Developer’s Guide

to

the PARI library

(version 2.13.3)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université de Bordeaux, 351 Cours de la Libération
F-33405 TALENCE Cedex, FRANCE
e-mail: pari@math.u-bordeaux.fr

Home Page:
http://pari.math.u-bordeaux.fr/
# Table of Contents

## Chapter 1: Work in progress

1.1 The type `t_CLOSURE` ............................................. 5  
1.1.1 Debugging information in closure .............................. 5 
1.2 The type `t_LIST` .................................................. 6  
1.2.1 Maps as Lists .................................................. 7 
1.3 Protection of noninterruptible code ............................... 8  
1.3.1 Multithread interruptions ..................................... 8 
1.4 $F_{l^2}$ field for small primes $l$ ................................ 9  
1.5 Public functions useless outside of GP context ................... 9  
1.5.1 Conversions .................................................. 9  
1.5.2 Output ....................................................... 10  
1.5.3 Input ....................................................... 10  
1.5.4 Control flow statements ....................................... 10  
1.5.5 Accessors .................................................... 11  
1.5.6 Iterators .................................................... 11  
1.5.7 Local precision ............................................... 11  
1.5.8 Functions related to the GP evaluator ......................... 12  
1.5.9 Miscellaneous ................................................ 12  
1.6 Embedded GP interpreter .......................................... 12  
1.7 Readline interface ................................................. 13  
1.8 Constructors called by `pari_init` functions ..................... 13  
1.9 Destructors called by `pari_close` ................................ 14  
1.10 Constructors and destructors used by the `pthreads` interface 14

## Chapter 2: Regression tests, benches

2.1 Functions for GP2C ................................................. 16  
2.1.1 Functions for safe access to components ....................... 16

## Chapter 3: Parallelism

3.1 The PARI multithread interface .................................. 17  
3.2 Technical functions required by MPI ............................. 18  
3.3 A complete example .............................................. 18

Index .............................................................. 21
Chapter 1:  
Work in progress

This draft documents private internal functions and structures for hard-core PARI developers. Anything in here is liable to change on short notice. Don’t use anything in the present document, unless you are implementing new features for the PARI library. Try to fix the interfaces before using them, or document them in a better way. If you find an undocumented hack somewhere, add it here.

Hopefully, this will eventually document everything that we buried in paripriv.h or even more private header files like anal.h. Possibly, even implementation choices! Way to go.

1.1 The type t_closure.

This type holds closures and functions in compiled form, so is deeply linked to the internals of the GP compiler and evaluator. The length of this type can be 6, 7 or 8 depending whether the object is an “inline closure”, a “function” or a “true closure”.

A function is a regular GP function. The GP input line is treated as a function of arity 0.

A true closure is a GP function defined in a nonempty lexical context.

An inline closure is a closure that appears in the code without the preceding -> token. They are generally attached to the prototype code 'E' and 'T'. Inline closures can only exist as data of other closures, see below.

In the following example,

```plaintext
f(a=Euler)=x->sin(x+a);
g=f(Pi/2);
plot(x=0,2*Pi,g(x))
```

f is a function, g is a true closure and both Euler and g(x) are inline closures.

This type has a second codeword \[ z[1] \], which is the arity of the function or closure. This is zero for inline closures. To access it, use

```plaintext
long closure arity(GEN C)
```

- \[ z[2] \] points to a t_STR which holds the opcodes. To access it, use

```plaintext
GEN closure get code(GEN C).
```

```plaintext
const char * closure codestr(GEN C) returns as an array of char starting at 1.
```

- \[ z[3] \] points to a t_VECSMALL which holds the operands of the opcodes. To access it, use

```plaintext
GEN closure get oper(GEN C)
```

- \[ z[4] \] points to a t_VEC which hold the data referenced by the pushgen opcodes, which can be t_CLOSURE, and in particular inline closures. To access it, use

```plaintext
GEN closure get data(GEN C)
```
• \( z[5] \) points to a \texttt{t_VEC} which hold extra data needed for error-reporting and debugging. See Section 1.1.1 for details. To access it, use

\[
\text{GEN closure_get_dbg(GEN C)}
\]

Additionally, for functions and true closures,

• \( z[6] \) usually points to a \texttt{t_VEC} with two components which are \texttt{t_STR}. The first one displays the list of arguments of the closure without the enclosing parentheses, the second one the GP code of the function at the right of the \texttt{-} \texttt{t} token. They are used to display the closure, either in implicit or explicit form. However for closures that were not generated from GP code, \( z[6] \) can point to a \texttt{t_STR} instead. To access it, use

\[
\text{GEN closure_get_text(GEN C)}
\]

