Parallel computation

in

PARI/GP

(version 2.12.1)

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/`

Copyright © 2000–2019 The PARI Group

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions, or translations, of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

PARI/GP is Copyright © 2000–2019 The PARI Group

PARI/GP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY WHATSOEVER.

Table of Contents

Chapter 1: Parallel PARI/GP interface 5
1.1 Configuration 5
1.1.1 POSIX threads 5
1.1.2 Message Passing Interface 5
1.2 Concept 6
1.2.1 Resources 6
1.2.2 GP functions 7
1.2.3 PARI functions 7
Chapter 2: Writing code suitable for parallel execution 9
2.1 Exporting global variables 9
2.1.1 Example 1: data 9
2.1.2 Example 2: polynomial variable 9
2.1.3 Example 3: function 10
2.2 Input and output 10
2.3 Using `parfor` and `parforprime` 11
2.4 Sizing parallel tasks 12
2.5 Load balancing 12
Chapter 3: PARI functions 13
3.1 The PARI multithread interface 13
3.2 Technical functions required by MPI 14
3.3 A complete example 14

Chapter 1:
Parallel PARI/GP interface

1.1 Configuration.

This booklet documents the parallel PARI/GP interface. The first chapter describes configuration and basic concepts, the second one explains how to write GP codes suitable for parallel execution, the final one is more technical and describes `libpari` functions. Two multithread interfaces are supported in PARI/GP:

- POSIX threads
- Message passing interface (MPI)

POSIX threads are well-suited for single systems, for instance a personal computer equiped with a multi-core processor, while MPI is used by most clusters. However the parallel GP interface does not depend on the multithread interface: a properly written GP program will work identically with both. The interfaces are mutually exclusive and one must be chosen at configuration time when installing the PARI/GP suite.

1.1.1 POSIX threads.

POSIX threads are selected by passing the flag `--mt=pthread` to `Configure`. The required library (`libpthread`) and header files (`pthread.h`) are installed by default on most Linux system. This option implies `--enable-tls` which builds a thread-safe version of the PARI library. This unfortunately makes the dynamically linked `gp-dyn` binary about 25% slower; since `gp-sta` is only 5% slower, you definitely want to use the latter binary.

You may want to also pass the flag `--time=ftime` to `Configure` so that `gettime` and the GP timer report real time instead of cumulated CPU time. An alternative is to use `getwalltime` in your GP scripts instead of `gettime`.

You can test whether parallel GP support is working with

```
make test-parallel
```

1.1.2 Message Passing Interface.

Configuring MPI is somewhat more difficult, but your MPI installation should include a script `mpicc` that takes care of the necessary configuration. If you have a choice between several MPI implementations, choose OpenMPI.

To configure PARI/GP for MPI, use

```
env CC=mpicc ./Configure --mt=mpi
```

To run a program, you must use the launcher provided by your implementation, usually `mpiexec` or `mpirun`. For instance, to run the program `prog.gp` on 10 nodes, you would use

```
  mpirun -np 10 gp prog.gp
```

(or `mpiexec` instead of `mpirun`). PARI requires at least 3 MPI nodes to work properly. *Caveats:* `mpirun` is not suited for interactive use because it does not provide tty emulation. Moreover, it is not currently possible to interrupt parallel tasks.

You can test parallel GP support (here using 3 nodes) with

```
make test-parallel RUNTEST="mpirun -np 3"
```

1.2 Concept.

In a GP program, the *main thread* executes instructions sequentially (one after the other) and GP provides functions, that execute subtasks in *secondary threads* in parallel (at the same time). Those functions all share the prefix `par`, e.g., `parfor`, a parallel version of the sequential `for`-loop construct.

The subtasks are subject to a stringent limitation, the parallel code must be free of side effects: it cannot set or modify a global variable for instance. In fact, it may not even *access* global variables or local variables declared with `local()`.

Due to the overhead of parallelism, we recommend to split the computation so that each parallel computation requires at least a few seconds. On the other hand, it is generally more efficient to split the computation in small chunks rather than large chunks.

1.2.1 Resources.

The number of secondary threads to use is controlled by `default(nbthreads)`. The default value of nbthreads depends on the mutithread interface.

- POSIX threads: the number of CPU threads, i.e., the number of CPU cores multiplied by the hyperthreading factor. The default can be freely modified.

- MPI: the number of available process slots minus 1 (one slot is used by the master thread), as configured with `mpirun` (or `mpiexec`). E.g., `nbthreads` is 9 after `mpirun -np 10 gp`. It is possible to change the default to a lower value, but increasing it will not work: MPI does not allow to spawn new threads at run time. PARI requires at least 3 MPI nodes to work properly.

The PARI stack size in secondary threads is controlled by `default(threadsize)`, so the total memory allocated is equal to `parisize` + `nbthreads` × `threadsize`. By default, `threadsize` = `parisize`. Setting the `threadsizemax` default allows `threadsize` to grow as needed up to that limit, analogously to the behaviour of `parisize` / `parisizemax`. We strongly recommend to set this parameter since it is very hard to control in advance the amount of memory threads will require: a too small stack size will result in a stack overflow, aborting the computation, and a too large value is very wasteful (since the extra reserved but unneeded memory is multiplied by the number of threads).

