Performance measurements of a small-space Chromatic Polynomial algorithm

Mats Rydberg

November 1, 2013

1 Chromatic Polynomial in small space

The algorithm measured in this performance test is described in Björklund et al [1]. It is based on the linear-space Fast Zeta Transform described in the same paper, and is proven to perform in time $O(2^n)$ and space $O(1.2916^n)$.

Our input is a graph G on n vertices and m edges. The main subroutine counts the number of ways to colour G using q colours. This is done for $q = 0, 1, \ldots n$, yielding $n + 1$ points (x_i, y_i). These are by definition points which the *chromatic polynomial* $\chi_G(t)$ passes through. $\chi_G(t)$ has exactly degree n, and so we have enough information to recover it explicitly using interpolation.

The algorithm in pseudo-code as follows:

Step I. For $q = 0, 1, \ldots, n$, do

1. Partition V into V_1 and V_2.
2. For each $X_1 \subseteq V_1$, do
   a) For each independent $Y_1 \subseteq X_1$, do
   $$h[V_2 \setminus N(Y_1)] \leftarrow h[V_2 \setminus N(Y_1)] + z^{|Y_1|}$$
   b) For each independent $Y_2 \subseteq V_2$, do
   $$l[Y_2] \leftarrow z^{|Y_2|}$$
   c) $h \leftarrow (h\zeta') \cdot l$
   d) $h \leftarrow h\zeta$
   e) For each $X_2 \subseteq V_2$, do
   $$r \leftarrow r + (-1)^{n-|X_1|-|X_2|} \cdot h[X_2]^q$$
3. Return coefficient c_n of z^n in r.

Step II. Construct interpolating polynomial $\chi_G(t)$ on points (q, c_{nq}).

Step III. Return $\chi_G(t)$.

$N(Y)$ is the set of all vertices adjacent to at least one vertex in Y, $x\zeta'$ denotes the fast up-zeta transform and $x\zeta$ the down-zeta transform (of x). h and l are arrays of size $|V_2|$ of polynomials (initialized to containing zeroes), r is a polynomial. For a more detailed description, see [1].

1.1 Optimizations

Graphs for which certain properties can be determined to exist, graph colouring schemes can be greatly improved in terms of performance.

Exploiting q First, we can consider optimizing on the basis of the value of q.

- For $q = 0$, any graph with $n > 0$ can be coloured in 0 ways. For $n = 0$, 0-colouring is undefined. This is a purely semantic question anyway, and empty graphs are not a relevant topic for this paper. This takes $O(1)$ time.
- For $q = 1$, there are 0 colourings if and only if $|E| > 0$, otherwise there is exactly 1 colouring. This takes $O(n^2)$ time to check, but in practice it is even faster, as we will encounter an edge with high probability before we've scanned the whole graph.
- For $q = 2$, it is well-known that the graph can be coloured (or found to be non-colourable) in polynomial time using standard techniques (such as breadth-first search). This has not been implemented in my programs.

These optimizations will reduce the iterations of the loop at step I by three.

Exploiting $w(G)$ A more sophisticated type of optimization involves exploiting the clique number $\omega(G)$, which is a lower bound of the chromatic number $\chi(G)$[**?**]. Knowing that $\omega(G) \geq a$ for some constant a would allow us to immediately skip all steps I where $q < a$. In the extreme case of the complete graph K_n (with $\omega(G) = n$) we would skip n evaluations and only run the algorithm for $q = n$, because K_n has no colorings for $q < n$.

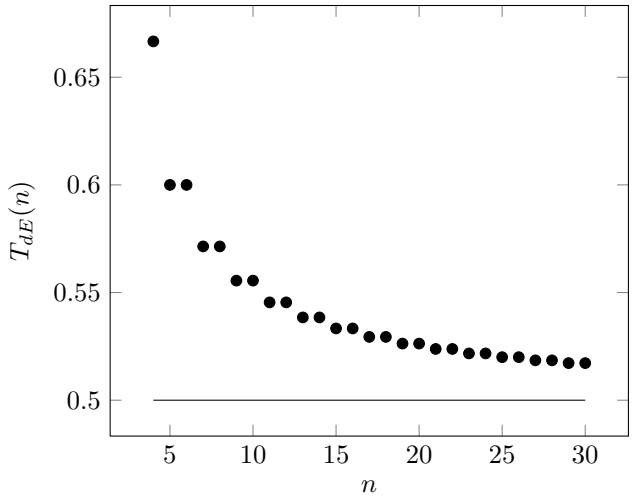
Judging from the density of the graph (which is part of the problem input, since $dE = m/m_{max} = \frac{2m}{n(n-1)}$), we can immediately know the *smallest possible* $\omega(G)$. Let us call it $\omega_{min}(G)$. In fact, the following holds:

$$\omega_{min}(G) = \begin{cases} n - a & \text{if } m = m_{max} - a & 0 \leq a < \lfloor \frac{n}{2} \rfloor \\ \lceil \frac{n}{2} \rceil - a & \text{if } \lfloor (\frac{n}{2})^2 \rfloor < m \leq m_{max} - (a+1)\lfloor \frac{n}{2} \rfloor & 0 \leq a \leq \lfloor \frac{n}{2} \rfloor - 2 \\ 2 & \text{if } 0 < m \leq \lfloor (\frac{n}{2})^2 \rfloor & \\ 1 & \text{if } m = 0 & \end{cases} \tag{1}$$

As we can see from the equation, only graphs with $m > \lfloor (\frac{n}{2})^2 \rfloor$ provides $\omega_{min}(G) > 2$ and for $q \leq 2$ we already have good optimizations. So how dense is a graph where this bound on m holds? Let us specify the threshold density $T_{dE}(n)$ as

$$T_{dE}(n) = \frac{\lfloor (\frac{n}{2})^2 \rfloor}{m_{max}} = 2\frac{\lfloor (\frac{n}{2})^2 \rfloor}{n(n-1)} = \begin{cases} \frac{n}{2(n-1)} & \text{if } n \text{ even} \\ \frac{n+1}{2n} = T_{dE}(n+1) & \text{if } n \text{ odd} \end{cases}$$

In conclusion, any graph with $dE > T_{dE}(n)$ can optimize away at least one computation of step I above. It also follows that as $n \to \infty$ we will have $T_{dE}(n) \to \frac{1}{2}$. The following plot shows how fast we converge for the graphs discussed in this paper.



This result is quite interesting, because for larger graphs, we have a smaller $T_{dE}(n)$, which gives us a higher probability to be able to optimize, and it is also for larger graphs that we are most interested in optimizing techniques. For a graph with $n = 23$ and $dE = 75$, we would be able to skip evaluating $q \leq 7$, which would yield an expected decrease in time by about 15%[1].

Caching As we will see below, in comparison to another algorithm for finding $\chi_G(t)$ (actually for finding the Tutte polynomial, but this encodes χ_G as well), we are using not just less space, but *much* less space. This tempts the thought of being able to do some kind of caching technique, to trade off space for time. No certain structure for a caching scheme has been invented yet, however.

Parallelization Most importantly, and also mentioned in [1], is to parallelize computations of step I2, as these are independent of eachother. This would yield significant time improvements in theory. With access $2^{|V_1|}$ processors, we would be able to execute the program in time $O(1.5486^n)$. Typically, we will only have access to a constant number of processors in practice, allowing each of them to execute a range of iterations of step I2. As presented below, we can expect to reduce the time consumption of the program by a factor of around 4.

We may also parallelize the steps I on $O(n)$ parallel CPUs. This would not reduce the exponential factor of the time complexity, but it does reduce the polynomial factor and it is likely to give significant results in practice. I do not investigate this optimization technique, as my resources are already exhausted by parallelizing step I2.

2 Implementation details

My test implementation only partially supports values of n over 64. In practice, the program does not perform very well for such large problems anyway, so for the mean-

[1] This number is based on experimental results presented below.

time this restriction is not critical. It also allows me to use a quite natural way of encoding sets by simply letting them be a whole integer word, 64 bits long. A one in position i of the word means that vertex i in the graph is present in the set represented by the word.

I've chosen to support adjacency matrices as input structure, representing an arbitrary graph[2]. Another common graph representation is the edge list, which is faster, but since our problem is exponentially hard, another degree of a polynomial term doesn't really affect our performance very much. It is also more straight-forward for me to generate randomized graphs using an adjacency matrix.