Additionally, for true closure,

• \( z[7] \) points to a \texttt{t_VEC} which holds the values of all lexical variables defined in the scope the closure was defined. To access it, use

\[
\text{GEN closure_get_frame(GEN C)}
\]

### 1.1.1 Debugging information in closure.

Every \texttt{t_CLOSURE} object \( z \) has a component \( \text{dbg}=z[5] \) which hold extra data needed for error-reporting and debugging. The object \( \text{dbg} \) is a \texttt{t_VEC} with 3 components:

- \( \text{dbg}[1] \) is a \texttt{t_VECSMALL} of the same length than \( z[3] \). For each opcode, it holds the position of the corresponding GP source code in the strings stored in \( z[6] \) for function or true closures, positive indices referring to the second strings, and negative indices referring to the first strings, the last element being indexed as \(-1\). For inline closures, the string of the parent function or true closure is used instead.

- \( \text{dbg}[2] \) is a \texttt{t_VECSMALL} that lists opcodes index where new lexical local variables are created. The value \( 0 \) denotes the position before the first offset and variables created by the prototype code \texttt{V}.

- \( \text{dbg}[3] \) is a \texttt{t_VEC} of \texttt{t_VECSMALLs} that give the list of \texttt{entree*} of the lexical local variables created at a given index in \( \text{dbg}[2] \).

### 1.2 The type \texttt{t_LIST}.

This type needs to go through various hoops to support GP’s inconvenient memory model. Don’t use \texttt{t_LISTs} in pure library mode, reimplement ordinary lists! This dynamic type is implemented by a \texttt{GEN} of length 3: two codewords and a vector containing the actual entries. In a normal setup (a finished list, ready to be used),

- the vector is malloc’ed, so that it can be realloc’ed without moving the parent \texttt{GEN}.

- all the entries are clones, possibly with cloned subcomponents; they must be deleted with \texttt{gunclone_deep}, not \texttt{gunclone}.

The following macros are proper lvalues and access the components

\[
\text{long list_nmax(GEN L): current maximal number of elements. This grows as needed.}
\]
GEN list_data(GEN L): the elements. If $v = \text{list\_data}(L)$, then either $v$ is NULL (empty list) or $l = \lg(v)$ is defined, and the elements are $v[1], \ldots, v[l-1]$.

In most gerepile scenarios, the list components are not inspected and a shallow copy of the malloc'ed vector is made. The functions gclone, copy_bin_canon are exceptions, and make a full copy of the list.

The main problem with lists is to avoid memory leaks; in the above setup, a statement like $a = \text{List}(1)$ would already leak memory, since $\text{List}(1)$ allocates memory, which is cloned (second allocation) when assigned to $a$; and the original list is lost. The solution we implemented is

- to create anonymous lists (from $\text{List}$, gtolist, concat or vecsort) entirely on the stack, not as described above, and to set list_nmax to 0. Such a list is not yet proper and trying to append elements to it fails:

```plaintext
? listput(List(),1)
*** variable name expected: listput(List(),1)
*** ^----------------
```

If we had been malloc'ing memory for the $\text{List}([1,2,3])$, it would have leaked already.

- as soon as a list is assigned to a variable (or a component thereof) by the GP evaluator, the assigned list is converted to the proper format (with list_nmax set) previously described.

GEN listcopy(GEN L) return a full copy of the t_LIST L, allocated on the stack (hence list_nmax is 0). Shortcut for gcopy.

GEN mklistcopy(GEN x) returns a list with a single element $x$, allocated on the stack. Used to implement most cases of gtolist (except vectors and lists).

A typical low-level construct:

```plaintext
long l;
/* assume L is a t_LIST */
L = list_data(L); /* discard t_LIST wrapper */
l = L? lg(L): 1;
for (i = 1; i < l; i++) output( gel(L, i) );
for (i = 1; i < l; i++) gel(L, i) = gclone( ... );
```

1.2.1 Maps as Lists.

GP’s maps are implemented on top of t_LISTs so as to benefit from their peculiar memory models. Lists thus come in two subtypes: t_LIST_RAW (actual lists) and t_LIST_MAP (a map).

GEN mklist_typ(long t) create a list of subtype $t$. GEN mklist(void) is an alias for

```plaintext
mklist_typ(t_LIST_RAW);
```

and GEN mkmap(void) is an alias for

```plaintext
mklist_typ(t_LIST_MAP);
```

long list_typ(GEN L) return the list subtype, either t_LIST_RAW or t_LIST_MAP.

void listpop(GEN L, long index) as listpop0, assuming that $L$ is a t_LIST_RAW.

