REPORT ON OpenDreamKit DELIVERABLE D5.16

PARI suite release (LIBPARI, GP and GP2C) that fully support parallelisation allowing individual implementations to scale gracefully between single core / multicore / massively parallel machines.

BILL ALLOMBERT, KARIM BELABAS

Due on	2019-08-31 (M48)
Delivered on	2019-08-29
Lead	CNRS (CNRS)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/114`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #114 ON 2019-08-29

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** CNRS
- **Due:** 2019-08-31 (month 48)
- **Nature:** Demonstrator
- **Task:** T5.1 (#99)
- **Proposal:** p. 51
- **Final report** (sources)

The Pari library is a state-of-the-art library for number theory and an important component of the Sage computational system. Together with the `gp` command line interface and the `gp2c` compiler, it forms the Pari/GP package. This deliverable implements a generic parallel engine in the Pari/GP system, uses it inside the system to implement fast parallel variants of existing sequential code and exports it for library users. The released Pari/GP suite (PARI-2.12) makes those improvements and new features available for the community, in particular Sage users and all softwares using the Pari library.

The MultiThread engine transparently supports: 1) sequential computation, 2) POSIX threads (for a single multicore machine) and 3) Message Passing Interface (MPI, for clusters). It is used throughout the library to improve a large number of high-level mathematical algorithms, including fast linear algebra over the rationals or cyclotomic fields, fast Chinese remainders, resultants, primality proofs, discrete logarithms, modular polymomials and isogeny-based algorithms, motivic L-functions... Those implementations scale transparently between single core, multicore and massively parallel machines.

CONTENTS

1. INTRODUCTION AND RATIONALE

The PARI library is a state-of-the-art library for number theory, developed at Bordeaux University. It is an important component of the SAGE computational system: SAGE uses it for example to implement number fields and elliptic curves. PARI itself is a C library but it comes with a command-line interface called `gp` which can be programmed in the GP scripting language, and a GP-to-C compiler called GP2C. Together, these form the software package PARI/GP. This deliverable is about allowing, simplifying, or spreading the use of parallel computation in the system.

In experimental number theory, many large computations are "embarassingly parallel": no communication is needed between parallel tasks and there is little data to pass from the main thread to the computing nodes. They usually fit one of the following scenarios:

(1) exhaustive enumeration for theorem proving: for instance, asymptotic methods prove that all integers larger than some moderate bound X satisfy the theorem and we must check the remaining finitely many integers $n \leq X$. This generalizes to more complicated parameter sets as long as some theoretical proof leaves us with a finite moderately-sized space to enumerate.
(2) building databases: number theorists love building tables and databases of interesting objects, usually all (finitely many) objects of given "kind" and bounded "size". The motivation is for instance to infer general models from finite statistics or to test algorithms on large unbiased data sets. The L-functions and Modular Forms Database (LMFDB) is a prominent example.
(3) sampling: exploring a huge space until a favourable outcome occurs, e.g., a collision (as in the rho or ECM integer factoring method), or until sufficiently many data has been acquired to solve a problem by linear algebra (as in *sieve* methods to factor integers or *index calculus* algorithms to solve discrete logarithm problems).
(4) modular algorithms: where bounded integers are recognized by their congruence classes modulo sufficiently many primes and the Chinese Remainder Theorem (CRT), or interpolation algorithms where polynomials of bounded degrees are determined by sufficiently many values. The modular computations are independent of each other and offer a general speedup compared to a direct computation because they operate on much smaller inputs and do not suffer from coefficient explosion.

In all these cases, most of the computational effort is easily split into independent parts of roughly equal sizes. In cases 3) and 4), post-processing the independent results may be non-trivial: integer factorization and discrete logarithms end with a linear algebra problem in huge dimensions; sophisticated quasi-linear time CRT or interpolation methods are required for efficient modular reconstruction.

Now, typical number theorists have scant access to HPC clusters and use personal machines. But even cheap laptops have 2 or 4 computing cores nowadays. How can we ensure that PARI/GP users get this expected two- or fourfold improvement ? And make that N-fold if an N-cores cluster becomes available ? Of course, we also need to limit the implementation effort: maintaining three or more variants of each algorithm (single core machine, multicore single machine, cluster) is not manageable.

