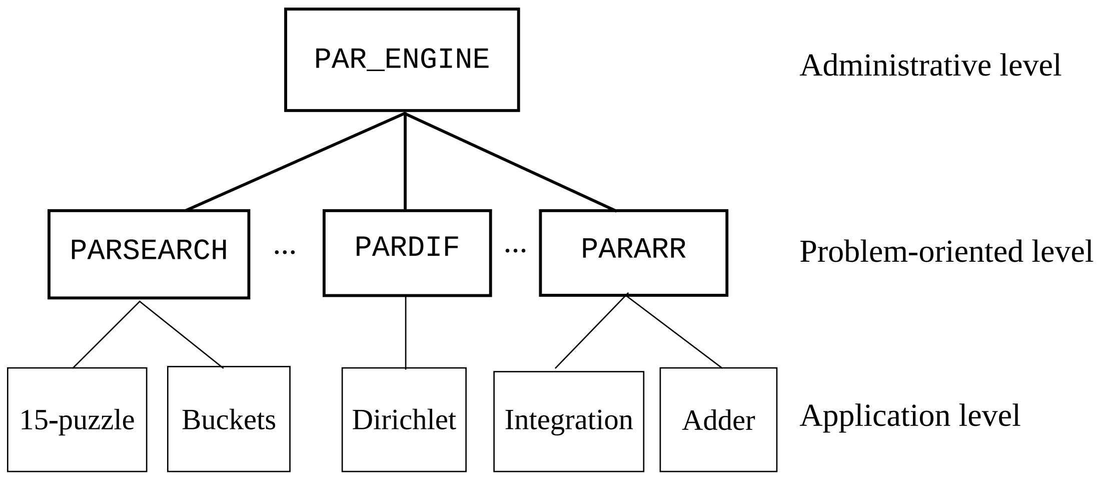


FIGURE 5 Class hierarchy for domain-specific parallel programming

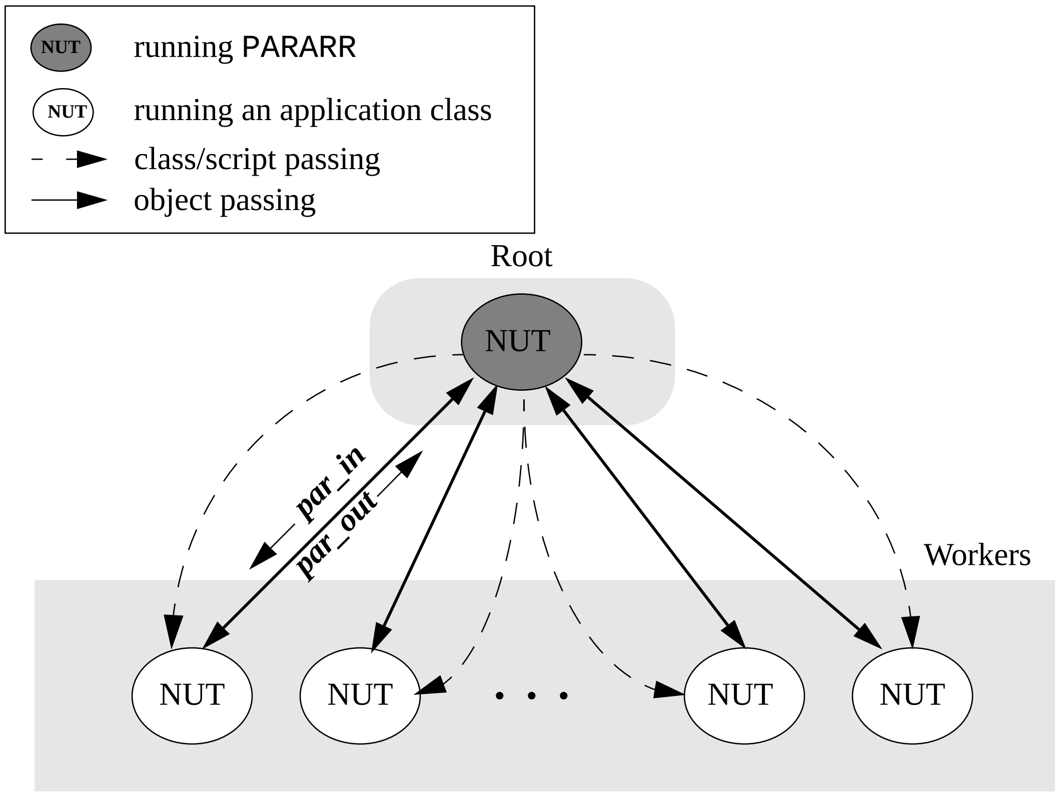