GEN listput(GEN list, GEN object, long index) as listput0, assuming that $L$ is a t_LIST_RAW.
1.3 Protection of noninterruptible code.

GP allows the user to interrupt a computation by issuing SIGINT (usually by entering control-C) or SIGALRM (usually using alarm()). To avoid such interruption to occur in section of code which are not reentrant (in particular malloc and free) the following mechanism is provided:

`BLOCK_SIGINT_START()` Start a noninterruptible block code. Block both SIGINT and SIGALRM.

`BLOCK_SIGALRM_START()` Start a noninterruptible block code. Block only SIGALRM. This is used in the SIGINT handler itself to delay an eventual pending alarm.

`BLOCK_SIGINT_END()` End a noninterruptible block code

The above macros make use of the following global variables:

`PARI_SIGINT_block`: set to 1 (resp. 2) by `BLOCK_SIGINT_START` (resp. `BLOCK_SIGALRM_START`).

`PARI_SIGINT_pending`: Either 0 (no signal was blocked), SIGINT (SIGINT was blocked) or SIGALRM (SIGALRM was blocked). This need to be set by the signal handler.

Within a block, an automatic variable `int block` is defined which records the value of `PARI_SIGINT_block` when entering the block.

1.3.1 Multithread interruptions.

To support multithreaded programs, `BLOCK_SIGINT_START` and `BLOCK_SIGALRM_START` call `MT_SIGINT_BLOCK(block)`, and `BLOCK_SIGINT_END` calls `MT_SIGINT_UNBLOCK(block)`.

`MT_SIGINT_BLOCK` and `MT_SIGINT_UNBLOCK` are defined by the multithread engine. They can call the following public functions defined by the multithread engine.

void `mt_sigint_block(void)`

void `mt_sigint_unblock(void)`

In practice this mechanism is used by the POSIX thread engine to protect against asynchronus cancellation.
1.4 $\mathbb{F}_p$ field for small primes $l$.

Let $l > 2$ be a prime ulong. A $\mathbb{F}_{l^2}$ is an element of the finite field $\mathbb{F}_p$ represented (currently) by a $\text{Fl}_x$ of degree at most 1 modulo a polynomial of the form $x^2 - D$ for some non square $0 \leq D < p$. Below $\pi$ denotes the pseudo inverse of $p$, see $\text{Fl}_1\_mul\_pre$.

```c
int Fl2_equal1(GEN x) return 1 if $x = 1$, else return 0.
GEN Fl2_mul_pre(GEN x, GEN y, ulong D, ulong p, ulong pi) return $xy$.
GEN Fl2_sqr_pre(GEN x, ulong D, ulong p, ulong pi) return $x^2$.
GEN Fl2_inv_pre(GEN x, ulong D, ulong p, ulong pi) return $x^{-1}$.
GEN Fl2_pow_pre(GEN x, GEN n, ulong D, ulong p, ulong pi) return $x^n$.
GEN Fl2_sqrtn_pre(GEN a, GEN n, ulong D, ulong p, ulong pi, GEN *zeta) $n$-th root, as $\text{Flx}_{q}\_sqrtn$.
GEN Fl2_norm_pre(GEN x, GEN n, ulong D, ulong p, ulong pi) return the norm of $x$.
GEN Flx.Fl2.eval_pre(GEN P, GEN x, ulong D, ulong p, ulong pi) return $P(x)$.
```

1.5 Public functions useless outside of GP context.

These functions implement GP functionality for which the C language or other libpari routines provide a better equivalent; or which are so tied to the gp interpreter as to be virtually useless in libpari. Some may be generated by gp2c. We document them here for completeness.

1.5.1 Conversions.

- `GEN toser_i(GEN x)` internal shallow function, used to implement automatic conversions to power series in GP (as in $\cos(x)$). Converts a `t_POL` or a `t_RFRAC` to a `t_SER` in the same variable and precision `precdl` (the global variable corresponding to `seriesprecision`). Returns $x$ itself for a `t_SER`, and `NULL` for other argument types. The fact that it uses a global variable makes it awkward whenever you’re not implementing a new transcendental function in GP. Use `RgX_to_ser` or `rfrac_to_ser` for a fast clean alternative to `gtoser`.

- `GEN listinit(GEN x)` a `t_LIST` (from `List` or `Map`) may exist in two different forms due to GP memory model:
  - an ordinary read-only copy on the PARI stack (as produced by `gtolist` or `gtomap`) to which one may not assign elements (`listput` will fail) unless the list is empty.
  - a feature-complete GP list using (malloc’ed) blocks to allow dynamic insertions. An empty list is automatically promoted to this status on first insertion.