This deliverable, a merger of subtask D5.10 and demonstrator D5.16 in the submitted proposal, implements a generic parallel engine in the PARI/GP system, uses it inside the system (expecting speed gains) and exports it for library users. The released PARI/GP suite (PARI-2.12 at `http://pari.math.u-bordeaux.fr/download.html`) makes those improvements and new features available for the community, in particular SAGE users and all software using the PARI library.

More precisely, the MultiThread engine, or MT engine for short, transparently supports: 1) sequential computation, 2) POSIX threads (for a single multicore machine) and 3) Message Passing Interface (MPI, for clusters). It can be used in two different ways:

- Explicit parallelism: the MT engine provides interfaces to launch subtasks in parallel and efficiently share global data with the tasks. The user must subdivide the tasks herself, handle load balancing and failures and recombine results, but she has complete control.
- Implicit parallelism: a relevant selection of high-level system routines decide on their own to use explicit parallelism, or not. This is easier on the user but more complicated to handle, because there are many possible strategies and no automated algorithm can be optimal in all cases without some amount of user input: for instance, if routine A calls routine B independently many times, it is easier and more efficient to parallelize A and leave B alone than to parallelize B. But how can the engine know that A is going to invoke B many times ? Or that the calls to B have roughly the same cost (or not, in which case load balancing becomes non-trivial)?

2. WORK ACCOMPLISHED

2.1. **Initial state at the start of the project**

Since PARI-2.7 (released 03/2014), it was already possible to build a thread-safe PARI library and to use the standard POSIX threads interface in separate autonomous C programs. The GP language also contained a handful of parallel control structures, mirroring sequential equivalents, which allowed the explicit scenario in the previous section: for instance, the sequential `for` loop has a parallel `parfor` equivalent.

Two main difficulties with parallel computing in GP prevent an automatic switch to the implicit paradigm: first, using the parallel machinery involves overhead and it is hard to decide automatically whether an arbitrary `for` loop will benefit or suffer; and second, global variables (more generally, side effects) cause the usual difficulties.

2.2. **The generic MT engine**

The MT engine was written and finalized in 2015 and 2016 and is in production since PARI-2.9 (released 11/2016). It supports sequential evaluation (no parallelism), POSIX threads and MPI within the same code base. The full suite is supported: GP2C correctly compiles GP code which is making use of the GP parallel interface.

Since then, the engine has been extensively tested on Unix/Linux desktop computers (up to 96 cores) and clusters (up to 128 nodes), and on laptops running Linux, OSX and Windows 10 (Bash on Windows). As expected, in embarassingly parallel algorithms, the measured speedup was essentially linear in the number of available cores; in more complicated programs where post-processing is not negligible, the gain is smaller.

User documentation for the engine is included in an annex at the end of the present document.

2.3. **Update of the GP interface**

The pitfalls of parallel GP programming have been listed and documented (see Chapter 2 of the user documentation in Annex). A major problem was the use of global GP variables, in particular user-defined global functions in parallel sections of the code. How to do this was obscure and hard to explain to beginners. This led to the introduction of GP functions `export` and `exportall` (in PARI-2.12) which allow to cleanly export specific global variables, including user-defined functions, in parallel threads. To avoid difficulties, the exported value is read-only in the thread as frozen at the time of the `export` statement.

In order to better understand the impact of parallelization on running time, the `gp` timer was changed to print both CPU time and elapsed real time (wall-clock time) when the MT engine is enabled.

A new GP function `parploth` for parallel hi-res plots of slow-to-evaluate functions was also contributed by Loïc Grenié.

