Distributed and Parallel Application Toolbox (DP Toolbox) for use with MATLAB(R) Version 1.7

R. Fink, S. Pawletta
Department of Electrical Engineering, University of Wismar, Germany
Department of Mechanical Engineering, University of Wismar, Germany
Institute of AutomaticControl, University of Rostock, Germany

Contact: r.fink@et.hs-wismar.de, s.pawletta@et.hs-wismar.de

February 2005

Abstract

This is a draft of the DP Toolbox user's guide and reference manual[1]. It contains an overview of the DP Toolbox, and how the public domain distribution can be obtained, installed and used.

DP Toolbox stands for Distributed and Parallel Application Toolbox. It realizes a multi instance approach to support a convenient development of distributed and parallel MATLAB applications. The current design of the included high-level interface is a proposal for a new approach to distributed and parallel processing in interactive environments. Therefore, it is a subject to discussion and change.

[1] **Please note:** We are not English natives, and our time budget to support the public domain release of the DP Toolbox is very limited. Therefore, this documentation is written in a non-reviewed English. Correction reports are very appreciated.

Contents

Chapter 1

DP Toolbox

1.1 Introduction

Starting in 1995, there were no parallel versions of MATLAB or similar packages available for practical use. Moler states the major reasons for this fact in [2] which are:

- The granularity (ratio of necessary computation operations to communication operations) of most MATLAB routines is to small for an effective parallelization on multi-processing systems without shared memory (e.g. clusters of networked PCs or workstations).
- On shared memory systems positive parallelization results can be reached, but the possible speedup is very limited because of the small number of processors of such systems (mostly less than a dozen).
- The portation of the sophisticated memory model of the sequential MATLAB version to multi-processing systems is very difficult.
- Under commercial aspects the effort for developing and supporting parallel MATLAB versions is too high in view of the limited number of customers with parallel machines.

That are serious technical and commercial aspects against a parallelization of a system like MATLAB. Substantially better prerequisites for distributed and parallel processing exist at the application level:

- Many scientific and engineering problems are characterized by a considerable run-time expenditure on one side and a medium- or high-grained logical problem structure (e.g. monte-carlo studies, complex simulations, evolutionary optimizations) on the other side. Such problems can be parallelized effectively even on networked computers.
- Because the parallelization and distribution of applications can be done on conventional MATLAB instances (i.e. sequential ones) no effort for developing and supporting special parallel versions is necessary.

To use multiple MATLAB instances for parallel and distributed processing they must be able to interact among one another. The necessary functionality for that is provided by the DP-Toolbox.

The DP-Toolbox for MATLAB is an example realization of the general *Multi-SCE Concept*, which is described in detail in [3]. This approach brings together the advantages of SCEs[1] (interactive way of working, array-oriented programming, rapid prototyping) and of parallel and distributed processing. The Multi-SCE Approach and the DP-Toolbox are the results of research activities in the field of parallel and distributed processing based on interactive environments at the Institute of Automatic Control, University of Rostock, the Department of Mechanical Engineering, University of Wismar and the Department of Electrical Engineering/Computer Science, University of Wismar since 1992.

Figure 1.1 shows how the MATLAB architecture is extended by the DP-Toolbox. Due to the added communication module MATLAB instances are able to pass messages among on another and to other programs. The DP-Toolbox by itself does not implement primitives for process communication and control. Instead an external message-passing systems is used. The current toolbox versions use the message-passing system PVM (*Parallel Virtual Machine*; [1]). Because PVM is available for many different operating systems and hardware platforms, MATLAB instances can be coupled via the DP-Toolbox also on heterogenous computer clusters and parallel computers.

