

Re: sparse polynomial arithmetic

Source: <http://sci.tech-archive.net/Archive/sci.math.symbolic/2008-04/msg00069.html>

- *From:* bluescarni <bluescarni@xxxxxxxxxx>
 - *Date:* Wed, 9 Apr 2008 13:00:44 -0700 (PDT)
-

Hi Roman,

I'm in a bit of a hurry these days, sorry for replying just now.

On Apr 9, 1:46 am, Roman Pearce <rpear...@xxxxxxxxxx> wrote:

This is much faster than Trip. That is quite impressive. Do you have a 64-bit Linux binary I can download ? I can't install git on a 64-bit machine.

You can get a tarball of the code from the gitweb interface:

http://git.tuxfamily.org/?p=gitroot/piranha/main.git;a=shortlog;h=recursive_series

(just click on the "Snapshot" link from the latest commit). Build instructions here:

<http://piranha.tuxfamily.org/index.php5?title=Installation>

Note that the build instructions deal with the Python bindings, which are not functional for now. You can run the tests mentioned above by enabling the BUILD_TESTS option (and _just_ that option!) in CMake and then running them from within the "tests" directory. Please let me know if you need further directions.

I can also provide binaries of the tests, but they link to quite a new version of libstdc++ (GCC 4.3.0), so I don't know if they can be useful to you.

```
Example 2: Fateman's benchmark, GMP mpz coefficients (16-bit integer
exponents)
f := (1+x+y+z+t)^30;
g := f+1;
multiply p := f*g;
real 6m44.520s
user 6m39.926s
sys 0m4.401s
```

Re: sparse polynomial arithmetic

This takes too long compared to the time for double precision. Something is off. Are you using `mpz_addmul` ? You are merging everything in a Boost hash table ? What is in the table ?

Thanks for pointing that out! I believed the C++ bindings for GMP automatically enabled FMA when composing arithmetic operations, but after a quick search on Google it seems I was wrong. Using `mpz_addmul` I get the following time:

```
real 3m12.544s
user 3m9.239s
sys 0m1.198s
```

More than 50% off :)

That's too bad, but 10 variables is a lot. Try an 8 variable version:
 $9276 \times 13073 = 2285257$ terms

```
f := x1*x2+x1*x8+x2*x3+x3*x4+x4*x5+x5*x6+x6*x7+x7*x8
+x1+x2+x3+x4+x5+x6+x7+x8+1;
g := x1^2+x2^2+x3^2+x4^2+x5^2+x6^2+x7^2+x8^2
+x1+x2+x3+x4+x5+x6+x7+x8+1;
multiply p := f*g and divide q := p/f;
```

My times are 5.5 and 5.9 sec on a 2.4 GHz Core 2 Duo 6600.

Thanks for the suggestion of this benchmark. I'll try it in the next days as soon as possible, and I'll report back here.

Regarding the poor performance of the last two tests, I should probably mention that memory constraints don't permit to use the fastest algorithm

What are the different algorithms ?

Essentially they are applications of the Kronecker trick. I.e., a mapping between the vector space of exponents vectors and \mathbb{Z} . For instance, 3-variables up to degree 3:

x y z code

```
0 0 0 0
0 0 1 1
0 0 2 2
```

Re: sparse polynomial arithmetic

Re: sparse polynomial arithmetic

0 0 3 3
0 1 0 4
0 1 1 5
0 1 2 6
0 1 3 7
.....

Remaining within the boundaries of this representation (which are established with a $O(n)$ analysis of the exponents), you can:

- 1) do arithmetics directly on codes
- 2) use the codes either as
 - a) indices in an array (fast, memory hungry)
 - b) perfect hash values in a hash set (slower, memory saving).

The algorithm first checks if the exponents of the polynomials to be multiplied and of the result of the multiplication can fit in a coded representation where the integer codes can be represented as hardware integers. If this is not the case, a slow term by term multiplication is performed using an hash table for the accumulation of monomials (using a generic hash function). If the coded representation is suitable, then the algorithm tries to allocate the memory needed to represent the polynomials as arrays of coefficients in which the index of each coefficient implicitly gives the code of the monomial. This is the fastest algorithm. If there is not so much memory available and the allocation hence fails, the codes are used as perfect hash values in a hash table instead (which is slower, but can still take advantage of doing arithmetics directly on codes instead of exponent-by-exponent).

With 64-bit integers, it is possible to use the coded representation in a wide class of real-world use cases.

Please note that Piranha is still in heavy development

Of course :) It looks like an interesting system. Thanks for showing it.

Thanks very much for your observations and for your help! I'll reply here when I have additional results.

With kind regards,

Francesco Biscani.

.