2.4. **Use of the MT engine throughout the PARI library**

After a preliminary assessment in early 2017, development work since PARI-2.9 has been targeted at using the MT engine wherever it made sense in the PARI code base. We strived to parallelize functions by order of decreasing impact inside the PARI library and by increasing complexity of existing sequential code. In particular, basic routines with multiple applications were attempted first, before the higher level ones. In the current public release (PARI-2.12, released 06/2019), the following functions are covered by the engine:

- simultaneous Chinese remaindering (CRT) problems for a fixed set of moduli: a fast (quasi-linear time) sequential CRT routine was actually written as the engine was developed (previous CRT implementation was quadratic instead of linear). This is a building block used to recognize matrices or polynomials with huge rational coefficients, in standard modular algorithms, e.g., polynomial gcd, characteristic polynomial, inverses of algebraic number; inverse or determinant of matrix.
- fast linear algebra over the rationals and cyclotomic fields: this uses intensively a new implementation of fast CUP decomposition over finite fields developed by Peter Bruin (2018) and turned into a general linear algebra package by Bill Allombert; this is a critical component of the "Modular Forms" and "Modular symbols" packages.
- polynomial resultant in $\mathbb{Z}[X] \times \mathbb{Z}[X, Y]$ via Chinese remainders and Lagrange interpolation: this operation is a building block for algebraic number theory computations over number fields (compositum, field extension, characteristic polynomial and norm of algebraic numbers).
- classical modular polynomials for about 20 invariants (elliptic j, Weber f, f_1, f_2 functions, small eta quotients, ...): this is a building block for complex multiplication and many algorithms exploiting isogenies between elliptic curves (SEA and Satoh point-counting algorithms for elliptic curves, supersingularity test, ECPP primality test).
- discrete logarithm over finite fields (prime fields and $\mathbb{F}_{p^e}$ for word sized prime p).
- Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) primality proving.
- Fourier coefficients of L-functions (Hasse-Weil function for elliptic curves over number fields, symetric powers of elliptic cuves or curves of genus 2 over $\mathbb{Q}$; Artin L-functions of Galois representations).

For the first two items, new sequential code (fast CRT and multimodular reduction) was written in order to maximize the effect of the MT engine. Both the parallel modular tasks (executed sequentially on each node/thread) and the pre- and post-processing were made as efficient as possible.

In the other cases, compared to the pre-existing sequential code, the parallel version adds about 15 extra lines of C on average, conveniently encapsulated in a separate `worker` object (see Annex's chapter 3). Since the engine takes care of most usual pitfalls of parallel programming, e.g. concurrent access or deadlocks, the modifications are routine *provided* 1) the original sequential code is free of side effects and well understood; 2) the code contains natural splitting or synchronization points.

Condition 1) was not always satisfied and we took the opportunity to simplify and rewrite previous implementations in order to improve the code quality before adding parallel instructions. Condition 2) was usually satisfied; one exception was modular computations in a dynamic setting, where we want to stop as soon as the result can be "recognized" and we do not know in advance how many modular threads will be needed. In that case, the behaviour very much depends on the size of the result, which can be bounded in advance, but possibly pessimistically. We handled that case with a naive doubling strategy: we double the number of modular threads until the result is obtained; this way we spend at most double the time which would be needed, had we known the result size in advance.

2.5. **Impact and Related work**

In this section we limit ourselves to a few representative applications for the functions listed in the previous one.

The fast linear algebra over the rationals is a major component of the "Modular Symbols" package (developed between 2013 and 2018 by Karim Belabas and Bernadette Perrin-Riou, released in two stages: PARI-2.9 (2016) and PARI 2.11 (2018)): modular symbols are a standard way to work with spaces of classical modular forms $M_k(\Gamma_0(N))$ using $\mathbb{Q}$-linear algebra. The linear algebra improvements allowed to write routines such as the `ellweilcurve` GP function, which computes the strong Weil curve in an isogeny class of rational elliptic curves starting from the modular eigensymbol having the same eigenvalues as the newform attached to the isogeny class (by the modularity theorem of Wiles at al.) In the range of the Cremona tables that we wanted to check (conductor $N \leq 500,000$), this requires computing kernels of rational matrices of dimension up to $N/6 \approx 83.333$.