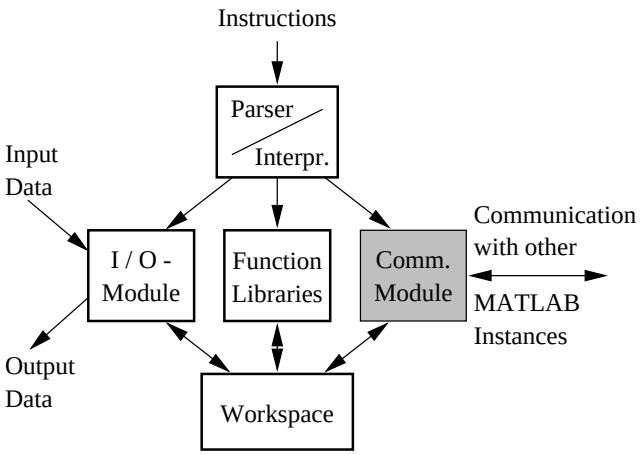

Figure 1.1: Extension of the MATLAB architecture

The DP-Toolbox is realized as toolbox set consisting of DPLOW- and DP(HIGH)-Toolbox (see fig. 1.2). Nevertheless, we still continue to use the term "DP-Toolbox" (or short: DP-TB) when we refer the toolbox set as a whole. The parentheses in the name of the individual toolbox DP(HIGH) should indicate, that this toolbox is the real core of the entire toolbox set and often seen *as the* DP-Toolbox. In that sense the DPLOW-Toolbox is a hidden auxiliary layer.

The DPLOW-Toolbox implements a MATLAB/PVM interface that provides the communication primitives of the PVM system in MATLAB. The major application field of this toolbox are couplings between MATLAB applications and other external programs. Additionally, it is very useful for educational purposes, because the DPLOW-Toolbox can be used as an "interactive PVM".

[1] *scientific and technical computing environments*; systems like MATLAB, Octave, Scilab etc.

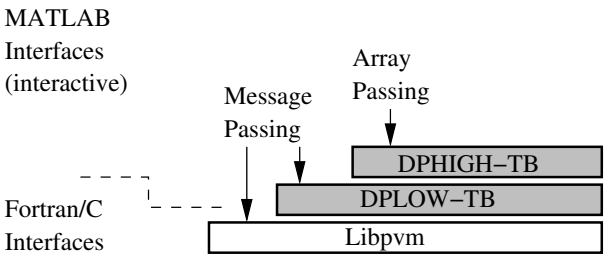

Figure 1.2: The DP Toolbox Set

For convenient couplings among MATLAB applications the abstraction level of the PVM routines is too low, because they are designed to meet the requirements of classical C or Fortran programming. For example data have to be packed before sending and unpacked after receiving. If complete MATLAB data objects (matrices, cell arrays etc) should be passed, all components have to be packed and unpacked separately, because PVM supports these operations only for primitive data types.

A suitable interface for developing distributed and parallel MATLAB applications is provided by the DP(HIGH)-Toolbox. Due to this interface MATLAB data objects can be sent and received directly (*array-passing*). The entire buffer management as well as the data packing and unpacking is done implicitly. Additionally, the DP(HIGH)-Toolbox contains routines for starting up and terminating MATLAB instances.

1.2 Getting and Installing the DP Toolbox

The current public domain releases of the DP-Toolbox version 1.7.0 work together with MATLAB 6.x and MATLAB 7.x, with PVM 3.4, and should run on most Unix platforms. See the release notes of the distribution for detailed informations.

Obtaining the DP Toolbox

The software and documentation can be downloaded from:

`http://www.mb.hs-wismar.de/cea/dp`

or can be requested by mail to:

`s.pawletta@et.hs-wismar.de`

Unpacking

Place the distribution file where the DP-Toolbox should reside (in a system disk area, e. g. `/usr/local/matlab/toolbox/`, or in a user private area, e. g. `$HOME/matlab/`) and unpack the distribution:

```
tar xvfz dp1.7.0.tgz
```

We recommend to create a symbolic link to the DP-TB directory:

```
ln -s dp1.7.0 dp
```

In this way it is easy to install new releases and keep old installations untouched by modifying only the symlink `dp`.

Building

To get the DP-Toolbox running, it is neccessary to compile the included MATLAB/PVM library. Fortunately, it consists only of one C file. All other files contained by DP-Toolbox are written in M-code.

