# PARI/GP, toward high performance computing for number theorists

B. Allombert

IMB
CNRS/Université de Bordeaux

26/06/2023

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 676541

# Introduction

PARI/GP is a computer algebra system oriented toward number theory.

- PARI is a C library, allowing fast computations.
- GP is an easy-to-use interactive shell giving access to the PARI functions.
- GP is the name of gp's scripting language.
- GP2C , the GP→ PARI compiler allows to convert GP scripts to C.
- available as a javascript application and a native Android app (PariDroid)
- part of Sagemath and jupyter
- Website: `https://pari.math.u-bordeaux.fr`
- Free software distributed under the GNU GPL 2 or superior

# What PARI/GP can do ?

- Linear algebra (over various fields and rings)
- Polynomial and power series
- Transcendental functions
- Numerical summation and integration.
- $p$-adic functions
- Finite fields
- Number fields
- Lattices and quadratic forms

# Advanced functionalities

- Interpolation methods and guessing methods
- Diophantine equations
- Galois theory
- Class field theory
- Simple central algebras
- Elliptic and hyperelliptic curves
- Modular forms
- $L$-functions
- Simplified parallel programming interface.

# PARI/GP interface to parallelism

. PARI now supports two common multi-threading technologies:

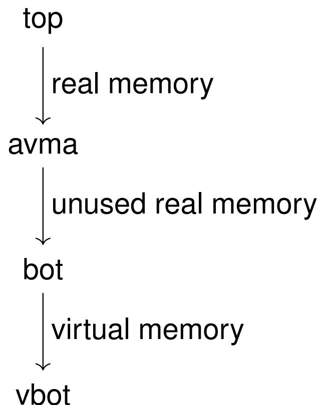
- POSIX thread: run on a single machine, lightweight, flexible, fragile.
- Message passing interface (MPI): run on as many machine as you want, robust, rigid, heavyweight. Used by most clusters.
- other technologies could be easily implemented.

However the parallel GP interface does not depend on the multithread interface: a properly written GP program will work identically with both.

# Some basic on the PARI C interface

The PARI library has its own memory management system which allows for fast creation and garbage collecting of objects and prevent memory leak.
It uses a continuous chunk of memory called the PARI stack.
This chunk is allocated as part of a larger segment of virtual memory.
If the stack need to be enlarged, part of the virtual memory is mapped to actual memory (using the `mmap` system call).

# The PARI stack

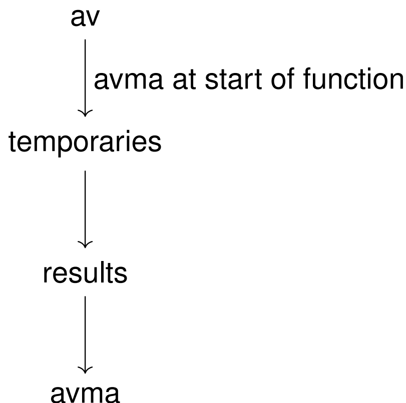


# Objects

New PARI objects are written on the stack starting at the address `avma`. Each object starts by a codeword indicating its type and size. Depending on the type, fields are either data or pointers to other objects.
Garbage collection is done by recording the height of the stack (`avma`) before the computation, and moving up the objects that we want to keep starting this address. Thus, all temporaries are automatically discarded.
This system has the good property that all objects are naturally serialized without incurring higher level language penalty.