The fast linear algebra over cyclotomic fields is a critical component of the new "Modular Forms" package (developed between 2016 and 2018 by Bill Allombert, Karim Belabas and Henri Cohen, released in PARI-2.11). To handle classical spaces of modular forms $M_k(\Gamma_0(N), \chi)$, this package requires working with large matrices (of dimension $\Omega(kN)$) over the cyclotomic field $\mathbb{Q}(\chi)$. In fact, the extension of fast linear algebra from the rationals to cyclotomic fields was motivated by this single critical application.

The fast computation of modular polynomials allowed Jared Asuncion to implement the ECPP primality proof in full generality (PARI-2.11), without relying on the presence of precomputed tables, which would have limited the size of the primes the implementation could handle. For the same reason, the Satoh point counting algorithm for elliptic curves over large non-prime finite fields could be implemented in full generality by Bill Allombert (PARI-2.9).

The parallel computation of Fourier coefficients is used in the "L-functions" package, which computes values of motivic L-functions $L_f(s) = \sum_{n>0} a(n,f)n^{-s}$ at complex points $s \in \mathbb{C}$, where f is some "motive" of geometric origin, for instance a modular form or a curve defined over some number field. For given f, this requires computing all Fourier coefficients $a(n,f)$, $n \leq B$, for some large bound B depending moderately on s (which allows for precomputations if all required s belong to the same region, e.g., a bounded rectangle in the critical strip). For simple

L-functions, the $a(n, f)$ are easy to compute and not worth the overhead of the MT engine; for instance, if L_f is the Riemann zeta function ζ, then $a(n, f) = 1$ for all n. We have written a generic wrapper valid for all L functions satisfying an Euler product that computes the required $a(n, f)$ from the $a(p^k, f)$, where p is now a prime number and $p^k \leq B$. The computations for different primes p are independent and are now made in parallel for all complicated L-functions. The attached generic speedup applies to all L-function related code.

2.6. **Work in progress**

In order to compute values $L_f(s)$ of L-functions, the PARI algorithm needs their Fourier coefficients as described above. Additionally, it requires to approximate inverse Mellin transforms of Γ-products (so-called Meijer G-functions). This part can also be parallelized and this required a major rewrite of the corresponding sequential code, which used complex data structures and far too much memory. This work was finished in August 2019 and is publicly available in the PARI git repository (`http://pari.math.u-bordeaux.fr/cgi-bin/gitweb.cgi`), but not yet in a released version.

We have rewritten PARI's Multiple Polynomial Quadratic Sieve (MPQS) integer factorization sequential implementation to allow its parallelization: this was a daunting task since the existing implementation was both efficient and complicated, full of side effects (modifying global state, writing to/reading from files); so this particular application was attempted last. The rewrite was completed at the beginning of August 2019 in the public PARI git repository, the parallel variant will be written in Autumn 2019. This is work in progress.

The existing class group and units implementation is an equally difficult case with a code base developed over 25 years by many people. After our 2017 assessment, we decided against modifying it during the course of the OpenDreamKit project: in most applications we looked at, making a single class group computation faster (say, by a factor two) would not make a major difference. Naive external parallelization appeared sufficient, where we handle N similar class group and unit computations in parallel instead of parallelizing each single instance to try and make it N times faster. This application may be considered again in the future but is not a high-priority task.

2.7. **Dissemination**

Activities were organized around parallel computation in PARI/GP at the following OpenDreamKit workshops:

- Atelier PARI/GP 2016 (Grenoble) *Parallel PARI with GP2C*, 1h20 tutorial by Bill Allombert.
- Atelier PARI/GP 2017 (Lyon) and PARI/GP 2018 (Besançon) crashtest of MT engine: all participants compiled their own PARI library and GP/GP2C binaries at the start of the Atelier, without altering their workflow or existing programs except by enabling the MT engine via `Configure --mt=pthread`. No configuration problems arose and this transparently yielded the expected speedups for all multicore laptops during the week.
- Atelier PARI/GP 2019 (Bordeaux) *Parallel GP programming*, 1h20 tutorial by Bill Allombert.