FIGURE 6 The communication pattern of PARARR

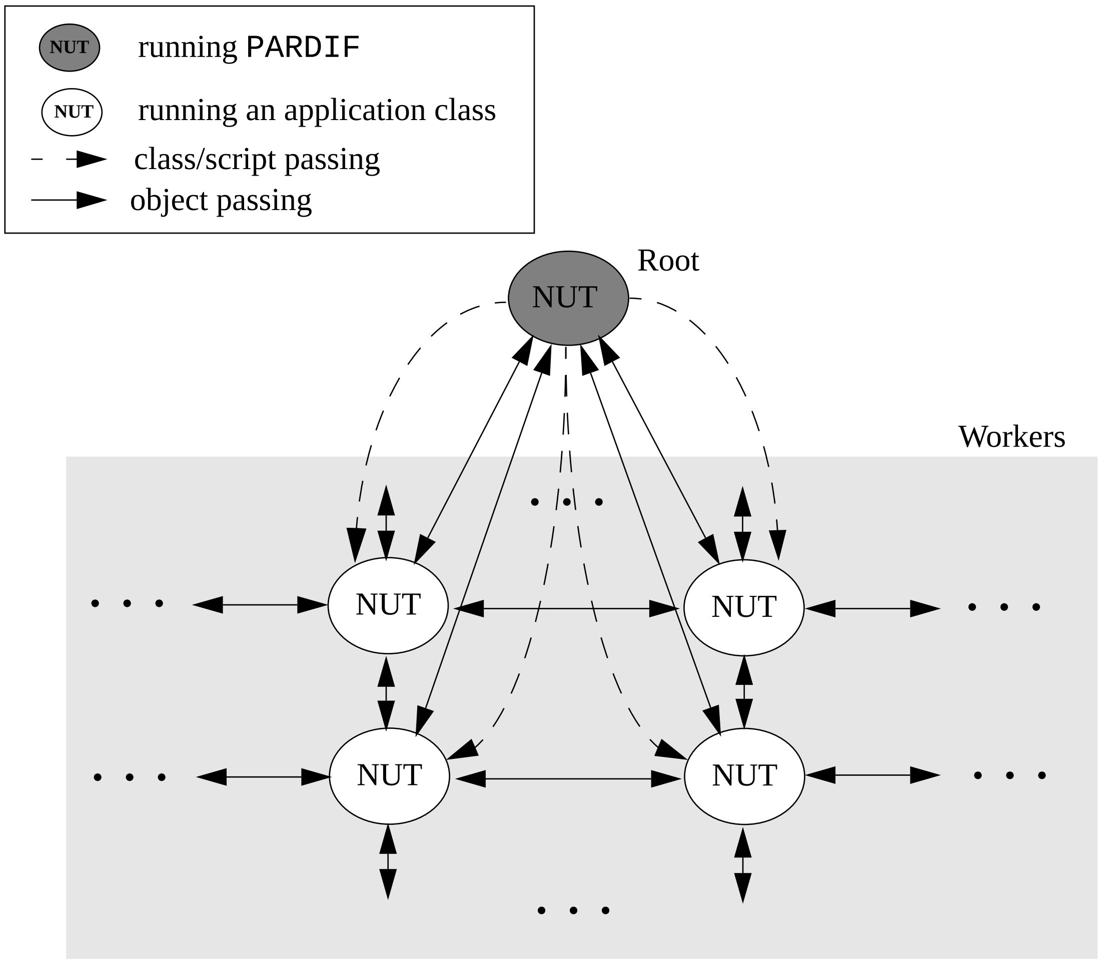


FIGURE 7 The communication pattern of `PARDIF` in two dimensions

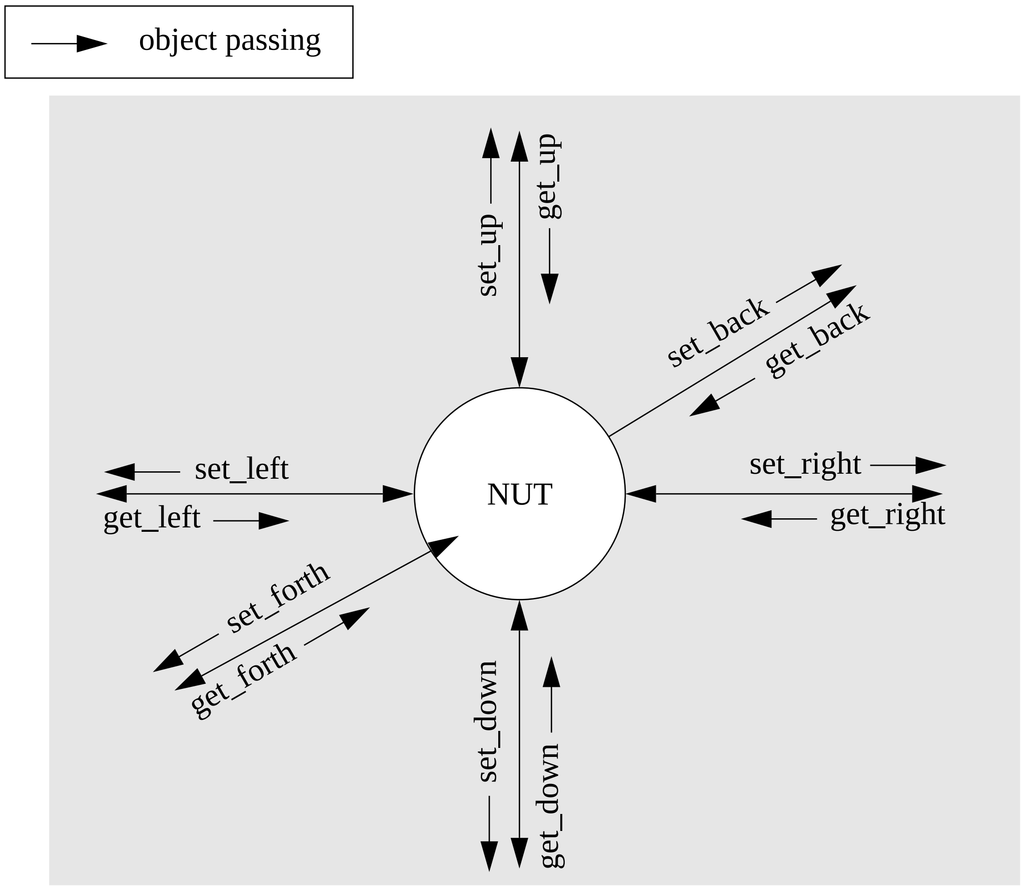