The `listinit` function creates a copy of existing `t_SER x` and makes sure it is of the second kind. Variants of this are automatically called by `gp` when assigning a `t_LIST` to a GP variable; the mechanism avoid memory leaks when creating a constant list, e.g. `List([1,2,3])` (read-only), without assigning it to a variable. Whereas after $L = \text{List}([1,2,3])$ (GP list), we keep a pointer to the object and may delete it when $L$ goes out of scope.

This `libpari` function allows `gp2c` to simulate this process by generating `listinit` calls at appropriate places.
1.5.2 Output.

void print0(GEN g, long flag) internal function underlying the print GP function. Prints the entries of the t_VEC g, one by one, without any separator; entries of type t_STR are printed without enclosing quotes. flag is one of f_RAW, fPRETTYMAT or f_TEX, using the current default output context.

void out_print0(PariOUT *out, const char *sep, GEN g, long flag) as print0, using output context out and separator sep between successive entries (no separator if NULL).

void printsep(const char *s, GEN g, long flag) out_print0 on pariOut followed by a newline.

void printsep1(const char *s, GEN g, long flag) out_print0 on pariOut.

char* pari_sprint0(const char *s, GEN g, long flag) displays s, then print0(g, flag).

void print(GEN g) equivalent to print0(g, f_RAW), followed by a \n then an fflush.

void printp(GEN g) equivalent to print0(g, fPRETTYMAT), followed by a \n then an fflush.

void print1(GEN g) as above, without the \n. Use pari_printf or output instead.

void printtex(GEN g) equivalent to print0(g, t_TEX), followed by a \n then an fflush. Use GENtoTeXstr and pari_printf instead.

void write0(const char *s, GEN g)
void write1(const char *s, GEN g) use fprintf

void writetex(const char *s, GEN g) use GENtoTeXstr and fprintf.

void printf0(GEN fmt, GEN args) use pari_printf.

GEN strprintf(GEN fmt, GEN args) use pari_sprintf.

1.5.3 Input.

gp’s input is read from the stream pari_infile, which is changed using

FILE* switchin(const char *name)

Note that this function is quite complicated, maintaining stacks of files to allow smooth error recovery and gp interaction. You will be better off using gp_read_file.

1.5.4 Control flow statements.

GEN break0(long n). Use the C control statement break. Since break(2) is invalid in C, either rework your code or use goto.

GEN next0(long n). Use the C control statement continue. Since continue(2) is invalid in C, either rework your code or use goto.

GEN return0(GEN x). Use return!

void error0(GEN g). Use pari_err(e_USER,)
void warning0(GEN g). Use pari_warn(e_USER,
1.5.5 Accessors.

GEN vecslic0(GEN A, long a, long b) implements \( A[a..b] \).

GEN matslic0(GEN A, long a, long b, long c, long d) implements \( A[a..b,c..d] \).

1.5.6 Iterators.

GEN apply0(GEN f, GEN A) gp wrapper calling genapply, where \( f \) is a t_CLOSURE, applied to \( A \). Use genapply or a standard C loop.

GEN select0(GEN f, GEN A) gp wrapper calling genselect, where \( f \) is a t_CLOSURE selecting from \( A \). Use genselect or a standard C loop.

GEN vecapply(void *E, GEN (*f)(void* E, GEN x), GEN x) implements \( [a(x)|x<-b] \).

GEN veccatapply(void *E, GEN (*f)(void* E, GEN x), GEN x) implements \( \text{concat}([a(x)|x<-b]) \) which used to implement \( [a0(x,y)|x<-b;y<-c(b)] \) which is equal to \( \text{concat}([[a0(x,y)|y<-c(b)]|x<-b]) \).

GEN vecselect(void *E, long (*f)(void* E, GEN x), GEN A) implements \( [x<-b,c(x)] \).

GEN vecselapply(void *Epred, long (*pred)(void* E, GEN x), void *Efun, GEN (*fun)(void* E, GEN x), GEN A) implements \( [a(x)|x<-b,c(x)] \).

1.5.7 Local precision.

These functions allow to change realprecision locally when calling the GP interpreter.

void push_localprec(long p) set the local precision to \( p \).

void push_localbitprec(long b) set the local precision to \( b \) bits.

void pop_localprec(void) reset the local precision to the previous value.

long get_localprec(void) returns the current local precision.

long get_localbitprec(void) returns the current local precision in bits.

void localprec(long p) trivial wrapper around push_localprec (sanity checks and convert from decimal digits to a number of codewords). Use push_localprec.

void localbitprec(long p) trivial wrapper around push_localbitprec (sanity checks). Use push_localbitprec.