Currently, there exists only a very simple Makefile for Gnu/Linux systems. You can easily adopt this Makefile by examining the help page for the mex commando (from shell):

```
mex -help
```

If you have an appropriate Makefile, the build process for DP-Toolbox is started with:

```
make
```

Installing

To make DP-Toolbox accessible from any path, you have to modify a special startup file, depending upon your DP-Toolbox installation. If you have made a system-wide installation (DP resides in e.g. `/usr/local/matlab/toolbox/dp`) then you have to edit the file `$MATLABROOT/toolbox/local/pathdef.m` by adding the following entry to the array `p`:

```
matlabroot,'/toolbox/dp:',...
```

If you have installed DP-Toolbox in your home directory, add the following line to your $HOME/matlab/startup.m file:

```
addpath /home/username/matlab/dp;
```

Of course, you can insert any other path depending upon where your DP package resides. For more detailed information about MATLAB path settings, refer to the page "Using the Path in Future Sessions" at the MATLAB help pages.

For more information about installing DP-Toolbox, look at the README file contained by the package.

Bibliography

[1] A. Geist et al. PVM 3 User's Guide and Reference Manual. Technical Report. Oak Ridge National Laboratory, May 1995.

[2] C. Moler. Why there isn't a parallel MATLAB. *MATLAB New and Notes*, page 12, Spring 1995.

[3] S. Pawletta. Extension of a Scientific and Technical Computation and Visualization System to a Development Environment for Parallel Applications. PhD Thesis, University of Rostock, June 1998. (in German)

Appendix A

DPLOW – User's Guide

A.1 Introduction

The DPLOW-Toolbox provides a set of interface functions to PVM routines. All these functions are prefixed with `pvm_` (i. e. they have the names of the original Libpvm3 routines). The design goal for these interface functions was to keep syntactical and semantical changes against the original Libpvm3 routines as small as possible. Consequently, their usage and behaviour are more like C functions than typical MATLAB commands(e. g. returning status codes instead of terminating immediately in case of errors, no variable input signature etc.).

The major application fields of the DPLOW-Toolbox are:

1. *Courses for parallel programming:*
   The DPLOW-Toolbox can be seen as an "interactive PVM". The interactive/interpretative way of working speeds up learning parallel programming significantly compared with traditional compiler based approaches (C or Fortran).

2. *Rapid prototyping of PVM based applications:*
   With the DPLOW-Toolbox applications can be developed and tested interactively in the MATLAB environment with a comparable minimal effort. After that, a verified application can be recoded strait forward in C or Fortran, because of the almost one-by-one semantics of the DPLOW-Toolbox functions to the original Libpvm3 and Libpvm3e routines.

3. *Parallel and distributed applications containing MATLAB, C and Fortran based components:*
   With DPLOW-Toolbox and Libpvm3, interfaces to PVM for MATLAB, C and Fortran are provided. Through these interfaces components coded in different languages can work together via the PVM system.

4. *Platform for building tools, which support parallel and distributed applications based on MATLAB components exclusively:*
   One example of such tool is the DP(HIGH)-Toolbox. The DPLOW-Toolbox by itself is not the right tool for developing pure MATLAB based parallel and distributed applications, because its abstraction level is too low to meet the productivity requirements of a "real" MATLAB user.

When we started to compile this user's guide the question arose: Should it be a guide without redundancy, which is understandable only for the experienced PVM user or only together with the original PVM documentation, respectively? Or should it be a complete, self-explanatory guide? We think the current guide is in between. for the beginner (without PVM knowledge) it should be possible to start with this guide, exclusively. The PVM expert will find a lot of redundant informations to the PVM documentation, but also the probably more interesting differences and extensions to the Libpvm3. Nevertheless, for understanding PVM in detail and for solving sophisticated problems you have to consult the original PVM documentation [1] and/or the PVM manual pages.

The typesetting conventions follow the usual standards known from MATLAB related documentations. We use a `typewriter font` to emphasize C or MATLAB functions. C function prototypes are preceded by `C:`, MATLAB function prototypes are preceded by `M:`.

