TR-529

FLIB User Manual

by

B. Burg & D. Dure

December, 1989

FLIB User Manual

Version 1.0 --- 11/11/89

Bernard Burg & Daniel Dure

Abstract:

The FLIB library is a set of modules which is provided to help people programming in KL1 on PDSS and on the Multi-PSI system. This manual describes in details the predicates found in these modules and give guidelines on such issues as synchronization, parallel programming, etc. Our point has been both to relieve the user from the tedium of rewriting trivial functions, and to underline some problems commonly encountered. While we do not display any magic solution for the problem of distributing an algorithm over a loosely coupled network of processors, we present some simple examples which, we hope it at the least, set out the typical trade-offs to operate during and after design of the program, and demonstrate that in the current environment, performance analysis is 99% experiment-driven.

概要：

本ライブラリ FLIB は、PDSS システムあるいは Multi-PSI システムの上で KL1 言語を用いてプログラムを書くユーザーにとって有用ないくつかのプログラムモジュールを提供するものである。各モジュール中の述語には詳細な説明を付け、また並列プログラミングにおける同期の問題などに関してガイドラインを与えた。わたしたちが狙いとしたのは、ユーザーが基本的な述語を最初から作り直す手間を省けるようにすること、そしてプログラムを書く際によく起こる代表的な問題点に関して説明することである。疎結合ネットワークによるマルティプルプロセッサーにおいて、与えられたアルゴリズムに対してプロセスをどう分散させるべきかという問題について万能な方法が提供できるわけではないが、このレポートでは少なくともいくつかの簡単な例を挙げてプログラム開発中または開発後に考えなければならない典型的なトレードオフの問題を切り出すことができたならば幸いである。またパーフォーマンスの分析は、現在の並列環境では 99% 実験的に進めて得られたものである。

Institute for New Generation Computer Technology

Fourth & Fifth Laboratory

Contents

Forewords 1

Acknowledgements 1

1 Introduction 2
- 1.1 Inside organization . . . 2
- 1.2 Words of advices . . . 3

2 Module fel 5
- 2.1 Virtual device open . . . 5
- 2.2 Strings . . . 5
- 2.3 Comparison . . . 6
- 2.4 Atoms . . . 6
- 2.5 Code . . . 6
- 2.6 Shoen . . . 6
- 2.7 Version . . . 6

3 Module list 7
- 3.1 Basic list operations . . . 7
- 3.2 Sublists . . . 8
- 3.3 Set operations . . . 9
- 3.4 Sorting . . . 9
- 3.5 Arithmetic . . . 10
- 3.6 Data conversion . . . 11
- 3.7 Version . . . 11

4 Matrix module 12
- 4.1 Type checking and creation . . . 12
- 4.2 Element access . . . 12
- 4.3 Basic matrix operations . . . 13
- 4.4 Data conversion . . . 13
- 4.5 Version . . . 13

5 Module par 14
- 5.1 Parallel processing . . . 14
- 5.2 Synchronization . . . 17
- 5.3 Version . . . 18

6 Module string 19
- 6.1 Basic string primitives . . . 19
- 6.2 Substrings . . . 19
- 6.3 Sorting . . . 19
- 6.4 Data conversion . . . 20
- 6.5 Version . . . 20

7 Module util 21
- 7.1 Integer to string and vice-versa . . . 21
- 7.2 General copy & compaction . . . 21
- 7.3 Parallel processing . . . 22
- 7.4 Synchronization . . . 22
- 7.5 Console output . . . 23
- 7.6 Enhanced timer & statistics . . . 23

7.7 Random number generator 24
7.8 Version 24

8 Module vect **25**
8.1 Basic vector primitives 25
8.2 Subvectors 26
8.3 Set operations 26
8.4 Sorting 26
8.5 Set arithmetic 27
8.6 Vector arithmetic 27
8.7 Data conversion 27
8.8 Version 28

9 Examples **29**
9.1 Search of a subvector in a vector 29
9.1.1 Sequential version 29
9.1.2 Naive parallel version 29
9.1.3 Final parallel version 30
9.1.4 Performance measurements 31
9.2 Sorting and removing doubles 32
9.2.1 Sequential version 32
9.2.2 First parallel version: Ver1 32
9.2.3 Second parallel version: Ver2 33
9.2.4 Performance measurements 34
9.3 Conclusion 35

Index

documentation (.dvi) ... 2
FLIB description ... 1
FLIB files ... 2
multiple references ... 3
sequentiality ... 3
version ... 2
.pdssrc ... 3
.sav ... 2
add/5 in **matrix** ... 13
andl/3 in **list** ... 10
andv/3 in **vect** ... 27
append/3 in **list** ... 7
append/3 in **vect** ... 25
appendstring/2 in **fel** ... 5
appendstring/3 in **fel** ... 5
apply_2_tree_of_p/12 in **par** ... 16
apply_list_of_p/8 in **par** ... 15
apply_list_of_p/8 in **util** ... 22
atom_to_name/2 in **fel** ... 6
atom_to_number/2 ... 6
bas_int_to_name/3 in **util** ... 21
bas_name_to_int/3 in **util** ... 21
char_to_ascii/2 in **fel** ... 5
comparator/5 in **list** ... 9
comparator/5 in **string** ... 19
comparator/5 in **vect** ... 26
copy/3 in **util** ... 22
create_2_tree_of_p/4 in **par** ... 16
create_list_of_p/4 in **par** ... 15
create_list_of_p/4 in **util** ... 22
dec_int_to_name/2 in **util** ... 21
dec_name_to_int/2 in **util** ... 21
fel module ... 5
findsubstr/4 in **string** ... 19
findsubstr_bm/4 in **string** ... 19
findsubvect/5 in **vect** ... 26
flash_copy/4 in **util** ... 22
flash_p_console/3 in **util** ... 23
get_code/5 in **fel** ... 6
hex_int_to_name/2 in **util** ... 21
hex_name_to_int/2 in **util** ... 21
inter/4 in **list** ... 9
inter/4 in **vect** ... 26
length/3 in **list** ... 7
list_to_string/3 in **list** ... 11
list_to_vector/2 in **list** ... 11
list module ... 7
Makefile ... 2
matrix/4 in **matrix** ... 12
matrix_col/4 in **matrix** ... 12
matrix_element/5 in **matrix** ... 12
matrix_line/4 in **matrix** ... 12
matrix_to_vector/2 in **matrix** ... 13
matrix module ... 12
maxi/3 in **fel** ... 6
maxi/3 in **list** ... 10
maxi/3 in **sync_list** ... 10
maxi/3 in **vect** ... 27
member/5 in **vect** ... 25
member/6 in **list** ... 8
mini/3 in **fel** ... 6
mini/3 in **list** ... 10
mini/3 in **vect** ... 27
mult/5 in **matrix** ... 13
name_to_atom/2 in **fel** ... 6
new_matrix/3 in **matrix** ... 12
nodoub/2 in **list** ... 10
nodoub/2 in **vect** ... 26
object_to_string/3 in **util** ... 22
open_file/4 in **fel** ... 5
open_window/3 in **fel** ... 5
orl/3 in **list** ... 10
orv/3 in **vect** ... 27
par module ... 14
product/3 in **list** ... 10
product/3 in **vect** ... 27
p_console/2 in **util** ... 23
quicksort_a/2 in **list** ... 9
quicksort_a/2 in **vect** ... 26
quicksort_d/2 in **vect** ... 26
random/2 in **util** ... 24
random_bound/4 in **util** ... 24
rec_list_to_vector/2 in **list** ... 11
rec_remove/4 in **list** ... 8
rec_remove/4 in **vect** ... 25
rec_reverse/2 in **list** ... 7
rec_reverse/2 in **vect** ... 25
rec_subst/5 in **list** ... 8
rec_subst/5 in **vect** ... 25
rec_vector_to_list/2 in **vect** ... 27
rel_get_count_start message ... 23
rel_get_count message ... 23
rel_on_at_start message ... 23
remove/4 in **list** ... 8
remove/4 in **vect** ... 25
req_red/2 in **util** ... 24
req_rel_red/2 in **util** ... 24
req_rel_time/2 in **util** ... 23

req_time/2 in util 23
reverse/2 in list 7
reverse/2 in string 19
reverse/2 in vect 25
setsublist/6 in list 8
setsubstring/4 in fel 5
setsubstring/7 in string 19
setsubvect/6 in vect 26
set_matrix_col/5 in matrix 12
set_matrix_element/6 in matrix 12
set_matrix_line/5 in matrix 12
shoen_execute/7 in fel 6
shoen_raise/3 in fel 6
simple_comparator/5 in list 9
sort_a/2 in list 9
sort_a/2 in vect 26
sort_d/2 in list 9
sort_d/2 in vect 26
split/4 in string 19
split/4 in vect 26
split/5 in list 8
start_stats/3 in util 23
string_to_list/2 in string 20
string_to_object/3 in util 22
string_to_vector/2 in string 20
string module 19
sub/5 in matrix 13
sublist/5 in list 8
subst/5 in list 7
subst/5 in vect 25
subvect/4 in vect 26
sub_string/5 in fel 5
sub_string/5 in string 19
suicksort_d/2 in list 9
sum/3 in list 10
sum/3 in vect 27
sync_andl/3 in list 10
sync_andv/3 in vect 27
sync_append/3 in list 7
sync_copy/3 in util 22
sync_create_2_tree_of_p/4 in par 16
sync_create_list_of_p/4 in par 15
sync_create_list_of_p/4 in util 22
sync_inter/4 in list 9
sync_length/3 in list 7
sync_member/6 in list 8
sync_mini/3 in list 10
sync_nodoub/2 in list 10
sync_orl/3 in list 10
sync_orv/3 in vect 27
sync_product/3 in list 10
sync_p_console/2 in util 23
sync_quicksort_a/2 in list 9
sync_quicksort_d/2 in list 9
sync_rec_list_to_vector/2 in list 11
sync_rec_remove/4 in list 8
sync_rec_remove/4 in vect 25
sync_rec_reverse/2 in list 7
sync_rec_reverse/2 in vect 25
sync_rec_subst/5 in list 8
sync_rec_subst/5 in vect 25
sync_rec_vector_to_list/2 in vect 27
sync_rec_wait/2 in util 22
sync_rec_wait_2_tree_of_p/2 in par 18
sync_rec_wait_list_of_p/2 in par 18
sync_rec_wait_list_of_p/2 in util 22
sync_remove/4 in list 8
sync_setsublist/6 in list 8
sync_sort_a/2 in list 9
sync_sort_d/2 in list 9
sync_split/5 in list 9
sync_string_to_list/2 in string 20
sync_sublist/5 in list 8
sync_subst/5 in list 7
sync_sum/3 in list 10
sync_union/3 in list 9
sync_vector_to_list/2 in vect 27
sync_wait/2 in util 22
sync_wait_2_tree_of_p/2 in par 18
sync_wait_list_of_p/2 in par 17
sync_wait_list_of_p/2 in util 22
timer/1 in fel 5
timer/1 in util 23
trace/3 in matrix 13
transpose/3 in matrix 13
union/3 in list 9
union/3 in vect 26
util module 21
vector_add/3 in vect 27
vector_scal_product/3 in vect 27
vector_sub/3 in vect 27
vector_to_col_matrix/2 in vect 28
vector_to_line_matrix/2 in vect 28
vector_to_list/2 in vect 27
vector_to_string/4 in vect 28
vect module 25
version/2 in fel 6
version/2 in list 11
version/2 in matrix 13
version/2 in par 18
version/2 in string 20
version/2 in util 24
version/2 in vect 28

Forewords

FLIB is a set of utility libraries, written in the KL1 language. It aims at improving various aspects of program development under PDSS and Multi-PSI, such as portability, synchronization of parallel programs, etc. It is by no mean a comprehensive library of functions, such as the ones commonly found in LISP systems, nor an interface to the environment, such as the UNIX libraries. Rather, we point here some specific problems with which we have been confronted during our development experiments. Therefore, some obviously useful functions were not introduced. On the other hand, we took care to program demonstratively, and to test both functionality and performance of these functions.