These two function are used to implement getlocalprec and getlocalbitprec for the GP interpreter and essentially return their argument (the current dynamic precision, respectively in bits or as a prec word count):

long getlocalbitprec(long bit)

long getlocalprec(long prec)
1.5.8 Functions related to the GP evaluator.

The prototype code C instructs the GP compiler to save the current lexical context (pairs made of a lexical variable name and its value) in a GEN, called pack in the sequel. This pack can be used to evaluate expressions in the corresponding lexical context, providing it is current.

GEN localvars_read_str(const char *s, GEN pack) evaluate the string s in the lexical context given by pack. Used by geval_gp in GP to implement the behavior below:

```c
? my(z=3); eval("z=z^2"); z
\%1 = 9
```

long localvars_find(GEN pack, entree *ep) does pack contain a pair whose variable corresponds to ep? If so, where is the corresponding value? (returns an offset on the value stack).

1.5.9 Miscellaneous.

char* os_getenv(const char *s) either calls getenv, or directly return NULL if the libc does not provide it. Use getenv.

sighandler_t os_signal(int sig, pari_sighandler_t fun) after a

```c
typedef void (*pari_sighandler_t)(int);
```

(private type, not exported). Installs signal handler fun for signal sig, using sigaction with flag SA_NODEFER. If sigaction is not available use signal. If even the latter is not available, just return SIG_IGN. Use sigaction.

1.6 Embedded GP interpreter.

These function provide a simplified interface to embed a GP interpreter in a program.

```c
void gp_embedded_init(long rsize, long vsize) Initialize the GP interpreter (like pari_init does) with parisize=rsize rsize and parisizemax=vsize.

char * gp_embedded(const char *s) Evaluate the string s with GP and return the result as a string, in a format similar to what GP displays (with the history index). The resulting string is allocated on the PARI stack, so subsequent call to gp_embedded will destroy it.
```
1.7 Readline interface.

Code which wants to use libpari readline (such as the Jupyter notebook) needs to do the following:

```c
#include <readline.h>
#include <paripriv.h>
pari_rl_interface S;
...
parsi_use_readline(S);
```

The variable `S`, as initialized above, encapsulates the libpari readline interface. (And allow us to move gp's readline code to libpari without introducing a mandatory dependency on readline in libpari.) The following functions then become available:

```c
char** pari_completion_matches(pari_rl_interface *pS, const char *s, long pos, long *wordpos) given a command string `s`, where the cursor is at index `pos`, return an array of completion matches.

If `wordpos` is not `NULL`, set `*wordpos` to the index for the start of the expression we complete.
```

```c
char** pari_completion(pari_rl_interface *pS, char *text, int start, int end) the low-level completer called by pari_completion_matches. The following wrapper

```c
char**
gp_completion(char *text, int START, int END)
{
    return pari_completion(&S, text, START, END);
}
```

is a valid value for `rl_attempted_completion_function`.

1.8 Constructors called by pari_init functions.