Because parts of this user's guide are based on or derived from the original PVM documentation respectively, the following PVM copyright notice is stated:

A.2 Process Control

pvm_spawn

```
C: int numt = pvm_spawn(char *task,char **argv,int flag,
                        char *where,int ntask,int *tids)
```

```
M: [numt,tids] = pvm_spawn(task,argv,flag,where,ntask)
```

The routine `pvm_spawn` starts `ntask` copies of the executable file `task` on the virtual machine. `argv` is a cell array of strings containing the arguments to the task (each cell one argument). If the task takes no arguments then `argv` is an empty cell. The `flag` argument is used to specify options, and is a sum of:

`0` (`PvmTaskDefault`) PVM chooses where to spawn processes

`1` (`PvmTaskHost`) the `where` argument specifies a particular host to spawn on

`2` (`PvmTaskArch`) the `where` argument specifies a PVM_ARCH to spawn on

`4` (`PvmTaskDebug`) starts these processes up under debugger

`8` (`PvmTaskTrace`) the PVM calls in these processes will generate trace data

`16` (`PvmMppFront`) starts process up on MPP front-end

`32` (`PvmHostCompl`) starts process up on complement host set

On return `numt` is set to the number of tasks successfully spawned or an error code if no task could be started. If tasks were started, then `pvm_spawn` returns a vector of the spawned tasks' `tids` and if some tasks could not be started the corresponding error code are placed in the last $(ntask - numt)$ positions of the vector.

`pvm_spawn` returns immediately after spawning the tasks.

pvm_kill

```
C: int info = pvm_kill(int tid)
M: info = pvm_kill(tid)
```

The routine `pvm_kill` kills some other PVM tasks identified by `tid`. This routine is not designed to kill the calling task, which should be accomplished by calling `pvm_exit`.

pvm_exit

```
C: int info = pvm_exit()
M: info = pvm_exit
```

The routine `pvm_exit` tells the local pvmd that this process is leaving PVM.

A.3 Information

pvm_mytid

```
C: int tid = pvm_mytid()
M: tid = pvm_mytid
```

The routine `pvm_mytid` returns the task identifier of the caller.

pvm_parent

```
C: int tid = pvm_parent()
M: tid = pvm_parent
```

The routine `pvm_parent` returns the task identifier of the process that spawned the caller or the value `-23` (`PvmNoParent`) if it was not created by `pvm_spawn`.

A.4 Message Buffers

pvm_initsend

```
C: int bufid = pvm_initsend(int encoding)
M: bufid = pvm_initsend(encoding)
```

The routine `pvm_initsend` clears the send buffer and prepares it for packing a new message. The encoding scheme used for the packing is set by `encoding`. The buffer identifier is returned in `bufid`. The encoding options are:

> `0` (`PvmDataDefault`) XDR encoding is used by default because PVM can not know if the user is going to add a heterogeneous machine before this message is sent. If the user knows that the next message will only be sent to a machine that understands the native format, then he can use `PvmDataRaw` encoding and save on encoding costs.
>
> `1` (`PvmDataRaw`) no encoding is done. Messages are sent in there original format. If the receiving process can not read this format, then it will return an error during unpacking.
>
> `2` (`PvmDataInPlace`) data left in place. The message buffer only contains the sizes and pointers to the items to be sent. When `pvm_send` is called the items are copied directly out of the user's memory. This option decreases the number of times a message is copied at the expense of requiring the user to not modify the items between the time they are packed and the time they are sent.

A.5 Data Packing/Unpacking

pvm_pk*

```
C: int info = pvm_pkbyte(char *cp,int nitem,int stride)
   int info = pvm_pkint(int *np,int nitem,int stride )
   int info = pvm_pkdouble(double *dp,int nitem,int stride)
M: info = pvm_pkbyte(array,stride)
   info = pvm_pkint(array,stride)
   info = pvm_pkdouble(array,stride)
```

Currently, only routines for packing Byte, Integer and Double data types are implemented. In our opinion, packing of other data types can be established by using these three functions. The argument `array` specifies a MATLAB numerical array, where the argument `stride` works as the corresponding C function argument.

pvm_upk*

```
C: int info = pvm_upkbyte(char *cp,int nitem,int stride)
   int info = pvm_upkint(int *np,int nitem,int stride )
   int info = pvm_upkdouble(double *dp,int nitem,int stride)
M: [info,array] = pvm_upkbyte(nitem,stride)
   [info,array] = pvm_pkint(nitem,stride)
   [info,array] = pvm_pkdouble(nitem,stride)
```

Corresponding to the packing functions, only routines for unpacking Byte, Integer and Double data types exist. The argument `nitem` specifies the number of values to unpack, where the argument `stride` works as the corresponding C function argument. The output argument `array` contains the unpacked data in a MATLAB Double array.