We dare think that the source code of these libraries can be usefully consulted by novice programmers. It gives useful clues as to practical realization of simple, low-level functions, and as to management of multi-processing. Of course, only the point of view of the authors is engaged here.

As a companion reading, we may advice our reader to order Technical Memorandum TM-0311 from ICOT, *A Collection of KL1 Programs*, by S. Takagi, 1987. Other documents can be consulted usefully, such as Technical Memorandum TM-0720 from ICOT, *PDSS Manual*, by K. Taki et S. Uchida[1], published in 1989, and the PIMOS user manual, which is not translated yet.

Acknowledgements

We would like to thank some people who helped us, either during the course of programming or debugging, or more directly, to write this document: Kawagishi Taro, Kimura Koichi and Yoshida Kaoru demonstrated an endless kindness to lighten the tedium of discovering PSI, Multi-PSI and PDSS. Inamura You and Sugino Eiji were always willing to help us when the system was seemingly in defect. During the initial translation of the PDSS user manual, much was due to Kohata Masaki and Wada Koichi, from Tsukuba University.

We are indeed forgetting many other helpful fellows. May them forgive us.

This work as been sponsored by INRIA, Rocquencourt, France through a grant and exchange program with ICOT, Tōkyō, Japan.

[1] **Besides, there had been 4 other fellow tranlators, including one of us.**

1 Introduction

This document describes in the large each library, and provides a list of the functions which can be found therein. So far, 7 modules have been developed:

fel This module forms a "stable" layer around system-related predicates and a few built-in predicates whose usage or precise syntax varies from PDSS to Multi-PSI. Its usage greatly relieves programmers from portability problems.

list This module shelters some predicates to manage or convert data organized into lists.

matrix This module introduces a format for matrices and provide some basic predicate to make computations on integer matrices, or convert them.

par This module holds predicates which can be used for synchronization or to spread computation over several processors, while taking care of duplicating data or setting up communication streams.

string This module holds predicates to manage, recognize and convert character strings.

util This module contains various utilities, to monitor activity, copy data, send messages on the console, etc.

vect Last of all, this module contains some predicates to manage or convert data organized into vectors.

Before the list of modules and the description of predicates they contain, the sequel of this section holds some comments on the library and its usage. After module description, the whole section 9 is dedicated to the detailed presentation of two programming examples.

Although source code of these modules is freely accessible, we would recommend not to modify it, except when a major flaw is encountered, the point being to avoid a crop of local versions and modifications. If something seems to be of general interest, please feel free to express your desiderata, and function will be added, as time allows it.

Version and history information are available through a separate predicate for each module. In case of problem or suspected bug, don't forget to give the release number. Although there is no guarantee of upward compatibility (there is none in PIMOS, basically, and we cannot do better than the host system at a reasonable cost), predicates will not disappear in the future. You are henceforth encouraged to use the latest version of this library. Comments can be sent to burg@icot.icot.jp or daniel@icot.icot.jp.

Have fun!

1.1 Inside organization

Two different organizations can be encountered, depending on the machine used: a UNIX system or a PSI. Originally, the library was developed on a UNIX system, and we used many facilities which are not available on SIMPOS, the PSI's system. Therefore, FLIB should be first installed on a UNIX system, at the least in order to get this documentation.

Organization is extremely simple: assuming that the directory containing FLIB is `./flib`, all user files (and makefile) are in directory `./flib` and all source files are in directory `./flib/src`. More precisely, directory `./flib` contains:

- `Makefile` which creates all the `.sav` files necessary to PDSS users and the documentation. Default targets are documentation and user files. To type `make` is enough to generate everything, assuming a pdss compiler is available as well as LaTeX.

- `intro.dvi` which is the result of a LaTeX compilation, and can be printed to produce the documentation you read now.

- `fel`, `list`, `matrix`, `par`, `string`, `util` and `vect`, all with extension `.sav`, which are the modules to be loaded by PDSS, and thus will be accessed by users. "chown" them accordingly.

- Also, `fel_util_test`, `list_test`, `matrix_test`, `par_test`, `string_test` and `vect_test`, all with extension `.sav`, which are the test modules for the FLIB library. They are not included in the default target of the makefile. You should type `make test` to generate them.

`flib/src` is the directory containing source files and some intermediary TEXfiles:

- pdss_fel.kl1 is the source of the module `fel` for PDSS. `mpsi_fel.kl1` is the source of the same module, on the Multi PSI. `list.kl1`, `matrix.kl1`, `par.kl1`, `string.kl1`, `util.kl1` and `vect.kl1` are the sources of the other modules.

- Files `*_test.kl1` are source files for test of user modules.

- `intro.tex` is the master TEXfile used to produce this documentation. `index.tex` is the index file, manually produced from a `.idx` file. File with suffix `.ps` are postscript files used to produce some of the figures in this document. If you don't have a postscript printer, that's too bad — replace these files with empty files. Other TEXfiles are generated by the makefile, and removed after the `.dvi` file has been produced.

Note that access to the user modules can be made fairly easy to users if the proper path is set. To this end, the following line should be added in the file `.pdssrc`, in the home directory of each user:

```
getenv(loaddir, L), setenv(loaddir,["path/flib" | L]).
```

where *path* is the file path leading to `./flib`.

The situation is more complicated on Multi-PSI, where no makefile facility is provided. It is suggested to dedicate a machine as a server. On this machine, some user, say `flib`, holds a PIMOS kernel with all the flib files linked. Be careful to compile `mpsi_fel.kl1` and not `pdss_fel.kl1`. Then, user can download the kernel via either ftp or psinet. If no net is available, since kernel file `.kbn` is large, users have no choice but copying the source files on a floppy, and compiling on their own machine.

If a definitive release mechanism is set up, these problems should ease.

1.2 Words of advices

Some basic rules have been followed during the craft of this library. We are conscious they can be argued, but let's summarize first a few points that motivated them:

- The first observation is that, due to the non-sequentiality of KL1, and quite independently from parallel execution, it is difficult to know at a given moment what happens in a program. As long as the program work, this is no major trouble as would say those people involved in semantics of KL1, but the point is that sometimes you have to debug your program...

- Then, again because of non-sequentiality, it may happen that the part which is executing tail-recursively in your program is not the one you'd like. This is especially excrutiating when you use vectors, which can be referred even if their components are not bound, or I/O which are slow and whose priority cannot be totally controlled. If such a case occurs, you may see much dynamic code generated, and the overall program speed can decrease much. Also, memory usage increases, due to the stacking of goals, and in the worst case, i.e. breadth-first exploration of the program execution tree, you will quickly run out of core.

- Multiple references to complex objects, especially vectors or lists, make the garbage collector load more heavy, and are likely to cause problems during multi-processing, as no more than a few objects can be referred by several processors at the same time.

This has contributed to a somewhat paranoiac view of KL1 programming, in which synchronization is emphasized, and multiple references are banned. To this end, the following rules have been applied:

- Whenever possible, i.e. when this is not demanding much extra code nor extra computing time, write predicates returning a copy of their arguments, when they are not atomic. This will avoid multiple references, and that will naturally allow sequential execution of tasks using the same data.

- Adopt a consistent layout of predicate arguments. In our case, inputs are on the left, output on the right (in the documentation, outputs are following the caret sign ^), and copies of arguments are put next to them.

- Make a clear distinction between predicates which work in an *asynchronous* manner, i.e. which start producing results as structured arguments are progressively bound, and synchronous predicates, i.e. which return values only after all work has been done and all structured arguments are internally bound. In our case, we put the prefix `_sync` to the name of predicates, when doubt was possible, or both synchronous and asynchronous predicates exist.

- Consider synchronization of your program, i.e. forcing sequentiality, from an early stage, or be ready to consider major rewriting as soon as: you debug, you decide to make precise timing measurements of inside work or your parallel implementation does not exhibit expected performances or behavior.

Let us emphasize that these rules can be subject to debate, and reflect the point of view of the authors only.

2 Module fel

The predicates in this section have been created to improve portability of KL1 programs between PDSS and Multi-PSI systems. Some of them are already obsolete, as both systems slowly but hopefully converge. They are kept for the sake of compatibility with previous versions.

Predicates have been arranged according to the following topics:

- virtual device open;
- strings;
- comparison;
- atom;
- code;
- shoen;
- version.

2.1 Virtual device open

The following predicates provide device protocols which are similar to the latest version of the PIMOS protocol.

- `fel:open_window(Stream, Name, ^Status)`
  Creates a window with `Stream` as a command stream. `Name` should be a default type string. It is used to name the window. Status is unified with atom `normal` in case of success, or atom `abnormal` otherwise. `show` and `activate` commands are implicit.

- `fel:open_file(Stream, Filename, Mode, ^Status)`
  Opens a file with name `Filename`, which should be a default type string. Access mode is specified by `Mode`, chosen among atoms `r` (read), `w` (write) or `a` (append). `Stream` is the command stream. Status is unified with atom `normal` in case of success, or atom `abnormal` otherwise.

- `fel:timer(Stream)`
  Creates a timer, with command stream `Stream`.

2.2 Strings

- `fel:appendstring(S1, S2, ^R)`
  Like `append_string/3` on PDSS, or `builtin#append_string/3` on Multi-PSI.

- `fel:appendstring([S1, S2, ...], ^R)`
  Like `builtin#append_string/2` on Multi-PSI.

- `fel:sub_string(String, Start, Length, ^Sub, ^Copy)`
  Like `substring/5` on PDSS or `builtin#substring/5` on Multi-PSI.

- `fel:setsubstring(String, Start, Replace, ^Result)`
  Like `set_substring/4` on PDSS or `builtin#set_substring/4` on Multi-PSI.

- `fel:char_to_ascii(Char,^Ascii)`
  Transforms a character `Char` in the default representation code (JIS or ASCII) into its ASCII equivalent. Result is unified with `Ascii`.

2.3 Comparison

- `fel:mini(A, B, ^Min)`
  Like `min/3` on PDSS.

- `fel:maxi(A, B, ^Max)`
  Like `max/3` on PDSS.

2.4 Atoms

- `fel:atom_to_name(Atom, ^Name)`
  Like `atom_name/2` on PDSS or `atom_table:get_atom/3` on Multi-PSI.

- `fel:name_to_atom(Name, ^Atom)`
  Like `intern_atom/2` (with reversed arguments) on PDSS or `atom_table:intern/2` on Multi-PSI.

- `fel:atom_to_number(Atom, ^Number)`
  Like `atom_number/2` on PDSS or `hash/3` on Multi-PSI.

It's too bad that there's no "`number_to_atom`"-like facility available.

2.5 Code

- `fel:get_code(Module, Predicate, Arity, ^Code, ^Status)`
  Like `get_code` message of module table protocol, on Multi-PSI. `Status` is unified with atom `normal` if all is ok, `abnormal` otherwise.