```c
void pari_init_buffers()
void pari_init_compiler()
void pari_init_defaults()
void pari_init_evaluator()
void pari_init_files()
void pari_init_floats()
void pari_init_graphics()
void pari_init_homedir()
void pari_init_parser()
void pari_init_paths()
void pari_init_primetab()
void pari_init_rand()
void pari_init_seadata()
```
1.9 Destructors called by pari_close.

void pari_close_compiler()
void pari_close_evaluator()
void pari_close_files()
void pari_close_floats()
void pari_close_homedir()
void pari_close_mf()
void pari_close_parser()
void pari_close_paths()
void pari_close_primes()

1.10 Constructors and destructors used by the pthreads interface.

- Called by pari_thread_close

void pari_thread_close_files()
Chapter 2: Regression tests, benches

This chapter documents how to write an automated test module, say fun, so that make test-fun executes the statements in the fun module and times them, compares the output to a template, and prints an error message if they do not match.

- Pick a new name for your test, say fun, and write down a GP script named fun. Make sure it produces some useful output and tests adequately a set of routines.

- The script should not be too long: one minute runs should be enough. Try to break your script into independent easily reproducible tests, this way regressions are easier to debug; e.g. include setrand(1) statement before a randomized computation. The expected output may be different on 32-bit and 64-bit machines but should otherwise be platform-independent. If possible, the output shouldn’t even depend on sizeof(long); using a realprecision that exists on both 32-bit and 64-bit architectures, e.g. \p 38 is a good first step. You can use sizebyte(0)==16 to detect a 64-bit architecture and sizebyte(0)==8 for 32-bit.

- Dump your script into src/test/in/ and run Configure.

- make test-fun now runs the new test, producing a [BUG] error message and a .dif file in the relevant object directory Oxxx. In fact, we compared the output to a nonexistent template, so this must fail.

- Now

  patch -p1 < Oxxx/fun.dif

  generates a template output in the right place src/test/32/fun, for instance on a 32-bit machine.

- If different output is expected on 32-bit and 64-bit machines, run the test on a 64-bit machine and patch again, thereby producing src/test/64/fun. If, on the contrary, the output must be the same (preferred behavior!), make sure the output template land in the src/test/32/ directory which provides a default template when the 64-bit output file is missing; in particular move the file from src/test/64/ to src/test/32/ if the test was run on a 64-bit machine.

- You can now re-run the test to check for regressions: no [BUG] is expected this time! Of course you can at any time add some checks, and iterate the test / patch phases. In particular, each time a bug in the fun module is fixed, it is a good idea to add a minimal test case to the test suite.

- By default, your new test is now included in make test-all. If it is particularly annoying, e.g. opens tons of graphical windows as make test-ploth or just much longer than the recommended minute, you may edit config/get_tests and add the fun test to the list of excluded tests, in the test_extra_out variable.

- You can run a subset of existing tests by using the following idiom:

  cd Oxxx    # call from relevant build directory
  make TESTS="1funttype lfun gamma" test-all

will run the 1funttype, lfun and gamma tests. This produces a combined output whereas the alternative
make test-lfuntype test-lfun test-gamma
would not.

• By default, the test is run on both the gp-sta and gp-dyn binaries, making it twice as slow.
If the test is somewhat long, it can be annoying; you can restrict to one binary only using the
stastest-all or dyntest-all targets. Both accept the TESTS argument seen above.
make test-lfuntype test-lfun gamma
would not.

• Finally, the get_tests script also defines the recipe for make bench timings, via the variable
test_basic. A test is included as fun or fun_n, where n is an integer \leq 1000; the latter means
that the timing is weighted by a factor n/1000. (This was introduced a long time ago, when the
nfields bench was so much slower than the others that it hid slowdowns elsewhere.)

2.1 Functions for GP2C.

2.1.1 Functions for safe access to components.
These functions return the address of the requested component after checking it is actually
valid. This is used by GP2C -C.
GEN* safegel(GEN x, long l), safe version of gel(x,l) for t_VEC, t_COL and t_MAT.
long* safeel(GEN x, long l), safe version of x[l] for t_VECSMALL.
GEN* safelistel(GEN x, long l) safe access to t_LIST component.
GEN* safegcoeff(GEN x, long a, long b) safe version of gcoeff(x,a, b) for t_MAT.
Chapter 3:  
Parallelism

PARI provides an abstraction, herafter called the MT engine, for doing parallel computations. The exact same high level routines are used whether the underlying communication protocol is POSIX threads or MPI and they behave differently depending on how `libpari` was configured, specifically on `Configure`'s `--mt` option. Sequential computation is also supported (no `--mt` argument) which is helpful for debugging newly written parallel code. The final section in this chapter comments a complete example.

3.1 The PARI multithread interface.

```c
void mt_queue_start(struct pari_mt *pt, GEN worker)
```
Let `worker` be a `t_CLOSURE` object of arity 1. Initialize the opaque structure `pt` to evaluate `worker` in parallel, using `nbthreads` threads. This allocates data in various ways, e.g., on the PARI stack or as malloc'ed objects: you may not collect garbage on the PARI stack starting from an earlier `avma` point until the parallel computation is over, it could destroy something in `pt`. All resources allocated outside the PARI stack are freed by `mt_queue_end`.

```c
void mt_queue_start_lim(struct pari_mt *pt, GEN worker, long lim)
```
where `lim` is an upper bound on the number of tasks to perform. Concretely the number of threads is the minimum of `lim` and `nbthreads`. The values 0 and 1 of `lim` are special:

- 0: no limit, equivalent to `mt_queue_start` (use `nbthreads` threads).
- 1: no parallelism, evaluate the tasks sequentially.

```c
void mt_queue_submit(struct pari_mt *pt, long taskid, GEN task)
```
submit `task` to be evaluated by `worker`; use `task = NULL` if no further task needs to be submitted. The parameter `taskid` is attached to the `task` but not used in any way by the `worker` or the MT engine, it will be returned to you by `mt_queue_get` together with the result for the task, allowing to match up results and submitted tasks if desired. For instance, if the tasks $(t_1, \ldots, t_m)$ are known in advance, stored in a vector, and you want to recover the evaluation results in the same order as in that vector, you may use consecutive integers $1, \ldots, m$ as `taskids`. If you do not care about the ordering, on the other hand, you can just use `taskid = 0` for all tasks.

The `taskid` parameter is ignored when `task` is `NULL`. It is forbidden to call this function twice without an intervening `mt_queue_end`.

