Thread boosting and polynomial factorization in NTL

Victor Shoup

November 23, 2016

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, as of version 10.3.0, basic `ZZ_pX` arithmetic has been thread boosted, as well as matrix arithmetic over `zz_p`. Also, as of version 10.3.0, matrix multiplication over `ZZ_p` has been re-implemented to use a multi-modular strategy, which can be much faster than the old (naive) strategy (it can be 10×–40× faster, depending on parameters). This implementation (as well as the naive implementation, which is still used in certain parameter ranges) is also thread boosted. The modular composition routines for `ZZ_pX` have also been re-implemented to make use of this faster matrix arithmetic (this new modular composition code can be 3×–5× faster than the old code, depending on parameters).

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 a 64-bit Intel "Haswell" CPU (Intel Xeon CPU E5-2698 v3 at 2.30GHz) with plenty of memory (over 250GB) with 32 cores. The system was configured with hyperthreading *disabled*. The operating system was Cent OS. The compiler was gcc v4.8.5. The NTL version was 10.3.0. NTL was built using GMP v6.1.0. 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)	196	106	58	36	28
speedup	1.0	1.8	3.4	5.4	7.0
effectiveness (%)	100	92	84	68	44

Note that effectiveness is computed as speedup/(# cores). Space used (measured as `MAX_RSS` via `getrusage`) was 241MB (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 44% of the total time was spent computing $x^p \bmod f$. Most of rest of the rest of the time was spent doing Most of the remaining time was spent doing modular compositions (using Brent/Kung) to complete the DDF stage of the algorithm.

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)	87	46	24	15	10
speedup	1.0	1.9	3.6	5.8	8.7
effectiveness (%)	100	95	91	73	54

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

# cores	1	2	4	8	16
time (sec)	2279	1171	631	369	247
speedup	1.0	1.9	3.6	6.2	9.2
effectiveness (%)	100	97	90	77	58

Space used was 1.29GB (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 45% of the total time.

# cores	1	2	4	8	16
time (sec)	1029	523	275	152	92
speedup	1.0	2.0	3.7	6.8	11.2
effectiveness (%)	100	98	94	85	70

Just for fun, we factored a degree 8×1024 polynomial modulo an (8×1024)-bit prime. Using 16 threads, the time was 2850 seconds (about 48 minutes), and the space was 7.04GB.

Also just fun, we factored a degree 16×1024 polynomial modulo a (16×1024)-bit prime. Using 16 threads, the time was 34369 seconds (just under 10 hours), and the space was 38.9GB.

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.

The modular composition also makes use of matrix multiplication over `ZZ_p`, which is also implemented using a multi-modular strategy that is similarly parallelized. Also, the small prime matrix multiplications over `zz_p` are implemented using highly-optimized code that is engineered to be cache friendly and to exploit the AVX instruction set.

The effect of the multi-modular matrix implementation is quite pronounced. In all of the benchmarks above, the new factoring code is 2×–3× faster than the old code (that is the speedup for the overall factoring routine — the speedup for modular composition alone is even more pronounced). This comes at a trade-off, though: the overall space usage of the new code is about double the space usage of the old code. However, this space is used in a fairly cache friendly manner.