The various workshops were also the most important way we got feedback from users about the proposed interfaces and features. In particular, the GP interface was updated when we became aware that it introduced some difficulties for GP programmers (see Chapter 2 in the user documentation in Annex).

3. EXAMPLES

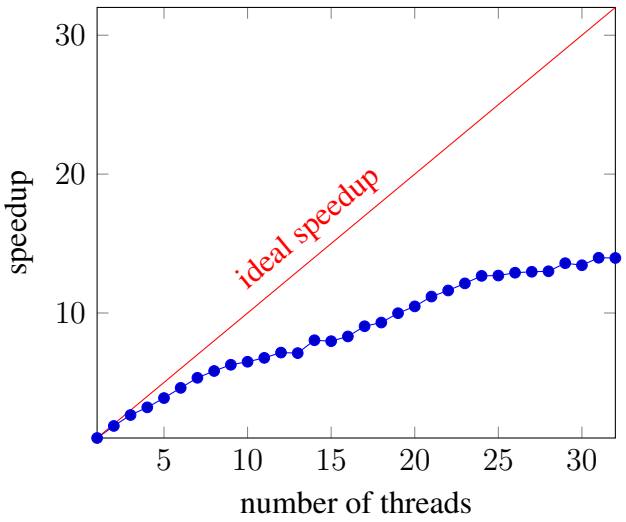
3.1. **POSIX threads**

In this example, we compute the first 10^7 Fourier coefficients of the L-function attached to the elliptic curve $E : y^2 = x^3 + ix + (i + 1)$ defined over $\mathbb{Q}(i)$, where $i = \sqrt{-1}$. We run `gp` configured to use POSIX threads on an idle Linux machine with 8 physical cores (32 logical cores through HyperThreading):

```
? E = ellinit([i,i+1], nfinit(i^2+1));
? default(nbthreads,1) \\ cancel MT engine: sequential version
? ellan(E, 10^7);
  time = 27,801 ms.
? default(nbthreads, 8) \\ use 1 thread / physical core
? ellan(E, 10^7);
cpu time = 34,568 ms, real time = 4,556 ms.
? default(nbthreads, 16)
cpu time = 48,284 ms, real time = 3,346 ms.
? default(nbthreads, 32) \\ 1 thread / logical core
cpu time = 51,336 ms, real time = 2,042 ms.
```

There is some overhead in using the MT engine. With 1 thread per physical core (`nbthreads` is 8), note that $34,568/4,556 \approx 7.6$ close to the number of threads, but the perceived speedup is lower: $27,801/4,556 \approx 6.1$. With 16 threads, the factor goes up to $27,801/3,346 \approx 8.3$ and with 32 we get $27,801/2,042 \approx 13.6$.

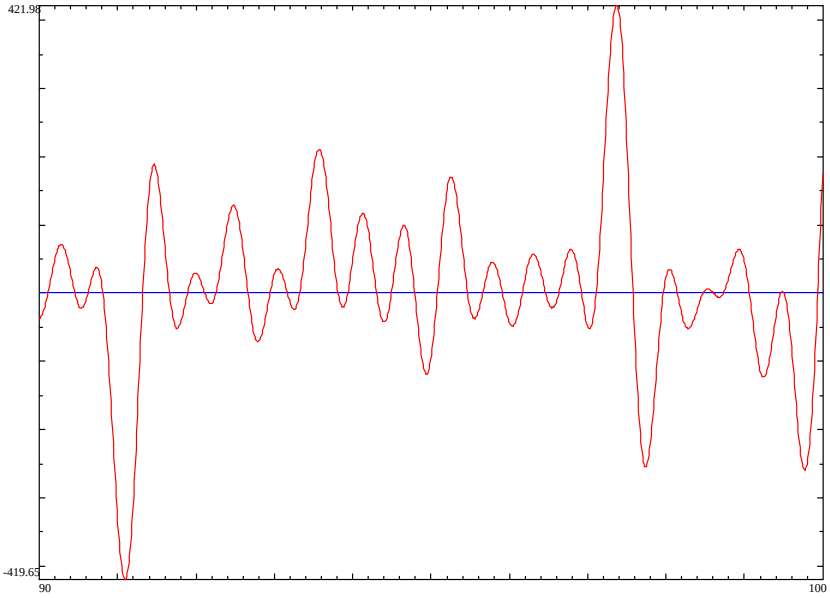
Real time speedup wrt. sequential run (ellan)