```c
GEN mt_queue_get(struct pari_mt *pt, long *taskid, long *pending)
```
return `NULL` until `mt_queue_submit` has submitted tasks for the required number (`nbthreads`) of threads; then return the result of the evaluation by `worker` of one of the previously submitted tasks, in random order. Set `pending` to the number of remaining pending tasks: if this is 0 then no more tasks are pending and it is safe to call `mt_queue_end`. Set `*taskid` to the value attached to this task by `mt_queue_submit`, unless the `taskid` pointer is `NULL`. It is forbidden to call this function twice without an intervening `mt_queue_submit`.

```c
void mt_queue_end(struct pari_mt *pt)
```
end the parallel execution and free resources attached to the opaque `pari_mt` structure. For instance malloc'ed data; in the `pthreads` interface, it would
destroy mutex locks, condition variables, etc. This must be called once there are no longer pending tasks to avoid leaking resources; but not before all tasks have been processed else crashes will occur.

long mt_nbthreads(void) return the effective number of parallel threads that would be started by mt_queue_start if it has been called in place of mt_nbthreads.

### 3.2 Technical functions required by MPI.

The functions in this section are needed when writing complex independent programs in order to support the MPI MT engine, as more flexible complement/variants of pari_init and pari_close.

void mt_broadcast(GEN code): do nothing unless the MPI threading engine is in use. In that case, evaluates the closure code on all secondary nodes. This can be used to change the state of all MPI child nodes, e.g., in gpinstall run in the main thread, which allows all nodes to use the new function.

void pari_mt_init(void) when using MPI, it is often necessary to run initialization code on the child nodes after PARI is initialized. This is done by calling successively:

- pari_init_opts with the flag INIT_noIMTm: this initializes PARI, but not the MT engine;
- the required initialization code;
- pari_mt_init to initialize the MT engine. Note that under MPI, this function returns on the master node but enters slave mode on the child nodes. Thus it is no longer possible to run initialization code on the child nodes.

void pari_mt_close(void) when using MPI, calling pari_close terminates the MPI execution environment and it will not be possible to restart it. If this is undesirable, call pari_close_opts with the flag INIT_noIMTm instead of pari_close: this closes PARI without terminating the MPI execution environment. You may later call pari_mt_close to terminate it. It is an error for a program to end without terminating the MPI execution environment.

### 3.3 A complete example.

We now proceed to an example exhibiting complex features of this interface, in particular showing how to generate a valid worker. Explanations and details follow.