2.6 Shoen

- `fel:shoen_execute(Code, Argv, MinP, MaxP, ExcpM, Ctl, ^Rpt)`
  Like `execute/7` on PDSS or `shoen:execute/7` on Multi-PSI. Messages sent in report stream are system dependent, except `raise`, which has been made compatible with the latest PIMOS spec, and `statistics`, which has been kept compatible with the old PIMOS format (a single integer for the statistics count).

- `fel:shoen_raise(Tag, Info, Data)`
  Like `raise/3` on PDSS or `shoen:raise/3` on Multi-PSI.

2.7 Version

- `fel:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

3 Module list

This module contains predicates to help management of data organized as lists. Curiously, whereas this data structure is highly suited to the representation of graphs, tree structures and dynamic data structures, little support is found for it in KL1. Besides, in the current implementation of KL1 on the Multi-PSI, the transfer of lists is done element by element, which makes it very unsuited to exchange of large chunks of data, for which vectors, or even strings are to be preferred. I'm sorry to say that nothing indicates that this would be about to change soon.

In the following, predicates are arranged according to the following topics:

- basic list operations;
- sublists operations;
- set operations;
- sorting;
- arithmetic;
- data conversion;
- version.

3.1 Basic list operations

- `list:length(List,^Copy,^Rinteg)`
  `Copy` is unified with the original list `List` after the number of elements in this list has been unified with `Rinteg`. Note that list elements don't need to be bound.

- `list:sync_length(List,^Copy,^Rinteg)`
  As above, but `Copy` is bound with a copy of `List` after the length has been computed.

- `list:append(ListA,ListB,^RList)`
  `RList` is unified with `ListA` and `ListB` is appended at the end of this list.

- `list:sync_append(ListA,ListB,^RList)`
  This works as the previous predicate, but `RList` is bound *after* the end of the append, i.e. when the end of `ListA` is encountered. Note that list elements don't need to be bound.

- `list:reverse(List,^RList)`
  `List` is reversed at the top level, and the result is unified with `RList`. Note that list elements don't need to be bound.

- `list:rec_reverse(List,^RList)`
  As above, but `List` is reversed at *all* levels, recursively.

- `list:sync_rec_reverse(List,^RList)`
  As the previous predicate, but `RList` is bound only when all sub-lists have been reversed. Note that list elements don't need to be bound.

- `list:subst(List,Old,^Copy,New,^RList)`
  All occurrences of `Old` in `List` at the top level are replaced by `New`. The result is unified with `RList`. `Copy` is a copy of `Old`. Note that if `New` is not atomic, multiple references will occur. Note that `Old` is not necessarily atomic.

- `list:sync_subst(List,Old,^Copy,New,^RList)`
  Same as above, but `RList` is bound only when all substitutions are done and `List` has been closed.

- `list:rec_subst(List,Old,^Copy,New,^RList)`
  All occurrences of `Old` in `List` at all levels are replaced by `New`. The result is unified with `RList`. `Copy` is a copy of `Old`. Note that if `New` is not atomic, multiple references will occur. Note that `Old` is not necessarily atomic.

- `list:sync_rec_subst(List,Old,^Copy,New,^RList)`
  Same as above, but `RList` is bound only when all substitutions are done and `List` and its sublists have been closed.

- `list:remove(List,Elem,^Copy,^RList)`
  Remove all occurrences of `Elem` in `List`, at the top level, and puts the result in `RList`. `Copy` is a copy of `Elem`. Note that `Elem` is not necessarily atomic.

- `list:sync_remove(List,Elem,^Copy,^RList)`
  Same as above, but `RList` is bounded when all suppressions have been done.

- `list:rec_remove(List,Elem,^Copy,^RList)`
  Remove all occurrences of `Elem` in `List`, at the top level, and puts the result in `RList`. `Copy` is a copy of `Elem`. Note that `Elem` is not necessarily atomic.

- `list:sync_rec_remove(List,Elem,^Copy,^RList)`
  Same as above, but `RList` is bounded when all suppressions have been done.

- `list:member(List,Elem,^Copy,^Before,Ebefore,^From)`
  Finds the first occurrence of `Elem` in `List`. `Before` is the sublist of `List` ending at this point. `Ebefore` is the end of this list. `From` is the remainder of `List`, starting from the first occurrence of `Elem`. If there is no occurrence of `Elem` in the list, `From` points to `[]`, and this predicate works like `append`. `Copy` is a copy of `Elem`. Note that `Elem` is not necessarily atomic.

- `list:sync_member(List,Elem,^Copy,^Before,Ebefore,^From)`
  As above, but `Before` and `From` are bound when `Elem` or the end of the list is found.

3.2 Sublists

- `list:sublist(List,Start,Lng,^RList,ERList)`
  Extracts a sublist of length `Lng`, from position `Start`, starting at 0. `ERList` points to the end of the extracted sub-list. Position of the first element is zero. `List` is completed on both sides with a virtual void list. Hence, if `Start` or `Start+Lng` are out of the list, intersection with the actual list is returned in `RList`.

- `list:sync_sublist(List,Start,Lng,^RList,ERList)`
  As above, but `RList` is bound only after the sublist has been closed. Note that list elements don't need to be bound.

- `list:setsublist(List,Start,Lng,^OldList,InsList,^RList)`
  Replaces in `List` the sublist of length `Lng` from position `Start`, starting at 0, with the new list `InsList`. if `Start` or `Lng` arguments are out of the boundaries of `List`, insertion occurs at the nearest boundary. `OldList` is unified with the replaced sublist.

- `list:sync_setsublist(List,Start,Lng,^OldList,InsList,^RList)`
  As above, but `RList` and `OldList` are bound only after the sublist has been closed. Note that list elements don't need to be bound.

- `list:split(List,Pos,^Before,EBefore,^From)`
  Splits `List` after the element at position `Pos`, starting from 0, `Before` is a pointer to the first list, `EBefore` is a pointer to the end of the first list and `From` points to the second list.

- `list:sync_split(List,Pos,^Before,EBefore,^From)`
  As above, but **Before** is bound only when splitting has been done. Note that list elements don't need to be bound.

3.3 Set operations

A set is basically a list of elements, in which there should not be any duplicated element, unless explicitly mentioned.

- `list:union(ListA,ListB,^RList)`
  The set union of **ListA** and **ListB** is put into **RList**. The same element can appear several times in ListB without changing the result.

- `list:sync_union(ListA,ListB,^RList)`
  As above, but **RList** is bound when closed, i.e. when union is over.

- `list:inter(ListA,ListB,^CopyListB,^RList)`
  The set intersection of **ListA** and **ListB** is put into **RList**. Elements in the result are taken from **ListA**. Consequently, the same element can appear several times in **ListB** without changing the result. CopyListB is a copy of **ListB**.

- `list:sync_inter(ListA,ListB,^CopyListB,^RList)`
  As above, but **RList** is bound when intersection is over.

3.4 Sorting

In the following predicates, unless user predicate is used for the purpose of comparison, sorting order is implementation dependent, especially between objects of different types. The point of these predicates is mainly to eliminate identical elements, or to provide quick access.

- `list:simple_comparator(A,B,^S,^L,^Swapped)`
  This predicate can be used for atoms or integers only. It compares **A** and **B**, then unifies the smaller element with **S** and larger with **L**. If **B** is strictly less than **A**, **Swapped** is unified with the atom **yes**, and otherwise with no.

- `list:comparator(A,B,^S,^L,^Swapped)`
  As above, but it can be used to compare lists of containing objects of arbitrary type.

- `list:sort_a(List,^RList)`
  `list:sort_d(List,^RList)`
  This predicate sorts **List**. **sort_a** returns an increasing list in **RList**, and **sort_d** a decreasing one, for integers. For other data types, this function provides an arbitrary order, according to the comparator:sort operation. The obtained order is a total order though.

- `list:sync_sort_a(List,^RList)`
  `list:sync_sort_d(List,^RList)`
  As above, but **RList** is bound when the sort is over.

- `list:quicksort_a(List,^RList)`
  `list:quicksort_d(List,^RList)`
  This is similar to **sort_a** and **sort_d**, but to gain speed it is assumed that all elements in **List** have the same type, which is actually checked for the first element only. This causes a speedup of 2 to 3.

- `list:sync_quicksort_a(List,^RList)`
  `list:sync_quicksort_d(List,^RList)`
  As above, but **RList** is bound when the sort is over.

- `list:nodoub(List,^RList)`
  Filters `List` into `RList`, such that all consecutive identical elements are reduced to a single occurrence. Its use, consequently to a sort, provides a simple way to remove multiple instances of the same object in a list.

- `list:sync_nodoub(List,^RList)`
  `RList` is bound after operations have been finished.

3.5 Arithmetic

- `list:mini(List,^CopyList,^Mini)`
  `Mini` is unified with the minimum element of `List`, which is assumed to hold integers. `CopyList` is a copy of `List`.

- `list:sync_mini(List,^CopyList,^Mini)`
  As above, but `CopyList` is bound only when minimum has been computed.

- `list:maxi(List,^CopyList,^Maxi)`
  As `mini`, but we get the maximum element, instead of the minimum.

- `list:sync_maxi(List,^CopyList,^Maxi)`
  As above, but `CopyList` is bound only when maximum has been computed.

- `list:sum(List,^CopyList,^Sum)`
  This predicate unifies `Sum` with the sum of all integers in `List`. `CopyList` is a copy of the initial `List`.

- `list:sync_sum(List,^CopyList,^Sum)`
  As above, but `CopyList` is bound only when sum has been computed.

- `list:product(List,^CopyList,^Prod)`
  As `sum`, but we get the product instead of the sum.

- `list:sync_product(List,^CopyList,^Prod)`
  As above, but `CopyList` is bound only when product has been computed.

- `list:andl(List,^CopyList,^Result)`
  This predicate operates the logic and of a list of booleans : if any element in `List` is zero, `Result` is unified with 0, otherwise it is unified with 1 (notably if the list is empty). The list is scanned in usual order, but asynchronous binding is supported. For example, the list `[_|0]` will produce a 0. `CopyList` is a copy of `List`.

- `list:sync_andl(List,^CopyList,^Result)`
  This works the same as above, but all terms before 0 or the end of the list must be bound. The scan is over before `Result` is bound.

- `list:orl(List,^CopyList,^Result)`
  This predicate is similar to `andl`, but 1 is returned if any 1 is in `List`, and 0 is returned otherwise, notably if the list is empty.

- `list:sync_orl(List,^CopyList,^Result)`
  This works the same as above, but all terms before 1 or the end of the list must be bound. The scan is over before `Result` is bound.

3.6 Data conversion

- `list:list_to_string(List,Type,^String)`
  `String` is unified with a character string, with **`Type`** bits per character, mapping the list of integer `List`.

- `list:list_to_vector(List,^Vector)`
  `Vector` is unified with a vector mapping `List`, at the top level.

- `list:rec_list_to_vector(List,^Vector)`
  Same as above, but mapping is done at all list levels.

- `list:sync_rec_list_to_vector(List,^Vector)`
  This is the same as above, but `Vector` is bound only when all mappings are over, i.e. `List` and its sublists are closed.

3.7 Version

- `list:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

4 Matrix module

A 2-dimensional matrix of *integers* is represented as a vector of vector. Each element in the first vector corresponds with a column. First element in a column or a row is denoted by 0.

Predicates of this module are arranged according to the following topics :

- type checking and creation;
- access to elements;
- basic matrix operations;
- data conversion;
- version.

All the following predicates are synchronous w.r.t. the elements of the source matrix they access.