For polynomial representation, I've implemented two libraries for number theoretic calculations to be used by my program. These also provide interpolation functionality. For reference, I've also supplied my own implementation of a polynomial, to measure how much the choice of polynomial implementation actually influences the overall performance of the program. Unwilling to implement also my own interpolation functionality, I've decided to use NTL's interpolation implementation. The interpolation step is not critical to performance, so this choice is pretty much arbitrary.

2.1 NTL 6.0.0

The first is NTL, Number Theoretic Library, written in C++ by Victor Shoup at New York University[4]. It is advertised as one of the fastest implementations of polynomial arithmetic, which is all that I am interested in. Unfortunately, it does not provide any non-trivial way of exponentiating polynomials, and its multiplication algorithms are a bit lackluster after some careful studying. It is very easy to use, provides its own garbage collection and has a rich, high-level interface for library usage.

The functions I use are primarily these:

- `ZZX.operator+=()`

  Addition and assignment for polynomials.

- `ZZX.operator*=()`

  Multiplication and assignment for polynomials.

Binaries are called `bhkk-ntl-x.y.z`.

2.2 PARI 2.5.5

Experiencing the relative lack of performance boost from the NTL implementation led me to find also the PARI project. It is written in C mainly by a group of French computer scientists at Université Bordeaux [6], together with a calculator-like interface (`gp`) to be used by an end-user (comparable to Maple). The PARI library is provided at a much lower level, requires the user to garbage collect (since it is written in C, after

[2]Multiple edges and self-edges are two graph invariants that often need special treatment in graph-based algorithms. This is not the case for graph colouring. Any self-edge means the graph is not colourable (a vertex would need to have a different colour from itself), so these are naturally not allowed. In my implementation, I assume they do not exist. Any multiple edge doesn't affect the problem at all, as we are merely considering the *existence* of an edge between two vertices; if there are more than one that doesn't matter.

all), has a much steeper learning curve and a very detailed but hard-to-grasp documentation. It is a bit unclear to me how exactly PARI implements polynomial exponentiation and multiplication, but from the impression I've got and, more importantly, its performance, it has to be non-trivial.

The functions I use are primarily these:

- `ZX_add()`

  Addition for polynomials.

- `ZX_mul()`

  Multiplication for polynomials.

- `gpowgs()`

  General exponentiation for PARI types. Used for polynomials.

Binaries are called `bhkk-pari-x.y.z`.

2.3 GMP 5.1.2

Both the libraries I use to represent polynomials allow (and encourage) the user to configure them using GMP[2] as the low-level interface for integral arithmetic (actually, for all arithmetic, but I only use integers). Authors of both libraries suggest that using GMP instead of their own, native low-level interface will yield significant performance boosts. For this reason, I have done just that. GMP is well documented, easy-to-use, provides both C and C++ interfaces and even has a well-maintained bug reporting facility (I got an answer the same day!). GMP allows the user a rich variety of configuration options, and I've tried to optimize as narrowly as possible to get maximum performance on one machine.

In my implementation, I only use GMP to generate randomized graphs. All arithmetic is performed via the interfaces of PARI and NTL, which themselves call underlying GMP functions. My own naive implementation of polynomials also uses GMP types directly.

3 Algorithm performance parameters

The algorithm has some perks that make it perform better or worse for different input. In this section I aim to explore a few of these characteristics.

3.1 Sparse and dense graphs

The algorithm in itself is designed in a way that allow for a smaller degree of complexity for *dense* graphs. This is in contrast to many of the algorithms which I've studied for the graph colouring problem. And this is not only for very dense graphs, but performance is in fact a function that is directly related to graph density, and consistently performs better for every additional edge to a graph. This follows directly from steps I(2)a and I(2)b above:

$$h[V_2 \setminus N(Y_1)] \leftarrow h[V_2 \setminus N(Y_1)] + z^{|Y_1|}$$

$$l[Y_2] \leftarrow z^{|Y_2|}$$

Recall that these lines will only be executed for *independent* sets Y_1 and Y_2. As graph density increases, fewer subsets of the vertex set V will be independent, and less of these lines will be executed. This has a direct effect in reducing some additions and assignments, but more importantly has side effects in all subsequent steps. Addition-assignment with zero is a non-operation. Multiplication with zero is trivial, and yields even more zeroes. In the extreme case of a complete graph $G = (V, E)$, where all independent subsets of V are of size 1, the most time-critical operation may be the independence testing function (which is executed on all subsets regardless of G).