A.6 Sending and Receiving Data

pvm_send

```
C: int info = pvm_send(int tid,int msgtag)
M: info = pvm_send(tid,msgtag)
```

The routine `pvm_send` sends a message stored in the active send buffer to the PVM process identified by `itid`. `msgtag` is used to label the content of the message.

pvm_recv

```
C: int bufid = pvm_recv(int tid, int msgtag)
M: bufid = pvm_recv(tid,msgtag)
```

The routine `pvm_recv` blocks the process until a message with label `msgtag` has arrived from `tid`. `pvm_recv` then places the message in a new active receive buffer, which also clears the current receive buffer.

Appendix B

DPLOW – Reference Guide

B.1 Quick Reference Tables

Process Control

Function	Description
pvm_spawn	Start new PVM process.
pvm_kill	Terminate PVM process.
pvm_exit	Leave PVM.

Information

Function	Description
pvm_mytid	Return tid of process.
pvm_parent	Return tid of parent process.

Message Buffers

Function	Description
pvm_initsend	Clear default send buffer and specify message encoding.

Packing / Unpacking Data

Function	Description
pvm_pkbyte	Pack data of type byte into active send buffer.
pvm_pkint	Pack data of type integer into active send buffer.
pvm_pkdouble	Pack data of type double into active send buffer.
pvm_upkbyte	Unpack data of type byte from active receive buffer.
pvm_upkint	Unpack data of type integer from active receive buffer.
pvm_upkdouble	Unpack data of type double from active receive buffer.

Sending and Receiving

Function	Description
pvm_send	Send data in active message buffer.
pvm_recv	Blocking receive.

Appendix C

DP(HIGH) – User's Guide

C.1 Introduction

As you have read probably before, the inconvenient term *DP(HIGH)* should indicate, that it stands for a set of functions which are on one side actually only one part of the *DP Toolbox Set*, but on the other side these functions can be seen as the real core of the entire toolbox set. In that sense and for better readability we will use the short term *DP-Toolbox* instead of DP(HIGH) throughout this user's guide.

The DP-Toolbox is realized on top of the DPLOW-Toolbox and supports a very simple model for parallel and distributed computing in MATLAB. The model is based on autonomous *DP instances*. Autonomous means that all actions and states inside a DP instance are caused exclusively by the instruction stream which is provided to the interpreter via the standard input (keyboard in case of an interactive session or file in batch mode). The only difference between a DP instance and an ordinary MATLAB instances is, that a DP instance is an enrolled process at the PVM system. With simple words: DP instances are able to communicate with the outside world - ordinary MATLAB instances not.

Beside the ability to communicate, DP instances can create new instances. Between the "spawner" and the spawned instance there exist a parent/child relation. Over and above that there are no further ordering relations in a set of DP instances. Therefore, no central or distributed control instance is known in the DP Model.

The two major objectives of the DP model are:

- bringing up the concept of message passing to a level that goes together with MATLAB (*array passing*), and
- hiding all the snares that occur when MATLAB processes (especially interactive ones) have to be launched.

The limits of the DP Model are:

- Due to the autonomy of DP instances only instructions that come in over the standard input stream can be processed. Therefore, the remote execution of MATLAB programs (M-scripts, M/MEX-functions, built-in functions) is not possible.
- The opportunity to pass a MATLAB program during starting up a MATLAB instance that executes the program immediately is only suitable for large applications in the

production phase. (because of the long startup times of MATLAB processes in the range of several seconds)

C.2 Connection to the Communication Subsystem

Presently, the DP-Toolbox uses as communication subsystem the external PVM system via the DPLOW-Toolbox. In the DP Model the communication subsystem is treated as a resource, that has to be present and already configured.

A connection to the PVM subsystem is established by calling a DP command the first time. After that, your DP instance becomes a PVM task. To disconnect from the subsystem, call `dpexit`.

