Introduction

to

parallel GP

(version 2.19.1)

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:
https://pari.math.u-bordeaux.fr/

Copyright © 2000–2025 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 © 2000–2025 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 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 Timing parallel GP scripts . . . 6
1.2.3 GP functions . . . 7
1.2.4 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
Index . . . 13

Chapter 1:
Parallel GP interface

1.1 Configuration.

This booklet documents the parallel GP interface. The first chapter describes configuration and basic concepts, the second one explains how to write GP codes suitable for parallel execution. Two multithread interfaces are supported:

- 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 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 Timing parallel GP scripts.

There are different notions of timers for parallel programs. The PARI timers behavior depends on the mutithread interface:

• POSIX threads: `getabstime` returns the total CPU time used by all threads, and `getwalltime` returns the real time elapsed. The GP timer will reports both.

Using 8 POSIX threads:

```
? polmodular(101);
? ##
  ***   last result: cpu time 5,785 ms, real time 878 ms.
```

The CPU time is larger than the real time because threads CPU time is counted.

• MPI: `getabstime` returns the CPU time use by the main thread, and `getwalltime` returns the real time elapsed. The GP timer will reports both.

Using 8 MPI threads:

```
? polmodular(101);
? ##
  ***   last result: cpu time 916 ms, real time 939 ms.
```

The CPU time is less than the real time because the main thread waits for the secondary threads to complete.

1.2.3 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`.

1.2.4 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);
? s = 0; for(n = 1, 2*10^6, s += thuemorse(n) / n * 1.)
cpu time = 1,064 ms, real time = 1,064 ms.
```

It is natural to try

```
? export(thuemorse);
? s = 0; parfor(n = 1, 2*10^6, thuemorse(n) / n * 1., S, s += S)
cpu time = 5,637 ms, real time = 3,216 ms.
```

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

```
? s = 0; parfor(N = 1, 20, sum(n = 1+(N-1)*10^5, N*10^5,
                                thuemorse(n) / n*1.), S, s += S)
? cpu time = 1,997 ms, real time = 186 ms.
```

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.

Of course, in this example, `parsum` is much easier to use since it will group the summations for you behind the scenes:

```
? parsum(n = 1, 2*10^6, thuemorse(n) / n * 1.)
cpu time = 1,905 ms, real time = 177 ms.
```

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.

Index

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

P

`parapply` 7
`pareval` 7
`parfor` 7
`parforprime` 7
`parforvec` 7
`parploth` 7
`parselect` 7
`parsum` 7
`parvector` 7