This discussion of course also applies to a *sparse* graph, in which case we will have *fewer* zeroes and more non-trivial operations.

3.2 Multiplication algorithms

Much of the complexity of the whole algorithm comes down to how polynomial multiplication is performed. The most common operation is to multiply two polynomials of *small* degree but with *large* coefficients. This is because the degree of the polynomials increase as $O(n)$ while their coefficients increase as $O(2^n)$.

Trivially, a polynomial multiplication would be to expand over both operands' coefficients and cross-multiply them in a standard fashion. This is very inefficient, and many techniques have been developed to deal with this problem. In fact, the original issue has always been to multiply two large integers, but the most sophisticated results show methods that make use of polynomials for this purpose. The algorithm with the best asymptotic complexity is the Schönhage-Strassen algorithm [8], but it has a large overhead and becomes useful only for huge operands. It is based on a Fast Fourier Transform. The most go-to algorithm seems to be the Toom-Cook [7] (aka Toom-k) family, in which Toom-2 (aka Karatsuba) or Toom-3 are the most common.

The technique used in Toom-k for multiplying two large integers is to split them up in parts, introduce a polynomial representation of degree $k - 1$ for these parts (the parts are coefficients), evaluate the polynomials at certain points (the choice of points is critical), perform pointwise multiplication of the evaluated polynomials, interpolate the resulting points into a resulting polynomial, and finally reconstruct the integer from the coefficients of the resulting polynomial. This technique is easily translated for polynomial multiplication as well, where the first and last steps would be skipped.

The GMP library supports Karatsuba, Toom-3, Toom-4, Toom-6.5, Toom-8.5 and Schönhage-Strassen [2, p 90], which means all libraries used in my programs uses these algorithms *at least* when multiplying integers (ie, coefficients of polynomials).

NTL implements Karatsuba, Schönhage-Strassen and another FFT-based technique for polynomials [5].

I have not found any documentation specifying which algorithms are implemented in the PARI library for polynomial multiplication. From analyzing the source code, it seems as if PARI "converts" the polynomial to an integer and submits it to its integer multiplication function (which would be one of GMPs).

4 Experimental results

My tests are performed on randomized graphs, generated for some values of n and dE. The algorithm does not assume anything about the general structure of the graphs, and

only exploits the graph invariants mentioned in 1.1.

The following instances have been executed:

- Increasing n in interval $[4, 23]$.

  "Dense" graphs, $dE = 75$.

  "Sparse" graphs, $dE = 40$.

- Increasing density, $n = 19$, $dE \in [5, 100]$.

- "Large" instances, $n = 25$ and $n = 30$, $dE = 75$.

  This is only done for the fastest implementation.

- Evaluation step, $q = 2, 3, \ldots, 19$, $n = 19$, $dE = 40$.

The program has evolved through the development process as new optimizations have been implemented. In order to measure the effects of each optimization simulations have been run on a number of different versions of the programs. In an effort to provide completeness while at the same time not flood this paper with graphs, I've tried to structure up this section by specifying some identifiers for the executables that have been run in simulations.

Due to the linearity of the development process, each executable implements all optimizations included in the ones above them (where applicable).

Identifier	Attributes	Polynomial library	Version number
`aka`	Base	PARI	0.1
`senko`	Base	NTL	0.1
`midori`	Parallelization	PARI	0.2
`kimidori`	Parallelization	NTL	0.2
`buryu`	Min-max clique number	PARI	0.3-beta
`mizudori`	Min-max clique number	NTL	0.3-beta

4.1 Measurements

All tests are performed on the same machine, with the following specifications.

CPU (cores, threads)	Intel i7-3930K 3.2GHz (6, 12)
OS	GNU/Linux 3.8.13.4-desktop-1.mga3 (Mageia 3) x86_64
Memory	16GB DDR3 1333Mhz
Compilator	GCC 4.7.2 (g++)

For all time and memory measurements, the GNU time 1.7 program is used [**?**]. The user time, elapsed time (real time) and peak resident set size are the data points recovered as measurements.