1.2.2 GP functions.

GP provides the following functions for parallel operations, please see the documentation of each function for details:

- `parvector`: parallel version of `vector`;
- `parapply`: parallel version of `apply`;
- `parsum`: parallel version of `sum`;
- `parselect`: parallel version of `select`;
- `pareval`: evaluate a vector of closures in parallel;
- `parfor`: parallel version of `for`;
- `parforprime`: parallel version of `forprime`;
- `parforvec`: parallel version of `forvec`;
- `parploth`: parallel version of `ploth`.

1.2.3 PARI functions. The low-level `libpari` interface for parallelism is documented in the *Developer's guide to the PARI library*.

Chapter 2:
Writing code suitable for parallel execution

2.1 Exporting global variables.

When parallel execution encounters a global variable, say V, an error such as the following is reported:

```
*** parapply: mt: please use export(V)
```

A global variable is not visible in the parallel execution unless it is explicitly exported. This may occur in a number of contexts.

2.1.1 Example 1: data.

```
? V = [2^256 + 1, 2^193 - 1];
? parvector(#V, i, factor(V[i]))
  *** parvector: mt: please use export(V).
```

The problem is fixed as follows:

```
? V = [2^256 + 1, 2^193 - 1];
? export(V)
? parvector(#V, i, factor(V[i]))
```

The following short form is also available, with a different semantic:

```
? export(V = [2^256 + 1, 2^193 - 1]);
? parvector(#V, i, factor(V[i]))
```

In the latter case the variable V does not exist in the main thread, only in parallel threads.

2.1.2 Example 2: polynomial variable.

```
? f(n) = bnfinit(x^n-2).no;
? parapply(f, [1..50])
  *** parapply: mt: please use export(x).
```

You may fix this as in the first example using `export` but here there is a more natural solution: use the polynomial indeterminate `'x` instead the global variable `x` (whose value is `'x` on startup, but may or may no longer be `'x` at this point):

```
? f(n) = bnfinit('x^n-2).no;
```

or alternatively

```
? f(n) = my(x='x); bnfinit(x^n-2).no;
```

which is more readable if the same polynomial variable is used several times in the function.

2.1.3 Example 3: function.

```
? f(a) = bnfinit('x^8-a).no;
? g(a,b) = parsum(i = a, b, f(i));
? g(37,48)
  *** parsum: mt: please use export(f).
? export(f)
? g(37,48)
%4 = 81
```

Note that `export`(v) freezes the value of v for parallel execution at the time of the export: you may certainly modify its value later in the main thread but you need to re-export v if you want the new value to be used in parallel threads. You may export more than one variable at once, e.g., `export`(a, b, c) is accepted. You may also export *all* variables with dynamic scope (all global variables and all variables declared with `local`) using `exportall()`. Although convenient, this may be wasteful if most variables are not meant to be used from parallel threads. We recommend to

- use `exportall` in the `gp` interpreter interactively, while developping code;
- `export` a function meant to be called from parallel threads, just after its definition;
- use $v =$ *value*; `export`(v) when the value is needed both in the main thread and in secondary threads;
- use `export`($v =$ *value*) when the value is not needed in the main thread.

In the two latter forms, v should be considered read-only. It is actually read-only in secondary threads, trying to change it will raise an exception:

```
  ***   mt: attempt to change exported variable 'v'.
```

You *can* modify it in the main thread, but it must be exported again so that the new value is accessible to secondary threads: barring a new `export`, secondary threads continue to access the old value.

2.2 Input and output.

If your parallel code needs to write data to files, split the output in as many files as the number of parallel computations, to avoid concurrent writes to the same file, with a high risk of data corruption. For example a parallel version of

```
? f(a) = write("bnf",bnfinit('x^8-a));
? for (a = 37, 48, f(a))
```

could be

```
? f(a) = write(Str("bnf-",a), bnfinit('x^8-a).no);
? export(f);
? parfor(i = 37, 48, f(i))
```

which creates the files `bnf-37` to `bnf-48`. Of course you may want to group these file in a subdirectory, which must be created first.

2.3 Using `parfor` and `parforprime`.

`parfor` and `parforprime` are the most powerful of all parallel GP functions but, since they have a different interface than `for` or `forprime`, sequential code needs to be adapted. Consider the example

```
for(i = a, b,
  my(c = f(i));
  g(i,c));
```

where `f` is a function without side effects. This can be run in parallel as follows:

```
parfor(i = a, b,
  f(i),
  c,     /* the value of f(i) is assigned to c */
  g(i,c));
```

For each i, $a \leq i \leq b$, in random order, this construction assigns `f(i)` to (lexically scoped, as per `my`) variable c, then calls `g`(i, c). Only the function `f` is evaluated in parallel (in secondary threads), the function `g` is evaluated sequentially (in the main thread). Writing `c = f(i)` in the parallel section of the code would not work since the main thread would then know nothing about c or its content. The only data sent from the main thread to secondary threads are f and the index i, and only c (which is equal to $f(i)$) is returned from the secondary thread to the main thread.