We agree that matrices of integers are of limited use. Quite recently, floating points have been introduced in KL1, but their usage requires syntactic sugar — there is little room for operator overloading in a language in which there is no type. Considering the amount of dynamic checks performed by the KL1 microcode, it would have been possible to perform type checking during execution, thus relieving the programmer from inserting $ signs whenever he wants to perform a floating point operation.
In the current state of achievement of KL1, we can consider floating points as a kludgy wart. It has been deliberately chosen not to use them in this library.

4.1 Type checking and creation

- **`matrix:matrix(Mat,^CopyMat,^Row,^Col)`**
  `Row` and `Col` are unified with the number of rows and columns in matrix `Mat`, whose copy is returned through `CopyMat`.

- **`matrix:new_matrix(^Mat,Row,Col)`**
  `Mat` is unified with a new matrix, with `Row` rows and `Col` columns, filled with 0.

4.2 Element access

- **`matrix:matrix_element(Mat,Row,Col,^Elem,^CopyMat)`**
  `Elem` is unified with the element in the `Row`-th row and `Col`-th column of matrix `Mat`, whose copy is returned via `CopyMat`.

- **`matrix:set_matrix_element(Mat,Row,Col,^OldValue,NewValue,^NewMat)`**
  Sets the value of element at row `Row` and column `Column` to `NewValue`. `OldValue` is unified with the old value of this element, and `CopyMat` points to the new matrix.

- **`matrix:matrix_col(Mat,Col,^ColumnVector,^CopyMat)`**
  In the matrix `Mat`, makes a copy of the column `Col` in `ColumnVector`. A copy of the matrix is unified with `CopyMat`.

- **`matrix:set_matrix_col(Mat,Col,^OldColumnVector,NewColumnVector,^NewMat)`**
  In matrix `Mat`, set column number `Col` with `NewColumnVector`, which is supposed to be a vector of the right length, and unifies the old vector with `OldColumnVector`. The new matrix thus constructed is returned in `NewMat`.

- **`matrix:matrix_line(Mat,LineNb,^LineVector,^CopyMat)`**
  This is analogous to predicate `matrix_col`, for a line instead of a column.

- **`matrix:set_matrix_line(Mat,LineNb,^OldLineVector,NewLineVector,^NewMat)`**
  This is analogous to predicate `set_matrix_col`, for a line instead of a column.

4.3 Basic matrix operations

- `matrix:add(Mat1,^CopyMat1,Mat2,^CopyMat2,^Rmat)`
  The result of the addition of matrix `Mat1` and `Mat2` is unified with `RMat`. Copies of the original matrices can be found in the argument next to the input matrices. Note that matrices are supposed to be of the same size.

- `matrix:sub(Mat1,^CopyMat1,Mat2,^CopyMat2,^Rmat)`
  This is similar to the previous predicate, besides the operation, which is substraction instead of sum.

- `matrix:mult(Mat1,^CopyMat1,Mat2,^CopyMat2,^Rmat)`
  This is similar to the previous predicate, besides the operations, which is a product, and the constraint on dimensions, which becomes the one common to matrix product.

- `matrix:transpose(Mat,^CopyMat,^Tmat)`
  This predicate unifies `Tmat` with the transposed matrix `Mat`. A copy of the original matrix can be found in `CopyMat`.

- `matrix:trace(Mat,^CopyMat,^Trace)`
  `Trace` is unified with the trace of `Mat`, which should be a square matrix. A copy of the matrix can be found in `CopyMat`.

4.4 Data conversion

- `matrix:matrix_to_vector(Mat,^Vect)`
  Transforms `Mat`, a matrix with a single line or column, into a vector, unified with `Vect`.

4.5 Version

- `matrix:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

5 Module par

The name of this module is derived from the word "parallel", as some of the predicates therein may help achieving the high deed of using several processors on the Multi-PSI.

The specification of some of these predicates is a little bit hairy, and we would advice our puzzled reader to have a look on the examples or into the actual code. We shall improve the text according to requests...

The following predicates are organized according to the following topics :

- parallel processing;
- synchronization;
- version.

Some of the predicates therein were previously featuring in the module `util`. For the sake of compatibility, references are still present in the later, but actual code is in file `./flib/src/par.kl1`.

5.1 Parallel processing

The primitive proposed here help the user to create 2 kinds of distributed data structures — list and tree — then to apply a user predicate to those, in a parallel fashion.

In the most general view of the problem, several points have to be addressed: communications between processors, broadcasting of data used by several processors, etc. We followed a rather simplistic approach, in which communications are achieved along edges of the data structure, or in a global fashion.

- In the case of list, as illustrated in figure 1, global communication exists prior to actual computation. User-supplied data are broadcast to all processors from a single processor. Results of the computation can then be collated globally.

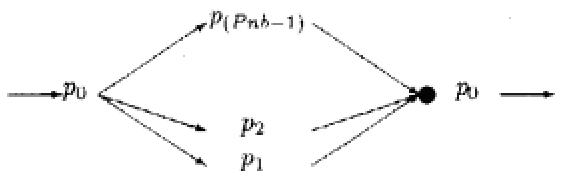

Figure 1: Spreading and collation in a list structure

- In the case of tree data structure, as illustrated in figure 2, data are spread from each node of the tree to each children. Accordingly, results of computation can be collated in a hierarchical manner.

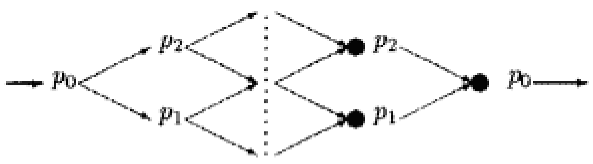

Figure 2: Spreading and collation in a tree structure

It's worth nothing that besides the initial duplication of user data, communication patterns of the list are kept for the tree data structure, which actually supports both types of communication.

- `par:create_list_of_p(Start, End, Data, ^List)`
  This predicate creates a list **List**, containing (**End** − **Start** + 1) pairs of elements. Each "pair" is made of a processor number and data which are local to this processor. Processor numbers are consecutive, from **Start** to **End**, and local data are copied from **Data**, which can be any complex or atomic object, recursively bound. **Data** should not contain any object of *code* type. The time required to complete this operation grows linearly with the number of processors, times the size of **Data**.

- `par:sync_create_list_of_p(Start, End, Data, ^List)`
  Same as above, but **List** is bound only when all copies are done, on all processors. Same remark as above for the processing time.

There is clearly a trade-off here, for the matter of copying data from one processor to another. We said that the computing time was growing like the number of processors times the size of data. The reason is that we copy the whole data structure on a processor before going to the next processor. This could be improved, by copying part of the data structures on the local processor, then transferring to the next processor, which can in turn start the copy, while we keep on with the remaining of the data structure. That way, i.e. using asynchronous copy, the total computing time would be the *sum* of the transfer time of the last sub-data along the whole processors chain, which grows linearly with the number of processors, and of the transfer time of the remaining data between the last processor and the previous processor. The trade-off lies in that performing the copy that way would be slower, especially for vectors of constants, or strings. We did not complete an implementation of this scheme yet, but it would be interesting to compare the alternative.

- `par:apply_list_of_p(List, ^Copy, Mod, Pred, Args, ^MO, ^SO, SI)`
  This predicate can be used with a list **List**, created using one of the two previous predicates: it applies the user predicate identified using atoms **Mod**, which identifies its module, and **Pred**, which identifies the predicate, to the data in **List**, on relevant processors. The user predicate has arguments specified by the vector **Args**, which is copied from one processor to another. Therefore, all of its elements should be bound, recursively. This vector should have a correct length, as it is used to find the arity of the user predicate to call. Also, this predicate has to be declared public in the module **Mod**. The calculations performed by the user predicate can be put back into the processor list, or inserted in a stream which goes from one processor to the other, or inserted in a stream which is subject to a merge, for all processors. The (possibly) modified list is unified with **Copy**, and has a structure similar to the one of **List**; the "merged" stream coming from all processors is unified with **MO**; the stream which connects one processor to the other has two ends: if we put processors on a line, with increasing processor numbers from the left to the right, the stream is going from the right to the left. Therefore, **SO** is the stream going out from the processor with the smallest number, while **SI** can be used to feed data in the processor with the largest number.
  Access to the list or streams is achieved by putting special atoms in the argument vector, which will be replaced before starting the user predicate on each processor. More precisely:

  - '**processor_in**' will be replaced by the current processor number;
  - '**data_in**' will be replaced by the piece of data in **List** which corresponds to the current processor;
  - '**data_out**' will be replaced by a variable which should be set within the user predicate. This variable will be put back in the list, *in lieu* of the data which was (maybe) accessed via '**data_in**'. If neither '**data_in**' nor '**data_out**' is used, **List** will be left untouched, and **Copy** will be an actual copy of **List**. If '**data_in**' only is specified, an undef variable '_' will be inserted in the list. If '**data_out**' only is used, the original data in **List** will be garbaged.
  - '**merge_out**' will be replaced by a stream which should be filled by the user. This stream will be merged with the ones of the same kind from other processors. We recall that merged stream is available as **MO**. If this atom is not used, **MO** will be unified with [].
  - '**stream_out**' will be replaced by a stream which should be filled by the user. This stream is connected to the input stream of the previous processor in the list. That way, all results can

be retrieved sequentially from all processors. We recall that the output stream from the first processor is available as `SO`.

– '`stream_in`' will be conversely replaced by the output stream coming from the next processor in the list. The last processor's input stream is set to `SI`.

Note that the direction of the sequential streams linking processors is arbitrary. User is free to output things on the input stream, if more convenient. Also, if neither '`stream_in`' nor '`stream_out`' are used, `SI` and `SO` are unified together, through the whole processor chain.
Figure 3 pictures a list of 3 processors, with the connections and data available on them.

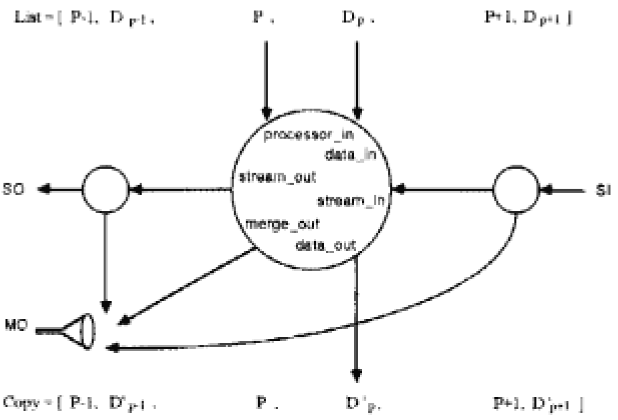

Figure 3: View of a 3 processors list

- **`par:create_2_tree_of_p(Start, End, Data, ^Tree)`**
  This is similar to the predicate **`create_list_of_p`**, but instead of a list, a tree is built. Overhead is a little bigger, but more parallelism is exhibited when spread data are large. Operation times grows like $\log_2$(number of processors + 1), with a factor twice of **`create_list_of_p`**. The result of data copy and so on is put in **`Tree`**. The tree is mapped on processors using the following recursive rule, assuming s is the start processor and e the end processor:

  – $\underline{e > s + 1}$
  The head of the tree is allocated on processor s; left sub-tree ranges in $[s + 1, \lfloor(s + e + 1)/2\rfloor]$ and right one ranges in $[\lfloor(s + e + 3)/2\rfloor, e]$.

  – $\underline{e = s + 1}$
  The head of the tree is allocated on processor s; left sub-tree ranges in $[e, e]$ and right sub-tree is empty.

  – $\underline{e = s}$
  The head of the tree is allocated on processor s; there is no sub-tree.

- **`par:sync_create_2_tree_of_p(Start, End, Data, ^Tree)`**
  Same as above, but **`Tree`** is bound only when all copies are over.