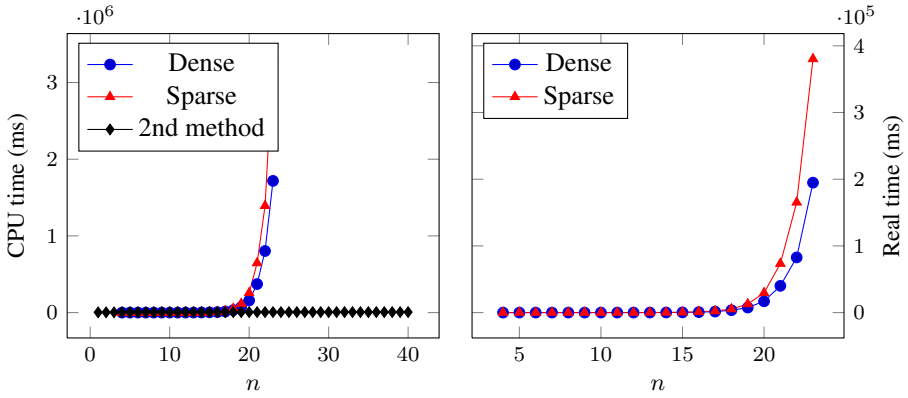
4.1.1 What is space?

4.2 PARI 2.5.5

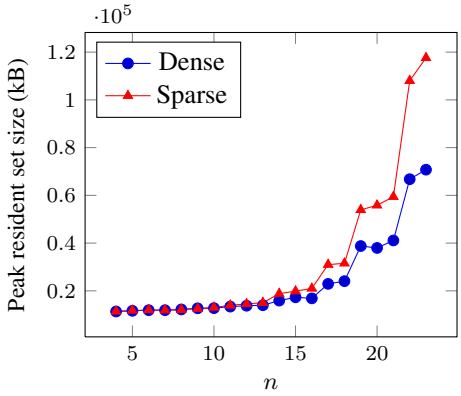
Some text about PARI.

4.2.1 Increasing n

This is time as a function of n. Expected function in theory is $O^*(2^n)$.

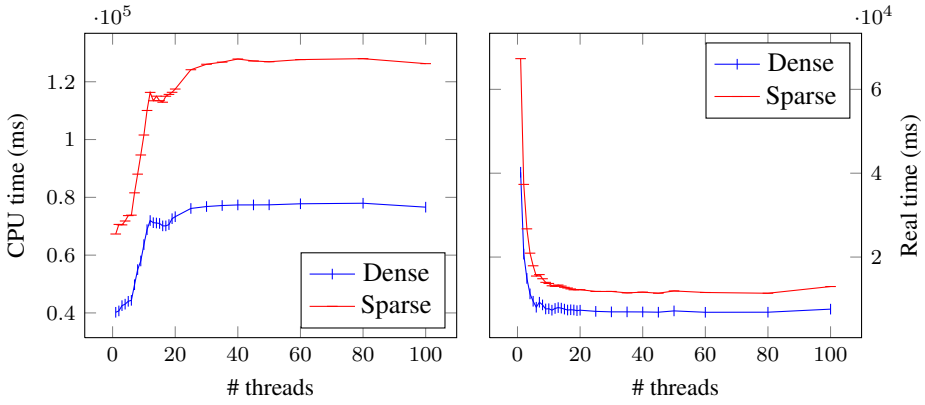


Here we have memory usage as a function of n. Expected function from theory is $O^*(1.2916^n)$.