C.3 DP-Instance Control

New DP instances can be started with the command `dpspawn`. On success `dpspawn` returns the unique identifier(s) of the new DP instance(s) (short: *dpid* or *dpids*; currently the dpids are the tids assigned by the PVM system). Optionally, `dpspawn` can process a number of input parameters with which certain properties of the spawned DP instance(s) can be controlled:

- where instances should be spawned
- the initial working directory of an instance
- instructions which should be executed after startup

For detailed informations see the online help or the reference guide.

Spawned DP instances can terminate in two ways. Either the MATLAB command `exit` (use `dpexit` before) is called inside the instance that should terminate, or the function `dpkill(dpids)` is used to kill one or multiple DP instances.

C.4 Information

The dpid of a DP instance can be determined by the function `dpmyid`. With the function `dpparent` it is possible to explore whether an instance has become a DP instance by itself, or if it has been spawned by another DP instance (with `dpspawn`).

If one of those functions return `-14`, MATLAB cannot connect to the PVM subsystem.

C.5 Communication

Communication with traditional message-passing systems (like PVM or MPI) can be characterised as follows:

- data that should be passed have to be packed explicitly and type-dependently to a *message* into a send buffer
- for passing a message it is put in an envelope that carries delivery informations (sender and receiver identifications, message tag, message context)

- many different semantical variants for receiving: non-selective (any message from any sender), selective (certain message tag, certain sender), blocking, non-blocking etc.; a received message is always provided in form of a receive buffer reference
- the date have to be unpacked explicitly and type-dependently from a message

Typically, the above functionality is provided by a dozen functions.

Communication in the DP Model is characterised as follows:

- Data in MATLAB are always of type *Array*. MATLAB by itself provides powerful methods for building compound data objects (again of type Array). Therefore, message-passing can be simplified in MATLAB to pure *array-passing*.
- arrays can be send directly (optionally with an array tag as delivery information)
- all semantical receiving variants are provided by a single function, that returns the received array directly

The above functionality is provided by two functions.

Any kind of array can be sent to one or several DP instances (including the sender) with `dpsend(array,dpids)`. If an array should be sent under a certain tag it can be provided as third parameter: `dpsend(array,dpids,tag)`.

With `dprecv` arrays can be received selective (certain senders and/or certain tags) or non-selective. For detailed informations about `dprecv` see the online help or the reference guide.

Beside passing complete arrays, the DP-Toolbox supports also scattering and gathering of arrays. Scattering of arrays is possible with `dpscatter(array,dpids,tag,dimension)`.

`dpscatter` divides the array along the specified dimension according to the number of destinations (dimension of dpids). That also works if the number of columns divided by the number of destinations gives a reminder. Scattered arrays may be received with `dprecv`.

The function `A = dpgather(dpids,tag,dimension)` receives an array from each source specified in `dpids` and builds them together into one array.

To test the functionality of `dpsend/dprecv` as well as `dpscatter/dpgather` functions two test applications are provided: `dp_sendtest` and `dp_scattertest`.

C.6 Application Examples

Three applications for the DP-Toolbox are provided as demonstration programs, see `dp_demo*`.

Appendix D

DP(HIGH) – Reference Guide

D.1 Quick Reference Tables

Connection to the Communication Subsystem

Function	Description
`dpexit`	Disconnect from PVM subsystem.

DP-Instance Control

Function	Description
`dpspawn`	Start new DP instance(s).
`dpkill`	Kill other DP instance(s).

Information

Function	Description
`dpmyid`	Return dpid of the calling DP instance.
`dpparent`	Return dpid of parent DP instance.

Communication

Function	Description
`dpsend`	Send Array.
`dprecv`	Receive Array.
`dpscatter`	Scatter Array.
`dpgather`	Gather Array.

Example Applications

Name	Description
`dp_demo1`	Perform dpsend and dprecv on one MATLAB instance.
`dp_demo2`	Perform dpspawn, dpsend and dprecv on local host.
`dp_demo3`	Performs a parallel vector multiplication (scalar product).
`dp_sendtest`	Tests dpsend/dprecv with several MATLAB data types.
`dp_scattertest`	Scatters a random matrix along three dimensions.