- **`par:apply_2_tree_of_p(Tree,^Copy,Mod,Pred,Args,^MO,^SO,SI,^TO,TI,^BO,BI)`**
  This applies a user predicate onto the tree created by one of the 2 previous predicates, in a similar way to **`apply_list_of_p`**. The predicate is identified as well by **`Mod`** and **`Pred`**, and the arity is the length of the argument vector **`Args`**. Arguments **`MO`**, **`SO`**, **`SI`** have the same function as in **`apply_list_of_p`**, especially for the matter of convention on sequential stream direction, with respect to the processor numbers.
  Some more capabilities have been introduced:

- Broadcasting of messages
  Each processor can send messages to all other processors, through the broadcast streams. On the top-level, BO is a stream receiving broadcasts from all processors, and BI can be used to send messages to all processors in the tree.
- Up/Down communication
  Each processor can send messages to the left and right sub-trees, and receive messages from them, through the tree streams. This is notably useful for a merged-sort, or an user implementation of filtered broadcast.
- Position
  Identification of the position in the binary tree.

Adequately, some special atoms can be used in the arguments list, besides the ones specified for `apply_list_of_p`. Namely:

- '`broadcast_in`' will be replaced by the stream carrying all messages from other processors in the tree.
- '`broadcast_out`' slot will conversely be replaced by a stream accepting messages to be duplicated and sent to all other processors in the tree.
- '`tree_left_in`' will be replaced by a stream of messages coming from the left sub-tree. Atoms '`tree_right_in`' and '`tree_up_in`' correspond to the right sub-tree and parent processor, respectively.
- '`tree_left_out`' will be replaced by a stream which can be used to send messages to the left sub tree. Atoms '`tree_right_out`' and '`tree_up_out`' correspond to the right sub-tree and parent processor, respectively.
- '`rank_in`' will be replaced by the depth of the current processor in the tree, starting at 0 for the root.
- '`path_in`' will be replaced by an integer, each bit of which corresponds to an ancestor node in the tree. More precisely, n-th ancestor's bit is at position $n-1$, the least significant bit being at position 0. If the subtree containing the current node is at the left of n-th ancestor, bit is set to 0, and to 1 otherwise. The root of the tree is arbitrarily given a path value of 0.
- '`children_in`' will be replaced by an integer indicating the number of children trees to the current node; 0 means the node is terminal, 1 means that only left child exists and 2 means that two children are present.

Other atoms are treated as within `apply_list_of_p`. We have also to precise the ends of various streams, for the top and end of the tree. For the top processor first; TO is connected to the up-output of this processor, i.e. the stream replacing atom '`tree_up_out`'. Conversely, TI corresponds to '`tree_up_in`'. For the nodes which are at the bottom of the tree, the tree link is looped back. Otherly said, a processor without right child sending a message to the stream which replaced '`tree_right_out`' would get this message back in the stream which has replaced '`tree_right_in`' in the argument list. Figure 4 pictures a tree with 4 nodes, with the connections available between processors, and some of the data not available in `apply_list_of_p`.

5.2 Synchronization

- `par:sync_wait_list_of_p(List,^Copy)`
  `Copy` is unified with a copy of `List` when all data therein are bound, at the first level. This is similar to `util:sync_wait/2`, but check operation respects the location of data on several processors. Using `util:sync_wait` over a distributed list would cause misplacing of the cons cells, and further slowdown of operations. One has to be careful, after an `apply_list_of_p`, that data have been written back or untouched, otherwise, this predicate will deadlock upon an unbound variable.

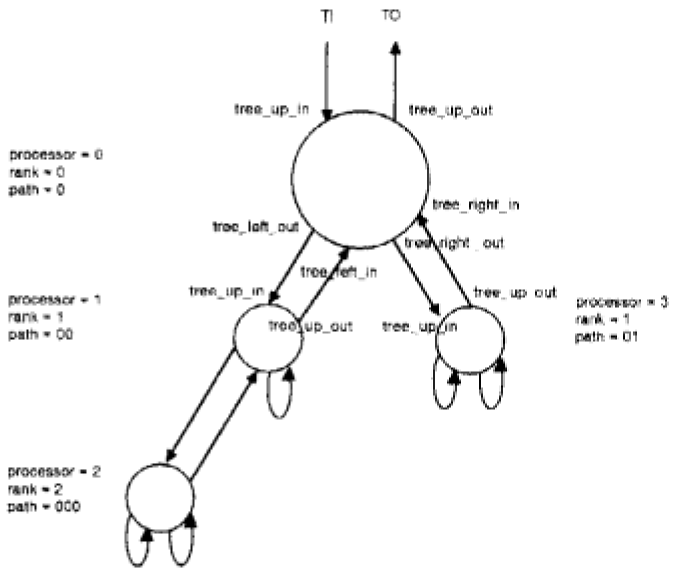

Figure 4: View of a 4 processors tree

- `par:sync_rec_wait_list_of_p(List,^Copy)`
  As `util:sync_rec_wait/2`, but this is also suited to distributed list. No need to say that using `util:sync_rec_wait` on a distributed list of data would wreck the nice single-pointer links between processors. Same remark as above about unbound variables, this time at all levels.

- `par:sync_wait_2_tree_of_p(List,^Copy)`
  `par:sync_rec_wait_2_tree_of_p(List,^Copy)`
  This is similar to the two previous predicates, but operates properly upon the tree data structures built by `create_2_tree_of_p` and possibly modified by `apply_2_tree_of_p`.

5.3 Version

- `par:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

6 Module string

The following predicates are an attempt at pattern matching, for the main part. We also provide some operations usually found for lists. Predicates are arranged according to the following topics:

- basic string primitives;
- substrings;
- sorting;
- data conversion;
- version.

6.1 Basic string primitives

- `string:reverse(String,^RString)`
  `RString` is unified with the reverse (palindrom) of character string `String`.

6.2 Substrings

The following section holds mainly predicates to find substrings in a character string, using Boyer-Moore or Knuth-Morris-Pratt algorithms, and predicates to use the position information thus provided.

- `string:findsubstr_bm(Pattern,String,^CopyString,^Res)`
  Searches for the character string `Pattern` in the string `String`. The result `Res` is a list containing the position of the first element of each occurrence, starting at 0. A copy of the original string is unified with `CopyString`. The Boyer-Moore algorithm is used to perform the task; it seems to be well suited for long `String`s, according to the overhead of table construction.

- `string:findsubstr(Pattern,String,^CopyString,^Res)`
  As above, but the Knuth-Morris-Pratt algorithm is used to perform the task. Its overhead to construct the recognition automaton is less than in the case of the Boyer-Moore algorithm, but it demands more comparisons to operate upon `String`.

- `string:sub_string(String,^CopyString,Start,Lng,^RString)`
  Extracts a substring of length `Lng`, from position `Start`, starting at 0, and unifies the result with `RString`. A copy of `String` is unified with `CopyString`. If `Start` and `Lng` arguments are out of the boundaries of the character string `String`, `sub_string` performs the intersection between the actual `String`, virtually extended on both sides, and the specified substring.

- `string:setsubstring(String,^CopyString,Start,Lng,^OutString,InsString,^RString)`
  Replaces the substring of length `Lng`, at position `Start`, starting from 0, with `InsString`, and unifies the former value of the substring with `OutString`. The modified string is unified with `RString`. If `Start` or `Lng` arguments are out of the boundaries of `String`, `setsubstring` inserts at the boundary. A copy of the original string is unified with `CopyString`.

- `string:split(String,Pos,^Before,^From)`
  Splits `String` at position `Pos`, starting from 0, and returns `Before` and `From` as the two parts of `String`.

6.3 Sorting

- `string:comparator(A,B,^S,^L,^Swapped)`
  Compares strings `A` and `B`, then unifies the smaller element with `S` and the other with `L`. If `L` is unified with `A`, `Swapped` is unified with atom `yes`, and with `no` otherwise.

6.4 Data conversion

- `string:string_to_vector(String,^Vect)`
  `Vect` is unified with a vector mapping string `String`.

- `string:string_to_list(String,^List)`
  `List` is unified with a list mapping string `String`.

- `string:sync_string_to_list(String,^List)`
  As above, but `List` is bound only when the conversion is over.

6.5 Version

- `string:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

7 Module util

This module is the historical startpoint of the FLIB library, as you could guess from its "vague" name. It contains various predicates which can neither fit in another module nor contribute alone to a new module. Some predicates were moved to other modules in the previous version, but we left here, for the sake of compatibility, the original predicates, which are in fact calling the code in the newer modules.

Predicates are gathered according to the following topics:

- integer to string and vice-versa;
- generic copy & compaction;
- parallel processing;
- synchronization;
- console output;
- enhanced timer & statistics;
- random number generator;
- version.

7.1 Integer to string and vice-versa

- `util:dec_int_to_name(Int,^String)`
  Returns `String`, a character string in the default representation, representing the integer `Int` in a decimal notation. A negative number is headed by the character '-'.

- `util:hex_int_to_name(Int,^String)`
  Same as above, but representation is done using hexadecimal notation. As well, negative numbers are appended to a '-'. Note that the result does not contain anything like `16#` or `16'` and is thus not suited to read using `gett` command or the like.

- `util:bas_int_to_name(Int,Base,^String)`
  Same as above, but an integer base **Base**, between 2 and 36, can be used for the representation. Same remarks as above about prefix and negative numbers.

- `util:dec_name_to_int(String,^Int)`
  Converts `String`, the decimal representation of an integer, into the integer `Int`. `String` should contain characters in the default representation. A negative number should start with '-'. Blank characters are ignored.

- `util:hex_name_to_int(String,^Int)`
  Same as above, but assumed representation is hexadecimal. There should be no prefix like `16#` or `16'`. Negative numbers start with '-'.

- `util:bas_name_to_int(Int,Base,^Int)`
  Same as above, but an arbitrary base, between 2 and 36, can be used. Same remarks as above about prefix and negative numbers.

7.2 General copy & compaction

These predicates are useful to avoid multiple references to complex objects which are to be used in different operations, possibly on different processors.

- `util:copy(Object,^Copy1,^Copy2)`
  Makes a copy of `Object`. `Copy1` contains the initial argument while Copy2 is freshly allocated. Copying is done recursively, but only atom, vectors, list, integer and string datatypes are supported (no *code* datatype). If there are unbound terms, this predicate will deadlock.

- `util:sync_copy(Object,^Copy1,^Copy2)`
  Same as above, but both results are returned only after all copies have been finished. That requires that all terms are bound at the bottom level before completion occurs.

- `util:flash_copy(Object,^Copy1,^Copy2,^Sync)`
  Same as `copy/3`, but if some source data is unbound at copy time, it is replaced in the copy by an unbound data. There is no double reference generated, nor deadlock. For the matter of lists, end of list is waited for. `Sync` is bound when copy is finished.

- `util:object_to_string(Object,^Copy,^String)`
  Converts an arbitrary data `Object` into the string `String`, in an implementation dependant format. This is useful to speed up communications, when an object is passed from one processor to another. `Copy` is a copy of `Object`. `String` is bound when the conversion is finished.

- `util:string_to_object(String,^Copy,^Object)`
  Conversely, this converts the string produced by the above predicated into the original object. **Warning**: atoms are not restored; they are replaced by the corresponding atom number. `Copy` is a copy of the original `String`.

7.3 Parallel processing

The following predicates are code in the module `par`. For upward compatibility reasons, they are callable from here also. See module `par` for explanations.