4.2.2 Parallelization

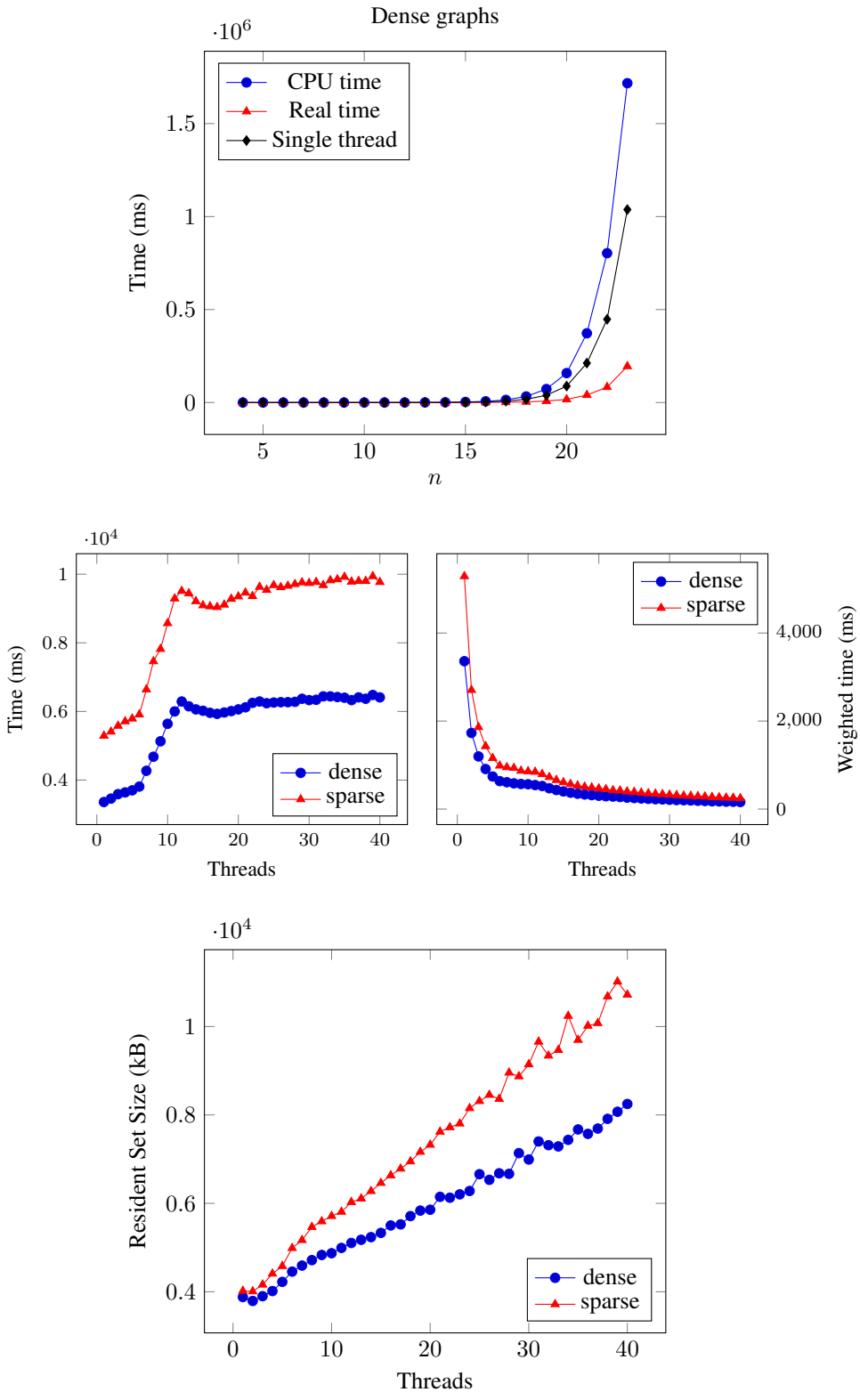
Here we investigate how our parallelization scheme acts as we increase the number of threads. We plot the function of user and real time as we increase the number of threads. Recall that our maximum *actual* parallelization width is 12.

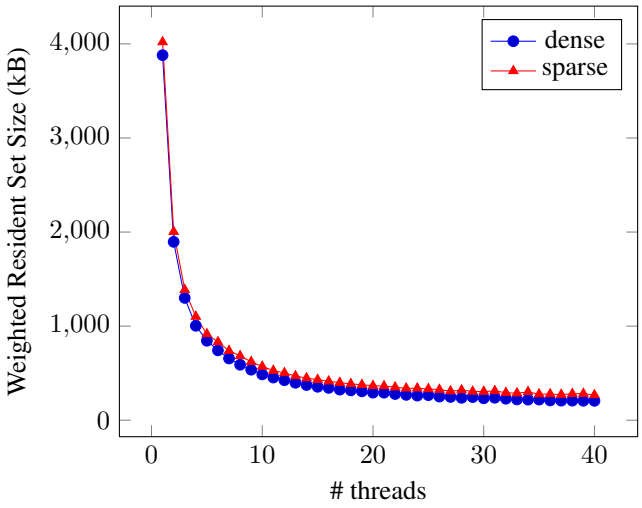


There seem to be no notable difference in how sparse and dense graphs parallelize. As expected, the plots peak at 12 threads. What is not expected is that the top performance was measured for 45 threads. It is not easy to realize that some threads will terminate much sooner than others (because they happened to get a range of subsets which were more frequently independent, see section 3.1), and so using > 12 threads could be useful. It is unclear why 45 is the best (measured) value, however.