With the above coefficients we can for instance compute values of the attached complex L-function, which is expected to satisfy the Riemann Hypothesis: all solutions $L(s) = 0$ are either trivial (here all negative even integers) or on the *critical line* $\Re(s) = 1$. We use 32 threads and start by precomputations allowing fast evaluation on the segment $[1, 1 + 101i]$ of the critical line:

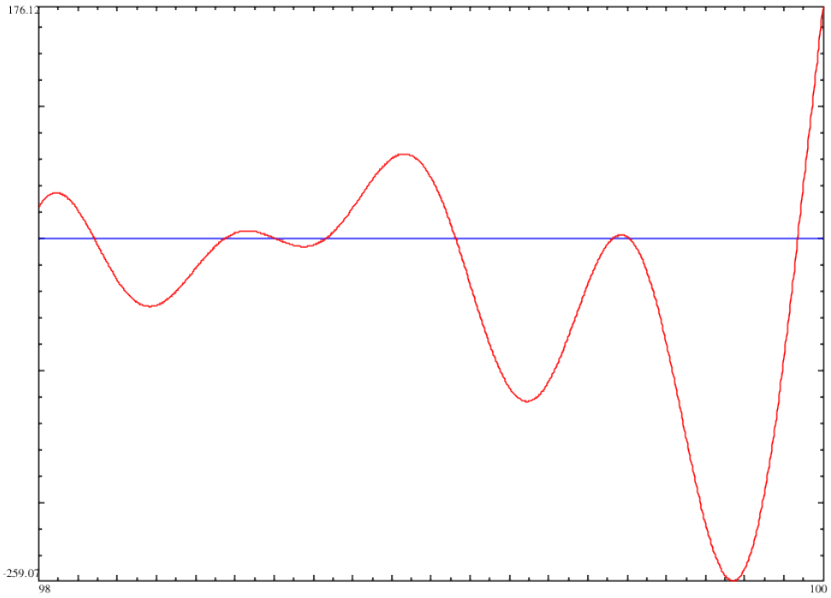
```
? L = lfuninit(E, [101]);
cpu time = 53,048 ms, real time = 2,673 ms.

? lfun(L,1) \\ instantaneous; L does not vanish at 1
%9 = 2.285308639009670220361224919191313551703

? #lfunzeros(L, [0,100]) \\ 308 zeros in the interval [1, 1+100i]
```

```
cpu time = 1,624 ms, real time = 1,630 ms.
%10 = 308

? #lfunzeros(L, [90,100]) \\ ... and 37 in [1+90i, 1+100i]
cpu time = 236 ms, real time = 238 ms.
%11 = 37

? export(L); parploth(t = 90,100, lfunhardy(L,t))
cpu time = 613 ms, real time = 51 ms.
```



The Hardy function plotted above is a real-valued function $H(t)$ of the real variable t such that $L(1+it) = 0 \Leftrightarrow H(t) = 0$. Assuming the Riemann hypothesis, the zeros are simple and H must change sign at each zero of $L(1+it)$, which is in fact the basis of the `lfunzeros` algorithm. We can actually count the 37 zeros in the graph, but two zeros at the right end of the interval are doubtful. Zooming in shows that the graph indeed crosses the real axis twice:

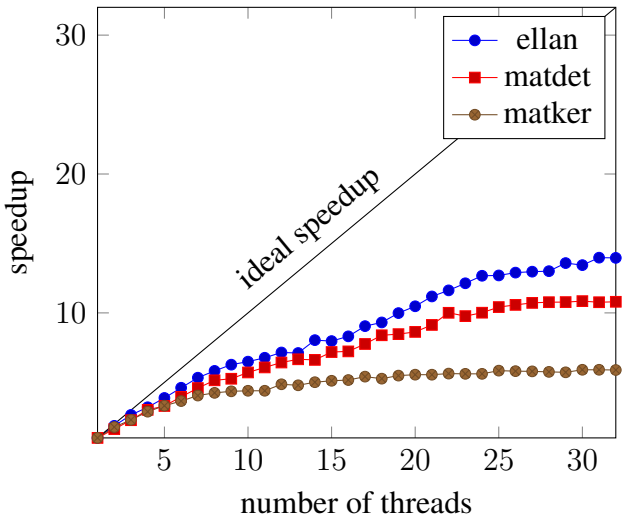
```
? parploth(t = 98,100, lfunhardy(L,t))
```



The speedup profile for each individual command depends on the relative cost of the pre- or post-processing phases. For instance testing determinants and matrix kernels, both computed by Chinese remaindering techniques, the perceived speedups caps around 10.8 and 5.9 respectively, instead of the previous 13.6:

```
n = 300; m = matrix(n,n,i,j,random(2^n)); matdet(m);
n = 150; m = matrix(n,n+40,i,j,random(2^n)); matker(m);
```

Real time speedup wrt. sequential run (det/ker)



The sequential times were respectively 32.3s for the determinant and 19.9s for the kernel. The difference with the previous `ellan` example is easily explained:

(1) We do not know in advance the actual result size so we resort to a doubling strategy, which makes it harder to optimize parameters. In fact, since our inputs are random, the a priori bounds are actually sharp in this example and this strategy is a waste of time.

(2) The overhead is more inportant now since modular linear algebra internally involve three parallelized operations: reducing our input modulo many small primes (p_i), the modular

matrix computations and chinese remaindering to obtain a result modulo the product N of all p_i.

(3) In addition, the kernel involves the rational reconstruction of basis vectors from modular results modulo N and the check that those belong to the kernel, contrary to the determinant which is here known to be an integer and is directly guaranteed to be correct. The final check is currently done in the main thread; in this example, it uses 25% of the real computing time, cancelling some of the parallelization gains. The check could be moved to the parallel section of the code but this causes difficulties in other examples. In any case, there is room for improvement here.

3.2. **MPI**

We now write part of the commands above in a file `E.gp`:

```
E = ellinit([i,i+1], nfinit(i^2+1));
ellan(E, 10^7);
```

We run the same computation non-interactively with a `gp` configured to use MPI; note that the argument to `mpirun -np` is the number of process slots, which is one more than the number of available secondary threads since one slot is used by the master thread:

```
# time mpirun -np 9 gp -q < E.gp
42.97s user 0.45s system 861% cpu 5.040 total
# time mpirun -np 17 gp -q < E.gp
60.28s user 6.86s system 1527% cpu 4.394 total
# time mpirun -np 33 gp -q < E.gp
73.73s user 14.36s system 2776% cpu 3.173 total
```

The overhead is more important than with POSIX threads but we are no longer limited by the number of logical cores on a single machine.

3.3. **External applications**

Since the MT engine is part of the PARI library, it is available without any additional effort to all applications and interfaces using it. This example, taken from deliverable D4.10 (PARI/Python bindings, `cypari2`) shows how easy it is for external applications (here IPython) to leverage the GP parallel interface:

```
In [1]: from cypari2 import Pari; pari = Pari()
In [2]: pari.default("nbthreads", 2)
In [3]: L = [2**i - 1 for i in range(1, 201)]
In [4]: %time res = pari.parapply("factor", L)
CPU times: user 6.76 s, sys: 170 ms, total: 6.93 s
Wall time: 3.6 s
```

This factors a few Mersenne numbers using 2 threads, without paying attention to sizing tasks or load balancing (some factorizations in the list are instantaneous, some much slower). Nevertheless, we gain roughly a factor 2.

APPENDIX A. SNAPSHOT OF THE DOCUMENTATION AT THE TIME OF DELIVERY

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/

Table of Contents

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 $\mathtt{g}(i, c)$. 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 $(t_1, \dots, t_m)$ 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 $1, \dots, m$ 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).

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.