Introduction

to

parallel GP

(version 2.12.0)

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 GP interface

1.1 Configuration.

This draft documents the (experimental) parallel GP interface. Two multithread interfaces are supported:

- POSIX threads
- Message passing interface (MPI)

As a rule, POSIX threads are well-suited for single systems, 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.

1.1.1 POSIX threads.

POSIX threads are selected by passing the flag `--mt=pthread` to `Configure`. The required library and header files are installed by default on most Linux system. Unfortunately this option implies `--enable-tls` which makes the dynamically linked `gp-dyn` binary about 25% slower. Since `gp-sta` is only 5% slower, you will definitely want to use the latter binary.

It is sometimes useful to 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`.

You can test parallel GP support 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 implementation, choose OpenMPI.

To configure for MPI, use

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

To run the program `fun.gp` on 10 nodes, you can then use

```
  mpirun -np 10 gp fun.gp
```

(or `mpiexec` instead of `mpirun` if such is the name used in your MPI implementation).

PARI requires at least 3 MPI nodes to work properly.

Note that `mpirun` is not suited for interactive use because it does not provide tty emulation. Also currently it is not 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 function, sharing the prefix `par`, that allow to execute subtasks in *secondary threads* in parallel (at the same time). The subtasks are subject to the following limitations: the parallel code

- must not access global variables or local variables declared with `local()`,
- must be free of side effect.

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 is:

- 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 span new threads at run time).

PARI requires at least 3 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`.

1.2.2 GP functions.

GP provides the following functions for parallel operations:

- `parvector`: parallel version of `vector`
- `parapply`: parallel version of `apply`
- `parsum`: parallel version of `sum`
- `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`

Please see the documentation of each function for details.

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, the following error 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 the following 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

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

with a different semantic: 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.

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

You may fix this as above 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):

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

or alternatively

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

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

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. It is recomended to use

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

In the two latter forms, v should be considered read-only. It is 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, we recommend to 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` and `forprime`, the 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 (local, as per `my`) variable c, then calls `g`(i, c). Only the function `f` is evaluated in parallel, the function `g` is evaluated sequentially.

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

```
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, we recommend to split the computation in tasks so that each parallel task requires at least a few seconds. Consider the following 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 much 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 addition 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 favorable in the second form.

Index

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

P