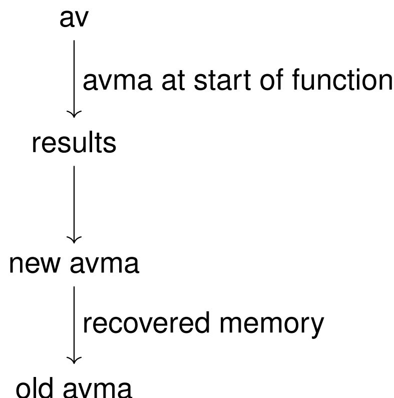
# The PARI stack

Before garbage collection



# The PARI stack

After garbage collection



# Types of objects

List of the PARI types:

| | |
|---|---|
| `t_INT` | long integers |
| `t_REAL` | long real numbers |
| `t_INTMOD` | integer mod n |
| `t_FRAC` | rationals |
| `t_FFELT` | finite field elements |
| `t_COMPLEX` | complex numbers |
| `t_PADIC` | $p$-adic numbers |
| `t_QUAD` | quadratic numbers |
| `t_POLMOD` | poly mod |
| `t_POL` | polynomials |
| `t_SER` | power series |
| `t_RFRAC` | rational functions |

# Types of objects

| | |
|---|---|
| `t_QFB` | binary quadratic form |
| `t_VEC` | row vector |
| `t_COL` | column vector |
| `t_MAT` | matrix |
| `t_LIST` | list |
| `t_STR` | string |
| `t_VECSMALL` | vector of small ints |
| `t_CLOSURE` | functions and closures |
| `t_ERROR` | error context |
| `t_INFINITY` | $\pm\infty$ |

# t_CLOSURE

The type `t_CLOSURE` is generated by the GP bytecode compiler from GP functions and closures.
It can also be used to generate GP wrappers around C functions that handle conversion between GP calling convention and C calling convention.

# Toward parallelism

From the above, we can see how parallelism can be implemented in a distributed or shared memory context:

1. We need separate PARI stack for each execution threads.
2. We use serialization to send data through the network.
3. We send code to execute to threads as `t_CLOSURE` objects.

This is sufficient to implement basic parallelism without shared state.

# C interface to parallelism

- `mt_queue_start` send a `t_CLOSURE` to threads to be evaluate and start parallel execution
- `mt_queue_submit` submit data to be evaluated to threads
- `mt_queue_get` obtain results of computation by threads.
- `mt_queue_end` stop parallel execution.

# PARI functions that support parallelism

- Algorithms based on the Chinese remainder theorem (linear algebra and polynomial over the rationals).
  - linear algebra over number fields
  - resultants over number fields
  - Modular polynomials and class polynomials
- Algorithms based on search of relations in a factor basis (discrete logarithm, class groups)
- Primality testing
- Euler products of $L$-function
- Modular forms
- Table of number fields

# Example of use

PARI/GP use generalized Kronecker interpolation to reduce most elementary algebraic operations to multiplication of large numbers that are then be computed in almost linear time. A number of algebraic operations are implemented using the Chinese remainder theorem strategy which has the advantage to be easy to parallelize while being asymptotically fast, and having good locality.

$$M_n(F_q) \xrightarrow{\text{lift}} M_n(\mathbb{F}_p[X]) \xrightarrow{\text{Kronecker}} M_n(\mathbb{Z}) \xrightarrow{\text{red}} \prod_\ell M_n(\mathbb{Z}/\ell\mathbb{Z})$$

$$\prod_\ell M_n(\mathbb{Z}/\ell\mathbb{Z}) \overset{\text{squaring}}{\dashrightarrow} \prod_\ell M_n(\mathbb{Z}/\ell\mathbb{Z})$$

$$\prod_\ell M_n(\mathbb{Z}/\ell\mathbb{Z}) \xrightarrow{\text{CRT}} M_n(\mathbb{Z}) \xrightarrow{\text{Kronecker}} M_n(\mathbb{F}_p[X]) \xrightarrow{\text{red}} M_n(\mathbb{F}_q)$$

Figure: Square of a matrix over a finite field of large characteristic p

The operations are done in parallel coefficient-wise except the squaring which is done in parallel over the small prime numbers $\ell$.

# Concept

The GP language provides functions that allow parallel execution of GP code, subject to the following limitations: the parallel code

- must be free of side effect,
- cannot access global variables
- instead access variables exported to the parallel world with `export`.