- `util:create_list_of_p(Start, End, Data, ^List)`
  Like `par:create_list_of_p/4`.

- `util:sync_create_list_of_p(Start, End, Data, ^List)`
  Like `par:sync_create_list_of_p/4`.

- `util:apply_list_of_p(List, ^Copy, Mod, Pred, Args, ^MO, ^SO, SI)`
  Like `par:apply_list_of_p/8`.

- `util:sync_wait_list_of_p(List,^Copy)`
  Like `par:sync_wait_list_of_p/2`.

- `util:sync_rec_wait_list_of_p(List,^Copy)`
  Like `par:sync_rec_wait_list_of_p/2`.

7.4 Synchronization

As we said in our forewords, it is better to think of synchronization when programming, in order to avoid a situation in which a structure is bound whereas some of its elements may be unbound. But in some case, e.g. for a kludge, the only way to be sure that a computation is over is to wait recursively until all relevant data are bound. The following predicates have this purpose:

- `util:sync_wait(Data, ^Copy)`
  Returns in `Copy` a copy of `Data` when the latter is bound. If it is a list or a vector, its element should also be bound. Check is *not* performed further deep in structures.

- `util:sync_rec_wait(Data, ^Copy)`
  This is similar to the previous case, but the check is performed recursively, until all terms in structures are found to be bound.

7.5 Console output

The guard predicate `display_console/1` is rather limitated, and in order to print atoms or complex objects in a readable format, we offer the following predicates:

- `util:p_console(Data, ^Copy)`
  Prints `Data` on the console, and returns a copy of it in `Copy`. Data should be recursively bound, or nothing will be print and the predicate will deadlock. Note that *code* datatype is not supported.
- `util:sync_p_console(Data, ^Copy)`
  Same as above, but `Copy` is bound after the display has been done.
- `util:flash_p_console(Data, ^Copy, ^Sync)`
  Same as above, but unbound data are print as an underscore sign '_', and do not cause the predicate to deadlock. `Sync` is bound only after that display has been completed. `Copy` contains a copy of `Data`.

7.6 Enhanced timer & statistics

For performance measurement, and especially to track synchronization problems, a comprehensive debugging environment could be wished. Very unfortunately, this would demand a long term effort in development and the consideration of implementation details, both points often disregarded by most computer scientists.

For a good idea of what would be a minimal framework, our reader may read the enlightened paper of T. Lehr, Z. Segall, D.F. Vrsalovic & al., *Visualizing performance debugging*, in Computer Magazine of October 1989.

More modestly, we present here some simple predicates which allow the capture of time or statistical data in an easier fashion than in the bare PIMOS. These primitives work also on PDSS, but the poor emulation of both timer and multi-processing makes them of little use.

- `util:timer(Stream)`
  Provides all commands of the usual timer device. In addition, starting time is kept, and additionnal messages are offered:
  - `rel_get_count(^Status)`
    Works like `get_count`, except that the returned count starts from the last `rel_get_count` message, or from the timer invocation, for the first invocation.
  - `rel_get_count_start(^Status)`
    Works like message `get_count`, except that the returned count starts from timer invokation.
  - `rel_on_at_start(Count, ^Status)`
    Works like message `on_at`, except that the time is specified starting at timer invokation.

  For all of these commands, `Status` specification is similar to the one of the latest PIMOS release.
- `util:start_stats(Code, Args, ^Stream)`
  Starts a shōen executing the predicate identified by *code* data `Code`, with arguments `Args`. When timing or statistics requests are sent via the following predicates, corresponding results are inserted in the stream `Stream`, as vectors of the form `{P,Str,V}`; `P` is the number of the processor on which the `start_stats` predicate has been run, `Str` is a user-supplied data and `V` is the result of the request. Predicates which should be used to emit a request are:
  - `util:req_time(Str, ^Back)`
    This predicate requests elapsed time, in milliseconds, since the invocation of `start_stats`. Actual request does not start before `Str` is bound. `Back` is bound when time is available and has the form `{P,V}`, where `V` is the time. `P` as the same meaning as in the `start_stats` stream:
  - `util:req_rel_time(Str, ^Back)`
    As above, but time is counted from the last invocation of this predicate, or from evocation of the `start_stats/2` predicate, in the case of the first invocation.

- `util:req_red(Str, ^Back)`
  Instead of the time, this requests the number of reductions performed within the shōen created by `start_stats/2`.
- `util:req_rel_red(Str, ^Back)`
  As above, but reductions are counted from the last call of this predicate, or from the creation of the shōen, in the case of the first invocation.

7.7 Random number generator

- `util:random(Seed, ^NextSeed)`
  This predicate takes a random seed `Seed`, between -2^{31} and $2^{31}-1$, and returns the next seed to use, a pseudo-random integer in the same range.
- `util:random_bound(Range, ^Random, Seed, ^NextSeed)`
  `Seed` and `NextSeed` have the same function as in the previous predicate. `Range` is an integer between 1 and $2^{31}-1$. `Random` is a pseudo-random integer in the range 0 to `Range`-1.

7.8 Version

- `util:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

8 Module vect

This module offers predicates of similar shape to the ones as in the module `list`, but is suited to evaluation over vectors. Predicates are arranged according to the following topics:

- basic vector primitives;
- subvectors;
- set operations;
- sorting;
- set arithmetic;
- vector arithmetic;
- data conversion;
- version.

Be aware when you use vectors that multiple references to vectors cause indirections when modified elements are further accessed. This implies increased acces time.

8.1 Basic vector primitives

- `vect:append(VectA,VectB,^RVect)`
  Appends vectors `VectA` and `Verb` and puts the result in `RVect`. Note that vector elements don't need to be bound.
- `vect:reverse(Vect,^RVect)`
  Reverses of `Vect` at the top level and puts the result in `RVect`.
- `vect:rec_reverse(Vect,^RVect)`
  As above, but reversing is done at all levels for vectors.
- `vect:sync_rec_reverse(Vect,^RVect)`
  Same as above, but `RVect` is bound only when reverse is over.
- `vect:subst(Vect,Old,^Copy,New,^RVect)`
  Operates substitution of `Old` by `New` in `Vect`, at the top level. Result is unified with `RVect`. Makes multiple references to `New`. `Copy` is a copy of `Old`.
- `vect:rec_subst(Vect,Old,^Copy,New,^RVect)`
  Same as above, but substitution is done at all levels.
- `vect:sync_rec_subst(Vect,Old,^Copy,New,^RVect)`
  Same as above, but `RVect` is bound when all substitutions are done.
- `vect:remove(Vect,Elem,^Copy,^RVect)`
  Removes all occurrences of `Elem` in `Vect`, at the top level. The result `RVect` may therefore be shorter than `Vect`. `Copy` is a copy of `Elem`.
- `vect:rec_remove(Vect,Elem,^Copy,^RVect)`
  Same as above, but remove is performed at all levels.
- `vect:sync_rec_remove(Vect,Elem,^Copy,^RVect)`
  Same as above, but `RVect` is bound after all remove have been done.
- `vect:member(Vect,^VectCopy,Elem,^Copy,^Pos)`
  Finds the first occurrence of `Elem` in `Vect`, and returns its position `Pos`. If no occurrence is found, the `Pos` is set to -1. `Copy` is a copy of `Elem`, and `VectCopy` a copy of `Vect`.

8.2 Subvectors

- `vect:findsubvect(Subvect,SubVectCopy,Vect,^CopyVect,^Res)`
  Searches for occurrences of `Subvector` in `Vector`. The result `Res` is a list containing the position of the first element of each occurrence. The Knuth-Morris-Pratt algorithm is used to perform this task, and any object may feature in `SubVect` and `Vect` as long as it is possible to check in the guard that they unify together or not without inducing deadlock.

- `vect:subvect(Vect,Start,Lng,^RVect)`
  Extracts the subvector `RVect` of length `Lng` from the vector `Vect`, starting from `Start`. Position of the first element is 0. `Vect` is completed virtually on the left and right by a void vector. Hence, if `Start` or `Start` + `Lng` are out of `Vect`, intersection with the actual `Vect` is returned.

- `vect:setsubvect(Vect,Start,Lng,^OldVect,InsVect,^RVect)`
  Replaces the subvector of length `Lng` starting at position `Start`,verb –InsVect—. `OldVect` contains the part which has been replaced. `RVect` contains the modified vector. If `Start` and `Lng` arguments are out of the boundaries of `Vect`, this predicate inserts respectively at the beginning or at the end of the vector.

- `vect:split(Vect,Pos,^Before,^From)`
  Splits `Vect` after the element at position `Pos`, starting from 0. `Before` is a pointer to vector before the split point, and `From` is the second part.

8.3 Set operations

As for sets represented as lists, a set in vector form is a vector with no identical elements, except if explicitely allowed in the following predicates.

- `vect:union(VectA,VectB,^RVect)`
  Union of sets `VectA` and `VectB`. the result, `RVect`, is bound when union is over.

- `vect:inter(VectA,VectB,^CopyVectB,^RVect)`
  As above, but intersection is done instead of union.

8.4 Sorting

- `vect:comparator(A,B,^S,^L,^Swapped)`
  For vectors containing only atoms or integers. Compares `A` and `B`, then unifies the smaller element with `S` and the larger one with `L`. If `L` is unified with `A`, `Swapped` is unified with the atom `yes`, and `no` otherwise.

- `vect:sort_a(Vect,^RVect)`
  `vect:sort_d(Vect,^RVect)`
  Sorts a vector. predicate `sort_a` produces an increasing order, `sort_d` the decreasing one, for integers. For atoms, this function gives an arbitrary order, according to the atom number. The obtained order is a total order though. The result of the sort operation is unified with `RVect`.

- `vect:quicksort_a(Vect,^RVect)`
  `vect:quicksort_d(Vect,^RVect)`
  Sorts a vector of elements with the same datatype. It tests the type of the first element in the vector and uses the specific sorting predicate. Its speeds up sorting by roughly a factor of 2.

- `vect:nodoub(Vect,^RVect)`
  Eliminates consecutive repetitions in vector `Vect`. Its use consequently to quicksort provides an ordered vector with unique elements. `RVect` is bound with the result when operations are finished.

8.5 Set arithmetic

- `vect:mini(Vect,^CopyVect,^Mini)`
  `Mini` is unified with the minimum element in the vector of integers `Vect`, whose copy is returned in `CopyVect`.

- `vect:maxi(Vect,^CopyVect,^Maxi)`
  As above, but we look for the maximum integer instead of the minimum.

- `vect:sum(Vect,^CopyVect,^Sum)`
  `Sum` is unified with the sum of all elements of the vector of integers `Vect`, whose copy is returned in `CopyVect`.

- `vect:product(Vect,^CopyVect,^Prod)`
  As above, but we look for the product instead of the sum.

- `vect:andv(Vect,^CopyVect,^Result)`
  This predicate operates the logic "and" of a vector of booleans : if any element in `Vect` is zero, `Result` is unified with 0, otherwise it is unified with 1 (notably if the vector is empty). The vector is scanned in usual order, but asynchronous binding is supported. For example, the vector `{_|0}` will produce a 0. `CopyVect` is a copy of `Vect`.

- `vect:sync_andv(Vect,^CopyVect,^Result)`
  This is the same as above, but `Vect` is checked sequentially.

- `vect:orv(Vect,^CopyVect,^Result)`
  This predicate is similar to `andv`, but 1 is returned if any 1 is in `Vect`, and 0 is returned otherwise, notably if the vector is empty.

- `vect:sync_orv(Vect,^CopyVect,^Result)`
  This is the same as above, but `Vect` is checked sequentially.

