Thread boosting and polynomial factorization in NTL

Victor Shoup

October 21, 2015

As of version 9.5.0, NTL provides a *thread boosting* feature. With this feature, certain code within NTL will use available threads to speed up computations on a multicore machine. This feature is enabled by setting `NTL_THREAD_BOOST=on`. during configuration.

This feature is a work in progress. Currently, basic `Z_pX` arithmetic has been thread boosted. More code will be boosted later.

To illustrate the effectiveness of this feature, we report some benchmarks for polynomial factorization in `ZZ_pX`.

All tests were carried out on a very lightly loaded machine with two 64-bit Intel "Ivy Bridge" CPUs, (Intel Xeon CPU E5-2680 v2 at 2.80GHz), each with 10 cores (so a total of 20 cores were available). The system was configured with hyperthreading *disabled*. The system had plenty of memory (over 100GB) The operating system was Cent OS. The compiler was gcc v4.9.2. NTL was built using GMP v6.0.0. All times were measured using "wall clock time" (using the high-resolution `clock_gettime` routine).

This first table gives benchmarks for factoring a degree 2048 polynomial modulo a 2048-bit prime using NTL's `CanZass` routine, which implements the Kaltofen-Shoup baby steps/giant steps variation of the Cantor-Zassenhaus algorithm.

# cores	1	2	4	8	16
time (sec)	436	228	123	70	43
speedup	1.0	1.9	3.5	6.2	10.1
effectiveness (%)	100	96	89	78	63

Note that effectiveness is computed as speedup/(# cores). Space used (measured as `MAX_RSS` via `getrusage`) was 120MB (which increased just slightly as more cores were used).

Note that the only code that was actually "thread boosted" was the basic polynomial arithmetic code in the low-level `ZZ_pX` module (including modular multiplication and modular composition) — none of the code in the `ZZ_pXFactoring` module was touched *at all*. Thus, the speedups obtained reflect only the parallelism of these low-level routines, and not any higher-level parallelism.

In the single-core computation, about 27% of the total time was spent computing $x^p \bmod f$. Most of the remaining time was spent doing modular compositions (using Brent/Kung) to complete the DDF stage of the algorithm. Also, about 75% of the modular composition time time was spent performing matrix multiplications, and the remaining 25% was spent doing polynomial multiplications.

The following table gives the benchmarks just for the $x^p \bmod f$ computation, which illustrates the effectiveness of thread boosting just the `SqrMod` and `MulByXMod` routines used within the `PowerMod` routine (which uses a simple repeated-squaring algorithm).

# cores	1	2	4	8	16
time (sec)	117	62	33	19	11
speedup	1.0	1.9	3.5	6.3	10.7
effectiveness (%)	100	94	88	78	67

This next table gives benchmarks for factoring a degree 4096 polynomial modulo a 4096-bit prime.

# cores	1	2	4	8	16
time (sec)	6416	3221	1669	922	519
speedup	1.0	2.0	3.8	7.0	12.4
effectiveness (%)	100	100	96	87	77

Space used was 586MB (again, increasing just slightly as more cores were used).

Again, we break out the benchmarks for the computation of $x^p \bmod f$, which in the single-core case took about 22% of the total time.

# cores	1	2	4	8	16
time (sec)	1403	752	384	208	115
speedup	1.0	1.9	3.7	6.7	12.2
effectiveness (%)	100	93	91	84	76

Just for fun, we factored a degree 8×1024 polynomial modulo an (8×1024)-bit prime. Using 16 threads, the time was 7239 seconds (just over two hours), and the space was 2.9GB.

Also just fun, we factored a degree 16×1024 polynomial modulo a (16×1024)-bit prime. Using 16 threads, the time was 107700 seconds (just under 30 hours), and the space was 15.4GB.

Implementation notes. Multiplication in `ZZ_pX` is done using a "multimodular FFT". This is easy to parallelize, and seems to be quite effective as long as both the modulus p and the polynomial degree are sufficiently large. The strategy is to chop up the polynomial coefficientwise to do the multi-remaindering/CRT steps, and let different cores work on different chunks of coefficients. The remaining work is basically FFTs modulo a number of single-precision primes (that number grows linearly in $\log_2 p$). Here, we let different cores work on different chunks of primes.