## Parallel algorithms

A number of PARI functions will use parallelism when available:

```
? default(timer,1);
? isprime(2^600+187)
cpu time = 1,244 ms, real time = 197 ms.
%2 = 1
? nbthreads = default(nbthreads);
? default(nbthreads,1)
? isprime(2^600+187)
time = 660 ms.
%5 = 1
? default(nbthreads,nbthreads);
```

Here the parallel version is three times faster. Under pthread, the CPU time is the sum of the time used by all threads. The real time is smaller than the CPU time due to parallelism.

# Simple examples

```
? ismersenne(x)=ispseudoprime(2^x-1);
? apply(ismersenne,primes(400))
cpu time = 1,248 ms, real time = 1,247 ms.
%7 = [1,1,1,1,0,1,1,1,0,0,1,0,0,0,0,0,0,1,0,...
? parapply(ismersenne,primes(400))
cpu time = 2,253 ms, real time = 298 ms.
%8 = [1,1,1,1,0,1,1,1,0,0,1,0,0,0,0,0,0,1,0,...
? select(ismersenne,primes(400))
cpu time = 1,192 ms, real time = 1,199 ms.
%9 = [2,3,5,7,13,17,19,31,61,89,107,127,521,607,127
? parselect(ismersenne,primes(400))
cpu time = 2,248 ms, real time = 299 ms.
%10 = [2,3,5,7,13,17,19,31,61,89,107,127,521,607,12
```

Compare the real time.

# The parallel world

`export` is used to set values in the parallel world.

```
? ismersenne(x)=ispseudoprime(2^x-1);
? fun(V)=parvector(#V,i,ismersenne(V[i]));
? fun(primes(400))
  *** parvector: mt: please use export(ismersenne)
> break
? export(ismersenne)
? fun(primes(400))
```

# exportall

`exportall` exports all current global variables.

```
? V=primes(400);
? parvector(#V,i,ispseudoprime(2^V[i]-1))
  *** parvector: mt: please use export(V).
> break
? exportall()
? parvector(#V,i,ispseudoprime(2^V[i]-1))
```

# Using parfor

`parfor` is the parallel version of `for` which have both a parallel section and a sequential section;

```
parfor(i=a,b,
 /* parallel section, depend on i */
  , R /* set to result of parallel section,
 /* sequential sequention, depend on i and R */)
```

The parallel section is executed in parallel for all $i$, while the sequential section is executed sequentially for all values of $R$ when they become available.

# Using parfor

```
? ismersenne(x)=ispseudoprime(2^x-1);
? export(ismersenne)
? parfor(p=1,999,ismersenne(p),c,if(c,print(p)))
? prodmersenne(N)=
  { my(R=1);
    parforprime(p=1,N,
      ismersenne(p),
      c,
      if(c, R*=p));
    R;
  }
? prodmersenne(1000)
cpu time = 108 ms, real time = 31 ms.
%13 = 637764906056784026430
```

# Using parforprime

```
? parforprime(p=1,999,ismersenne(p),c,\
      if(c,print(p)))
? prodmersenne(N)=
{ my(R=1);
  parforprime(p=1,N,
    ismersenne(p),
    c,
    if(c, R*=p));
  R;
}
? prodmersenne(1000)
%15 = 637764906056784026430
```

# return

```
? ismersenne(x)=ispseudoprime(2^x-1);
? export(ismersenne)
? findmersenne(a)=
  {
    parforprime(p=a,,ismersenne(p),c,
      if(c,return(p)));
  }
? findmersenne(4000)
cpu time = 2,600 ms, real time = 366 ms.
%17 = 4253
? findmersenne(8)
cpu time = 4 ms, real time = 1 ms.
%18 = 13
? findmersenne(8)
%19 = 13
```

# return

```
? parfirst(fun,V)=
  parfor(i=1,#V,fun(V[i]),j,if(j,return([i,j])));
? parfirst(ismersenne,[4001..5000])
cpu time = 3,104 ms, real time = 442 ms.
%21 = [253,1]
```