```c
#include <pari/pari.h>
GEN
Cworker(GEN d, long kind) { return kind? det(d): Z_factor(d); }
int
main(void)
{
    long i, taskid, pending;
    GEN M,N1,N2, in,out, done;
    struct pari_mt pt;
    entree ep = {"_worker",0,(void*)Cworker,20,"GL",""};
    /* initialize PARI, postponing parallelism initialization */
```
We start from some arbitrary C function `Cworker` and create an `entree` summarizing all that GP would need to know about it, in particular

- a GP name `worker`; the leading _ is not necessary, we use it as a namespace mechanism grouping private functions;
- the name of the C function;
- and its prototype, see `install` for an introduction to Prototype Codes.

The other three arguments (0, 20 and "") are required in an `entree` but not useful in our simple context: they are respectively a valence (0 means “nothing special”), a help section (20 is customary for internal functions which need to be exported for technical reasons, see ??20), and a help text (no help).

Then we initialize the MT engine; doing things in this order with a two part initialization ensures that nodes have access to our `Cworker`. We convert the `ep` data to a `t_CLOSURE` using `strtofunction`, which provides a valid `worker` to `mt_queue_start`. This creates a parallel evaluation queue `mt`, and we proceed to submit all tasks, recording all results. Results are stored in the right order by making good use of the `taskid` label, although we have no control over when each result is returned. We finally free all resources attached to the `mt` structure. If needed, we could have collected all garbage on the PARI stack using `gerepilecopy` on the `out` array and gone on working instead of quitting.

Note the argument passing convention for `Cworker`: the task consists of a single vector containing all arguments as `GENs`, which are interpreted according to the function prototype, here GL so the first argument is left as is and the second one is converted to a long integer. In more complicated situations, this second (and possibly further) argument could provide arbitrary evaluation contexts. In this example, we just used it as a flag to indicate the kind of evaluation expected on the data: integer factorization (0) or matrix determinant (1).

Note also that
gel(out, taskid) = mt_queue_get(&mt, &taskid, &pending);

Instead of our use of a temporary done would have undefined behaviour (taskid may be uninitialized in the left hand side).
**Index**

*SomeWord* refers to PARI-GP concepts.
*SomeWord* is a PARI-GP keyword.
*SomeWord* is a generic index entry.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply0</td>
<td>BLOCK_SIGALRM_START</td>
<td>closure</td>
<td>closure</td>
<td>error0</td>
</tr>
<tr>
<td></td>
<td>BLOCK_SIGINT_END</td>
<td>closure_arity</td>
<td>closure_get_code</td>
<td>closure_get_data</td>
</tr>
<tr>
<td></td>
<td>BLOCK_SIGINT_START</td>
<td>closure_get_dbg</td>
<td>closure_get_frame</td>
<td>closure_get_oper</td>
</tr>
<tr>
<td></td>
<td></td>
<td>closure_get_text</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>get_localbitprec</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>get_localprec</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>geval_gp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>gpinstall</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>gp_embedded</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>gp_embedded_init</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>gp_read_file</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>guncclone</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>guncclone_deep</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INIT_noIMTm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>listcopy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>listinit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>listpop</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>listput</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list_data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list_mmax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list_typ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>localbitprec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>localprec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>localvars_find</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>localvars_read_str</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mapdomain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mapdomain_shallow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matomt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matomt_shallow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>matslice0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mklist</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mklistcopy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mklist_typ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mkmap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_broadcast</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_nbthreads</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_queue_end</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_queue_get</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_queue_start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_queue_start_lim</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_queue_sub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MT_SIGINT_BLOCK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mt_sigint_block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MT_SIGINT_UNBLOCK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>F12_equal1</td>
<td>genapply</td>
<td>getenv</td>
<td>getlocalbitprec</td>
<td>getlocalprec</td>
</tr>
<tr>
<td>F12_inv</td>
<td>genselct</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12_mul</td>
<td>GENtoTeXstr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12_norm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12_pow</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12_sqrtn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F12_sqr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fix_F12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21
N
next0 ........................................ 10

O
os_getenv ................................... 12
os_signal .................................. 12
output ...................................... 10
out_print0 .................................. 10

P
pariOut .................................... 10
pari_close .................................. 18
pari_close_compiler ......................... 13
pari_close_evaluator ........................ 13
pari_close_files ............................ 13
pari_close_floats ........................... 14
pari_close_homemdir ........................ 14
pari_close_mf ................................ 14
pari_close_opts ............................. 18
pari_close_parser ........................... 14
pari_close_paths ............................ 14
pari_close_primes ........................... 14
pari_close_floats ........................... 14
pari_completion ............................. 12
pari_completion_matches ..................... 12, 13
pari_infile .................................. 10
pari_init_buffers ............................ 13
pari_init_compiler ........................... 13
pari_init_defaults ........................... 13
pari_init_evaluator .......................... 13
pari_init_files .............................. 13
pari_init_floats ............................. 13
pari_init_graphics ........................... 13
pari_init_homemdir ........................... 13
pari_init_opt ....................... 18
pari_init_parser ............................. 13
pari_init_paths .............................. 13
pari_init_primetab ........................... 13
pari_init_rand ............................... 13
pari_init_seadata ............................ 13
pari_mt_close ............................... 18
pari_mt_init ................................. 18
pari_putchar ................................. 10
PARI_SIGINT_block ........................... 8
PARI_SIGINT_pending ......................... 8
pari_sprint0 .................................. 10
pari_sprintf .................................. 10
pari_thread_close ............................ 14
pari_thread_close_files ..................... 14
pop_localprec ............................... 11
print ........................................ 10
print0 ....................................... 9, 10
print1 ........................................ 10
printf ....................................... 10
printf ....................................... 10
printsep ..................................... 10
printtex ..................................... 10
push_localbitprec ........................... 11
push_localprec ............................... 11

R
return0 ..................................... 10
rfrac_to_ser ................................ 9
RgX_to_ser ................................... 9
rl_attempted_completion_function ........... 13

S
safeel ....................................... 16
safegcoeff .................................. 16
safegel ...................................... 16
safelistel ................................... 16
SA_NODEFER .................................. 12
select0 ....................................... 11
sigaction ................................... 12
signal ......................................... 12
SIG_IGN ....................................... 12
strprintf .................................... 10
switchin ..................................... 10

T
toser_i ....................................... 9
t_CLOSURE .................................. 5
t_LIST ....................................... 6

V
vecapply .................................... 11
veccatapply .................................. 11
veccatapply .................................. 11
vecselect .................................... 11
vecslice0 .................................... 10

W
warning0 ..................................... 10
write0 ....................................... 10