8.6 Vector arithmetic

- `vect:vector_add(Vect1,Vect2,^AddVect)`
  Adds the two vectors of integer `Vect1` and `Vect2`, which should have the same length, to produce the vector `AddVect`.

- `vect:vector_sub(Vect1,Vect2,^SubVect)`
  As above, but `Vect2` is subtracted to `Vect1`.

- `vect:vector_scal_product(Vect1,Vect2,^Scal)`
  `Scal` is unified with the scalar product of vectors `Vect1` and `Vect2`.

8.7 Data conversion

- `vect:vector_to_list(Vect,^List)`
  Transforms the vector `Vect` into the list `List`, at the top level.

- `vect:sync_vector_to_list(Vect,^List)`
  As above, but `List` is bound when the conversion is over.

- `vect:rec_vector_to_list(Vect,^List)`
  As `vector_to_list/2`, but transformation is done at all levels, for all vectors.

- `vect:sync_rec_vector_to_list(Vect,^List)`
  As above, but `List` is bound when the conversion is over.

- `vect:vector_to_string(Vect,^Copy,CharLength,^String)`
  Transforms the vector of integers `Vector` into the character string `String`. Characters have `Charlength` bits, taken amongst 1, 4, 8, 16 or 32 bits per character.

- `vect:vector_to_col_matrix(Vect,^Mat)`
  Transforms the vector of integers `Vector` into the column matrix `Mat`, in a format compatible with the one used in the module `mat`.

- `vect:vector_to_line_matrix(Vect,^Mat)`
  Transforms the vector of integers `Vector` into the line matrix `Mat`, in a format compatible with the one used in the module `mat`.

8.8 Version

- `vect:version(^Num, ^Comment)`
  `Num` is unified with a vector holding version and release number. -1 and -2 correspond with beta and alpha release. `Comment` holds a string with some information on the current release.

9 Examples

Two toy problems are presented: searching occurrences of a subvector in a vector and sorting a list. We introduce first the sequential version of these programs, then the parallel version. It shows the use of FLIB's parallel facilities and highlights difficulties related to parallel programming. Time measurements follow, in order to give some advice to the next programmers about the nature of the technical choices to do.

9.1 Search of a subvector in a vector

The basic algorithm used to perform the search is from Knuth-Morris-Pratt. We begin with the sequential implementation, then we make a naive parallel version, splitting the task on the Multi-PSI processors. Finally we develop a more sophisticated version, including data-compression, synchronization and time measurements. These versions have been run on the multi-PSI and the results are commented.

9.1.1 Sequential version

The sequential version of Knuth-Morris-Pratt exists in FLIB, for strings and vectors.

9.1.2 Naive parallel version

To make the parallel version we use the `par:apply_list_of_p/8` predicate. It distributes the data to all the processors and gather results.

```
par_nkmp(ProIn, PatV, Vect, Result) :-
    vector(PatV, Patln),
    vector(Vect, Vln) |
        MaxPro := Vln/Patln,
        fel:mini(MaxPro, ProIn, Pro),
        par:create_list_of_p(1, Pro, [PatV, Patln, Vect, Pro], Data),
        par:apply_list_of_p(Data, _, ex1, loc_nkmp,
                            {data_in, processor_in, merge_out}, Result, _, []).
```

Predicate `fel:mini/3` restricts the number of processors in the case of short lists. `create_list_of_p/4` takes the arguments list, duplicates it and spreads the arguments onto the processors from 1 to `Pro`. `apply_list_of_p/8` evaluates the `loc_nkmp/3` predicate on every processor. The local results are merged, then output via `Result`.

```
loc_nkmp([Pat, Patln, Vect, Pro], Cur_pro, STREAM_out) :- true |
        my_subvect(Vect, Patln, Pro, Cur_pro, My_part, Offset),
        vect:findsubvect(Pat, _, My_part, _, Res),
        add_offset(Res, Offset, STREAM_out).
```

`loc_nkmp/3` is the predicate applied by each processor. It extracts the data used by the processor `Cur_pro` and gives it to the sub-vector search `vect:findsubvect/5`. The `add_offset/3` predicate inputs the position of the occurrences relative to the local work and outputs the absolute positions.

```
my_subvect(Vect, Patln, Pro, Cur_pro, Resul, Start) :-
    wait(Patln),
    vector(Vect, Vln) |
        Seg:= Vln/Pro+1,
        Start:= (Cur_pro - 1)* Seg,
        (Start < Vln ->
            Len := Seg+Patln-1,
            vect:subvect(Vect, Start, Len, Resul);
         otherwise;
            true -> Resul={}).
```

my_subvect/6 extracts the part of the vector which will be used locally by processor Cur_pro. add_offset/3 adds an offset.

```
add_offset([A|B], Offset, Res) :- true |
        Res=[~(A + Offset) | Res1],
        add_offset(B, Offset, Res1).
add_offset([], _, Res) :- true | Res=[].
```

9.1.3 Final parallel version

The naive version performs tasks in a parallel manner, but does not take care of data transmission, synchronization and time measurements. To make an efficient version, these problems should be taken into account, in order to debug the algorithm as well as to understand the parallel performances.

```
par_kmp(ProIn, PatV, Vect, Resul, STREAM) :- true |
        fel:get_code(ex1, par_kmp1, 4, FctCode, normal),
        util:start_stats(FctCode, {ProIn, PatV, Vect, Resul}, STREAM).
```

start_stats/3 begins a shoen and allows measurement facilities. STREAM collects the time measurements. One has to ensure the good synchronization of the program to collect reliable measurements.

```
par_kmp1(ProIn, PatV, Vect, Resul1) :-
    vector(PatV, Patln),
    vector(Vect, Vln) |
        MaxPro := Vln/Patln,
        fel:mini(MaxPro, ProIn, Pro),
        util:object_to_string(PatV, _, PatVZ),
        util:object_to_string(Vect, _, VectZ),
        (wait(Pro), wait(PatVZ), wait(VectZ) ->
            sync_create_list_of_p(1, Pro, [PatVZ, Patln, VectZ, Pro], Data),
            apply_list_of_p(Data, _, ex1, loc_kmp,
                        {data_in, processor_in, merge_out}, Resul, _, []),
            measure_time(Resul,Resul1)).
```

The util:object_to_string/3 compresses objects into strings. Because Multi-PSI sends vectors and strings in one packet to the other processors, it is suggested to use these data-representations, or to compress the data in the same manner as above. Notice the synchronization done before duplicating and sending the data to the processors, it is requested for efficiency.

```
loc_kmp([PatZ, Patln, VectZ, Pro], Cur_pro, STREAM_out) :- true |
        util:string_to_object(PatZ, _, Pat),
        util:string_to_object(VectZ, _, Vect),
        (wait(Pat), wait(Vect) ->
            my_subvect(Vect, Patln, Pro, Cur_pro, My_part, Offset),
            vect:findsubvect(Pat, _, My_part, _, Res),
            add_offset(Res, Offset, STREAM_out)).
```

The data is decompressed by util:string_to_object/3. Again we synchronize.

```
sync_create_list_of_p(1, Pro, [PatVZ, Patln, VectZ, Pro], Data) :- true |
        util:req_rel_time("compression", Sync),
        (wait(Sync) ->
             par:sync_create_list_of_p(1, Pro, [PatVZ, Patln, VectZ, Pro],Data)).
```

Since two test structures cannot be nested, a new predicate is needed for the measurement synchronization.

```
apply_list_of_p(Data, _, ex1, loc_kmp, {data_in, processor_in, merge_out},
                Resul, _, []) :-
    wait(Data) |
        util:req_rel_time("data spreading", Sync),
        (wait(Sync) ->
            par:apply_list_of_p(Data, _, ex1, loc_kmp,
                            {data_in, processor_in, merge_out}, Resul, _, [])).


measure_time(Resul,Resul1) :- true |
        util:sync_rec_wait(Resul, Resul1),
        (wait(Resul1) -> util:req_time("Total time", _)).
```

9.1.4 Performance measurements

Three examples have been run on a Multi-PSI/V2. In these examples we search for a subvector of 15 elements in a 5000 elements vector. These vectors have been randomly generated, taking elements of a given vocabulary. The vocabularies for the three examples are:

ex1	{a,b,c,d}
ex2	{[a,[b,c]],[1,2,3,4],[apply,add,a,b,c,d],mult,[1,2,3,4],5,6}
ex3	{[a,[b,c,d,e,f]],[1,2,3,4,5,6],[a,a,a,b,c,d,g,h,i],[m,[1,a,b,c,[2,3],4],5,6]}

kmp		sequential	2 processors		5 processors		10 processors		15 processors	
			naive	final	naive	final	naive	final	naive	final
ex1	Kred	20	54	109	76	176	112	288	150	406
	s	0.5	1.3	2.3	2.1	3.2	3.7	5.1	5.4	7.2
ex2	Kred	20	138	477	275	919	501	1660	732	2410
	s	0.5	9	9.9	24.4	17.8	49.3	31.5	74.5	45.3
ex3	Kred	20	166	654	344	863	639	2277	937	3308
	s	0.5	13.6	13.5	29.4	25	56.8	44.4	83.2	65.8

Table 1: Time and reduction measures of kmp

As a first comment of Table 1, let's note that the sequential version runs in constant time, whatever the vocabulary is. It is always faster than the parallel version. This is explained by Figure 5. First of all, to speed-up the transmission, we compress the data. This takes approximatively 5 seconds. Then, this data is spread over the processors. This takes between 5 to 55 seconds, according to the number of processors. The data is then decompressed, in about 3 seconds, and finally the kmp algorithm is performed in less than 0.5 seconds.

Because of the high simplicity of kmp search, the sequential version is several degree of magnitude faster than parallel implementations on Multi-PSI/V2. The ratio between data transmission and the work to be carried out is too high. Time measures and the related listings show that the additional work to be done by the parallel version is not reasonable (with 15 processors, we have to make 165.4 times more reductions than the sequential version).

Another interesting point concerns the speed of the naive parallel version which is faster than the final version of kmp in example 1. In fact, in the case of a simple vocabulary, the data compression used in the final version does not change the transmission performances, but introduces some compression overhead. The more the data becomes hairy, the more the compression is efficient. The two last examples show it. We can note that the final version of kmp has a much higher *reduction-rate* $= \frac{Reductions}{Time}$. This illustrates the vital role of synchronization and inter-processor communication. For a discussion of evaluation criteria, one can refer the paper of K. Taki, *Measurements and Evaluation of the Multi-PSI/V1 System, in* Programming of Future Generation Computers, vol. 2, North-Holland, published in 1988.