FIGURE 8 Communication pattern between application classes in `PARDIF`

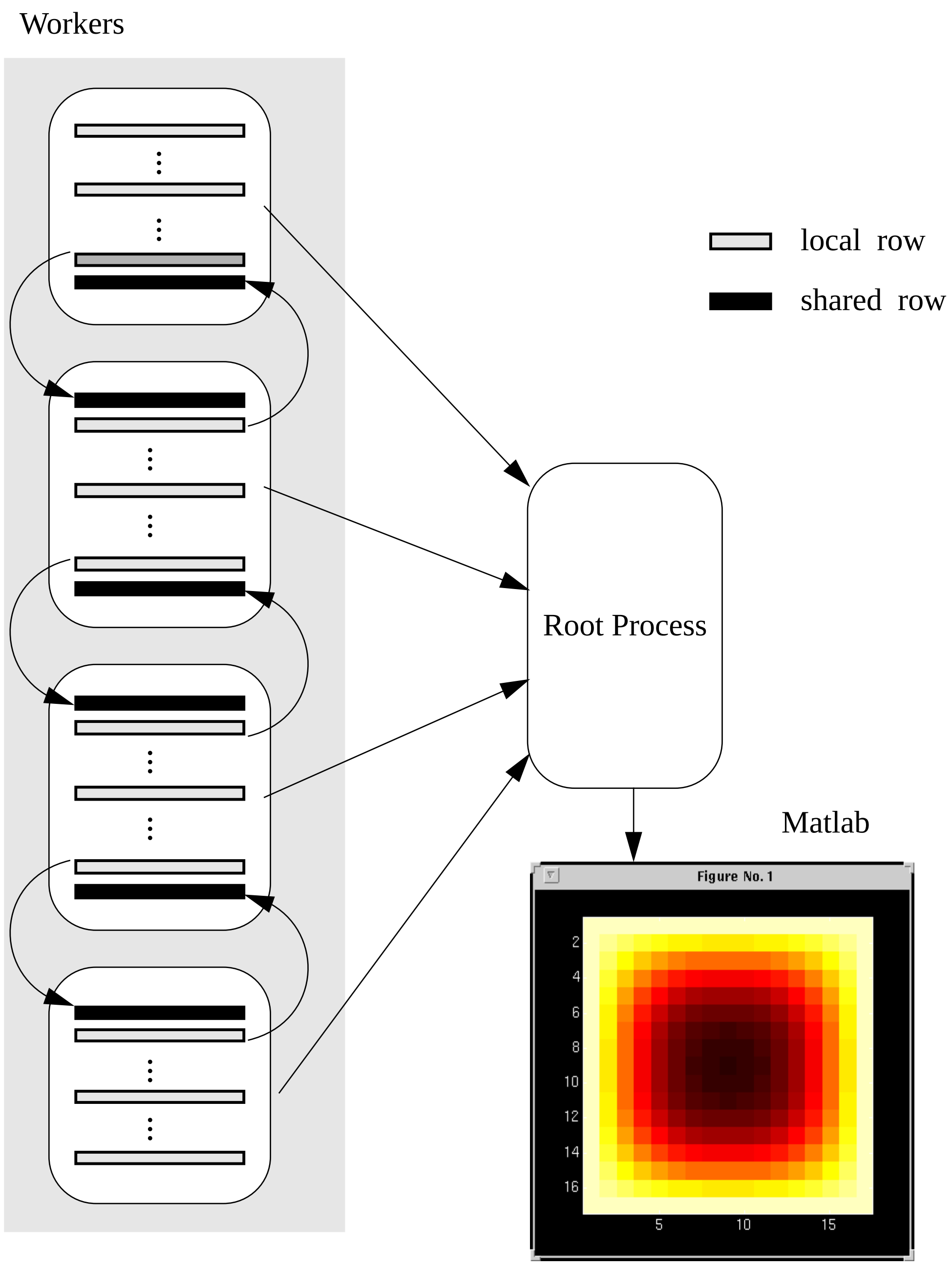


FIGURE 9 Data communication scheme of the Dirichlet problem