The following function finds the index of the first component of a vector V satisfying a predicate, and 0 if none satisfies it:

```
parfirst(pred, V) =
{
  parfor(i = 1, #V,
    pred(V[i]),
    cond,
    if (cond, return(i)));
  return(0);
}
```

This works because, if the second expression in `parfor` exits the loop via `break` / `return` at index i, it is guaranteed that all indexes $< i$ are also evaluated and the one with smallest index is the one that triggers the exit. See `??parfor` for details.

The following function is similar to `parsum`:

```
myparsum(a, b, expr) =
{ my(s = 0);

  parfor(i = a, b,
    expr(i),
    val,
    s += val);
  return(s);
}
```

2.4 Sizing parallel tasks.

Dispatching tasks to parallel threads takes time. To limit overhead, split the computation so that each parallel task requires at least a few seconds. Consider the following sequential example:

```
thuemorse(n) = (-1)^n * hammingweight(n);
sum(n = 1, 2*10^6, thuemorse(n) / n * 1.)
```

It is natural to try

```
export(thuemorse);
parsum(n = 1, 2*10^6, thuemorse(n) / n * 1.)
```

However, due to the overhead, this will not be faster than the sequential version; in fact it will likely be *slower*. To limit overhead, we group the summation by blocks:

```
parsum(N = 1, 20, sum(n = 1+(N-1)*10^5, N*10^5, thuemorse(n) / n*1.))
```

Try to create at least as many groups as the number of available threads, to take full advantage of parallelism. Since some of the floating point additions are done in random order (the ones in a given block occur successively, in deterministic order), it is possible that some of the results will differ slightly from one run to the next.

2.5 Load balancing.

If the parallel tasks require varying time to complete, it is preferable to perform the slower ones first, when there are more tasks than available parallel threads. Instead of

```
parvector(36, i, bnfinit('x^i - 2).no)
```

doing

```
parvector(36, i, bnfinit('x^(37-i) - 2).no)
```

will be faster if you have fewer than 36 threads. Indeed, `parvector` schedules tasks by increasing i values, and the computation time increases steeply with i. With 18 threads, say:

- in the first form, thread 1 handles both $i = 1$ and $i = 19$, while thread 18 will likely handle $i = 18$ and $i = 36$. In fact, it is likely that the first batch of tasks $i \leq 18$ runs relatively quickly, but that none of the threads handling a value $i > 18$ (second task) will have time to complete before $i = 18$. When that thread finishes $i = 18$, it will pick the remaining task $i = 36$.

- in the second form, thread 1 will likely handle only $i = 36$: tasks $i = 36, 35, \ldots 19$ go to the available 18 threads, and $i = 36$ is likely to finish last, when $i = 18, \ldots, 2$ are already assigned to the other 17 threads. Since the small values of i will finish almost instantly, $i = 1$ will have been allocated before the initial thread handling $i = 36$ becomes ready again.

Load distribution is clearly more favourable in the second form.

Chapter 3:
PARI functions

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.

`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 ressources allocated outside the PARI stack are freed by `mt_queue_end`.

`void mt_queue_start_lim(struct pari_mt *pt, GEN worker, long lim)` as `mt_queue_start`, 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.

`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 `taskid`s. 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_get`.

`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`.

`void mt_queue_end(struct pari_mt *pt)` end the parallel execution and free ressources 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 ressources; but not before all tasks have been processed else crashes will occur.

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.

```
#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, F1,F2,D, in,out, done;
  struct pari_mt pt;
  entree ep = {"_worker",0,(void*)Cworker,20,"GL",""};
  /* initialize PARI, postponing parallelism initialization */
  pari_init_opts(8000000,500000, INIT_JMPm|INIT_SIGm|INIT_DFTm|INIT_noIMTm);
  pari_add_function(&ep); /* add Cworker function to gp */
  pari_mt_init(); /* ... THEN initialize parallelism */
  /* Create inputs and room for output in main PARI stack */
  N1 = addis(int2n(256), 1); /* 2^256 + 1 */
  N2 = subis(int2n(193), 1); /* 2^193 - 1 */
  M = mathilbert(80);
  in  = mkvec3(mkvec2(N1,gen_1), mkvec2(N2,gen_1), mkvec2(M,gen_0));
  out = cgetg(4,t_VEC);
  /* Initialize parallel evaluation of Cworker */
  mt_queue_start(&pt, strtofunction("_worker"));
  for (i = 1; i <= 3 || pending; i++)
  { /* submit job (in) and get result (out) */
    mt_queue_submit(&pt, i, i<=3? gel(in,i): NULL);
    done = mt_queue_get(&pt, &taskid, &pending);
    if (done) gel(out,taskid) = done;
  }
  mt_queue_end(&pt); /* end parallelism */
  output(out); pari_close(); return 0;
}
```

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 ressources 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 `GEN`s, 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).