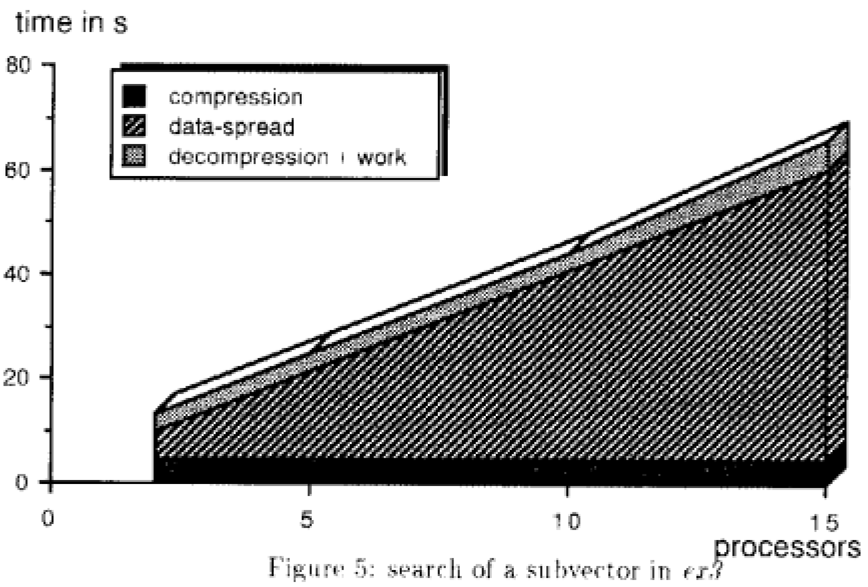

Figure 5: search of a subvector in *ex3*

9.2 Sorting and removing doubles

We implemented a sort to remove all the double elements in a list. For the sake of clarity, this program has been designed for integers only. The sorting algorithm we use in the sequential version is the quick-sort proposed in the Technical Memorandum TR-209 from ICOT, *Introduction to Guarded Horn Clauses*, by K.Ueda, published in 1986. Subsequently, a simple loop removes consecutive doubles.

We took this algorithm and put it in the parallel environment. The tree-spreading function of FLIB provides an efficient data spreading structure to do that. The first parallel version spreads the whole data set to the processors. Each of them performs a sort on its data and then merges the results with those issued by the child-processors. The second parallel version differs by the data transmitted to processors. Each of them takes its own part of the work and shares the data to be sorted between the children. Performance measurements give an idea about the gained efficiency.

9.2.1 Sequential version

Sort, then suppression of double, using FLIB predicates.

9.2.2 First parallel version: Ver1

In short, it consists in a tree-spreading of all the data, local sorting, then a merging of the results.

```
par_sort(ProIn, Xin, Xout, STREAM) :- true |
        fel:get_code(ex2, par_sort1, 3, FctCode, normal),
        util:start_stats(FctCode, {ProIn, Xin, Xout}, STREAM).
```

To allow time measurements.

```
par_sort1(ProIn, Xin, Xout) :- true |
        list:length(Xin, Xin1, Xln),
        fel:mini(Xln, ProIn, Pro),
        list:list_to_string(Xin1, 32, XinZ),
        (wait(Pro),wait(XinZ) ->
            sync_create_2_tree_of_p(1, Pro, [XinZ, Xln, Pro], Data),
            apply_2_tree_of_p(Data, ex2, loc_sort,
                              {data_in, processor_in, tree_left_in,tree_right_in,
                              tree_up_out}, Global_result),
            string:string_to_list(Global_result, Final_result),
            measure_time(Final_result, Xout)).
```

The number of processors is limited by the fel:mini/2. Data are compressed by list:list_to_string/3 to reduce data-transmission. The data is spread over a tree. Each processor gets the same data. The results of sorting are output via Global_result. They are then decompressed by string:string_to_list/2.

```
loc_sort([XinZ, Xln, Pro], Cur_pro, TLI, TRI, TUO) :- true |
        string:string_to_list(XinZ, Xin),
        loc_sublist(Xin, Xln, Pro, Cur_pro, My_part),
        list:sync_quicksort_a(My_part, Res),
        list:nodoub(Res, Resdl),
        list:list_to_string(Resdl, 32, ResZ),
        merge_result(ResZ, TLI, TRI,TUO).
```

Data coming from par_sort1/3 are decompressed by string:string_to_list/2. The local work to be done is extracted by loc_sublist/5. The sorted result is compressed into a string by list:list_to_string/3. The merge_result/4 predicate takes this local result and the results of the left and right childs, merges them, removes the redundancies and transmits the result to the father processor (we do not include its listing here for the sake of brevity).

```
sync_create_2_tree_of_p(1, Pro, Elem, Data) :- true |
        util:req_rel_time("compression", Sync),
        (wait(Sync) -> par:sync_create_2_tree_of_p(1, Pro, Elem, Data)).
```

Synchronization of time measurements

```
apply_2_tree_of_p(Data, Mod, Pred, Arg, Resul) :-
    wait(Data) |
         util:req_rel_time("data spreading", Sync),
        (wait(Sync) ->
            par:apply_2_tree_of_p(Data, _, Mod, Pred, Arg, _, _, [], Resul, [], _, [])).


loc_sublist(Xin, Xln, Pro, Cur_pro, Resul) :- true |
        Seg:= Xln/Pro + 1,
        Start:= (Cur_pro - 1)* Seg,
        list:sublist(Xin, Start, Seg, Resul, []).
```

Extraction of the local work to be done.

9.2.3 Second parallel version: Ver2

The main difference with the preceding version is the data transmission. Here we transmit only the necessary data, thus transmission speed should be improved.

```
par_sort1(ProIn, Xin, Xout) :- true |
        list:length(Xin, Xin1, Xln),
        fel:mini(Xln, ProIn, Pro),
        list:list_to_string(Xin1, 32, XinZ),
        (wait(Pro),wait(XinZ) ->
            Seg := Xln / Pro + 1,
            sync_create_2_tree_of_p(1, Pro, Seg, Data),
            apply_2_tree_of_p(Data, XinZ, ex3, loc_sort, {data_in,tree_left_in,
                              tree_right_in, tree_up_in, tree_left_out,
                              tree_right_out, tree_up_out}, Global_result),
            string:string_to_list(Global_result, Final_result),
            measure_time(Final_result, Xout)).
```

The data are transmitted in two different manners. The small ones, Xln and Pro are written in a vector which is spread over all processors. The large data, the list to be sorted, is given to the first processor. The latter extracts the data treated locally and splits the remaining one into two equal parts. Those are then transmitted to the children processors.

```
apply_2_tree_of_p(Data, XinZ, Mod, Pred, Arg, Resul) :-
    wait(Data) |
         util:req_rel_time("data spreading", Sync),
         (wait(Sync) ->
             par:apply_2_tree_of_p(Data, _, Mod, Pred, Arg, _, _, [],
                                    Resul, XinZ, _, [])).

loc_sort(Seg, TLI, TRI, TUI, TLO, TRO, TUO) :- true |
        loc_sublist(TUI, Seg, My_part, TLO, TRO),
        list:sync_quicksort_a(My_part, Res),
        list:nodoub(Res, Resdl),
        list:list_to_string(Resdl, 32, ResZ),
        ex2:merge_resul(ResZ, TLI, TRI,TUO).
```

TUI is the data coming from the upper processor, it is the list to be treated.

```
loc_sublist(Xin, Seg, Resul, TLO, TRO) :-
    string(Xin, Xinln, _), Xinln > Seg |
        Middle := (Xinln - Seg)/2,
        string:split(Xin, Seg, Resul1, Next),
        string:split(Next, Middle, Begin, End),
        (wait(Resul1), wait(Begin), wait(End) -> Resul1=Resul, Begin=TLO, End=TRO).
otherwise.
loc_sublist(Xin, _, Resul, TLO, TRO) :- true |
         Resul = Xin, TLO = string#"", TRO = string#"".
```

The data to be treated is cut in 3 parts, the first will be treated by the local processor. The remaining data is cut into two equal parts, forwarded to left and right children.

9.2.4 Performance measurements

This program was run to sort 3 lists, from 100, 500 and 5000 atomic elements. The results are shown in Table 2.

sort		sequential	2 processors		5 processors		10 processors		15 processors	
			ver1	ver2	ver1	ver2	ver1	ver2	ver1	ver2
100	Kred	3.8	10	9.3	17	16	29	28	42	39
	s	0.16	0.27	0.35	0.3	0.36	0.35	0.44	0.41	0.46
500	Kred	61	46	34	41	32	53	42	69	53
	s	0.8	0.51	0.6	0.41	0.46	0.5	0.47	0.51	0.5
5000	Kred	5104	2650	1677	1197	841	764	552	680	529
	s	58.4	15.5	15.7	3.8	3.8	2.3	2.8	2.1	2.1

Table 2: Time and reductions measures with *sort*

By dealing with 100 elements, the parallel implementation looses, compared to the sequential one. With 500 elements, the best performance is obtained with 5 processors using the first parallel version. To sort 5000 elements, the first parallel implementation using 15 processors turns out to be faster. The parallel

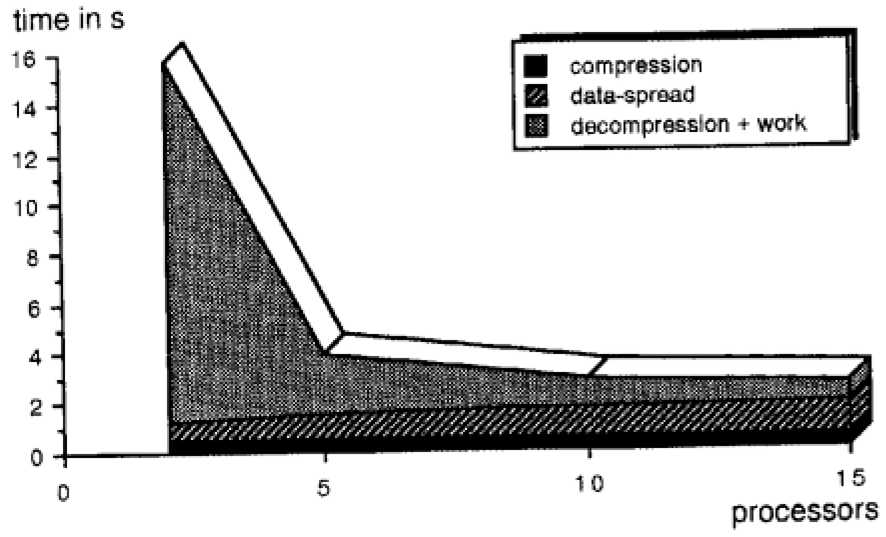

Figure 6: Sorting of 5000 elements

performance increase is super-linear. One can observe the evolution of the number of reductions performed by the program. It simply means that the sequential version is far from optimality (the number of reductions gives obvious information). Figure 6 represents in detail, the work done to sort 5000 elements. Roughly, the data compression time is constant, the data spreading increases slightly, the sorting time decreases dramatically. This result is due to the complexity of the sorting algorithm. Although related to quicksort, which is in $\mathcal{O}(n\log(n))$, its complexity tends to $\mathcal{O}(n^2)$ when data are not well distributed in the initial list to sort. This figure of complexity becomes in parallel, considering the cost of the merge operation, $\mathcal{O}\left(\lceil\frac{n}{P_{ro}}\rceil^2\right) + \mathcal{O}(n\log n)$. When sort involves 5000 elements, the detailed analysis predicts a slowdown for more than 15 processors, since the transmission time increases more than the decrease of the processing time. The tree-data structure used in this algorithm allows fast data transmissions and merging of the partial results in parallel.

9.3 Conclusion

The two examples we showed above lead to some hints in parallel programming on the Multi-PSI:

- The search of a subvector in a vector by the kmp algorithm is a brilliant counter-example of the usefulness of parallelism. In the worst case, the parallel implementation with 15 processors takes 166 times longer (naive version), or makes 165 more reductions (final version) that the sequential version. These awful results are due to the simplicity of the kmp algorithm, compared to the size of the data to be transferred.
- The sorting algorithm showed in contrary a super-linear speed-up, when dealing with big examples. In fact, the principle used by the parallel implementation transformed the sorting, breaking the complexity of the original algorithm. The parallel algorithm shows however a speedup limit, with a certain number of processors. By using more processors, time is wasted in communications, etc.

Conception of efficient parallel programs remains an art. More work has to be done towards a better knowledge of the multi-PSI.