Another interesting note to take is that for 100 threads, the gain in processing dense versus sparse graphs has disappeared. This can partly be explained by the fact that the OS is now spending lots of time context switching the 100 threads over the 12 CPUs, but from a more algorithmic point of view, we can make another argument. As we let each thread iterate over fewer and fewer subsets from step I2, our real time consumption is more and more dominated by the one thread that takes the longest time to execute. That is, the thread that was allocated with the ”worst” range of subsets. ”Bad” ranges are more common when the graph is sparse, but as we divide the subsets into smaller ranges, the ”worst” range decreases in ”badness” faster for the sparse graphs than for the dense. It would seem that they have more or less converged for the graph above at 100 threads.

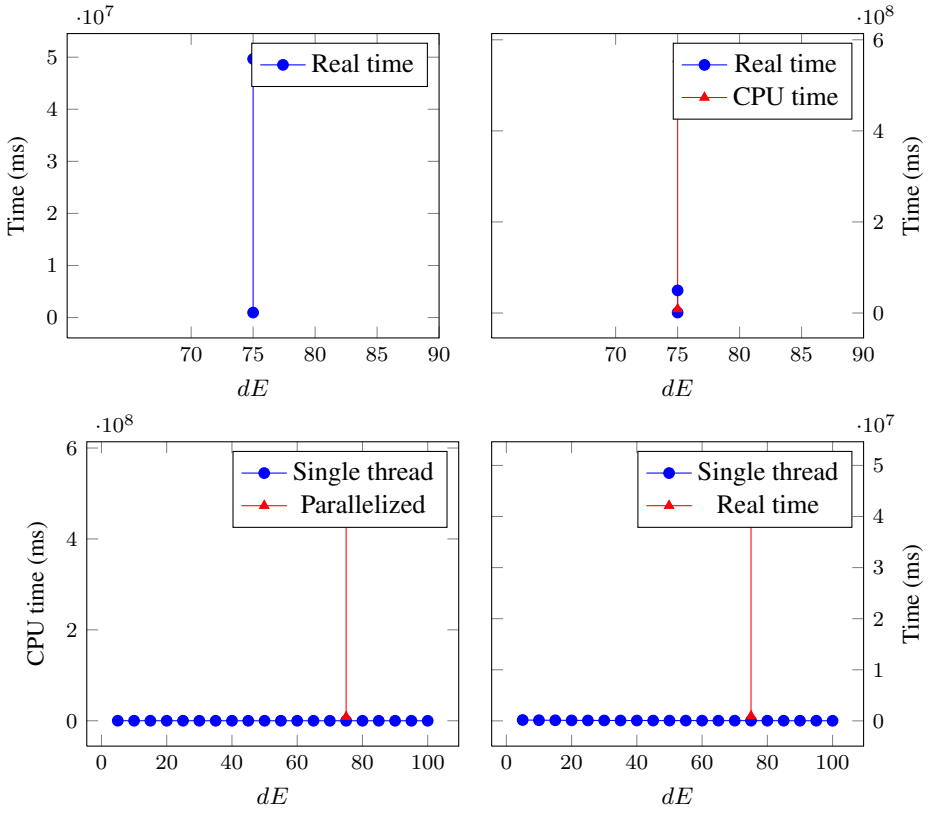
Here we compare CPU time to real time consumption and also to the CPU time used by the single threaded version, and get an indication of how powerful our parallelization is to performance.


Dense graphs
·10^6
CPU time
Real time
Single thread
Time (ms)
n
·10^4
dense
sparse
Time (ms)
Threads
Weighted time (ms)
·10^4
Resident Set Size (kB)
Threads



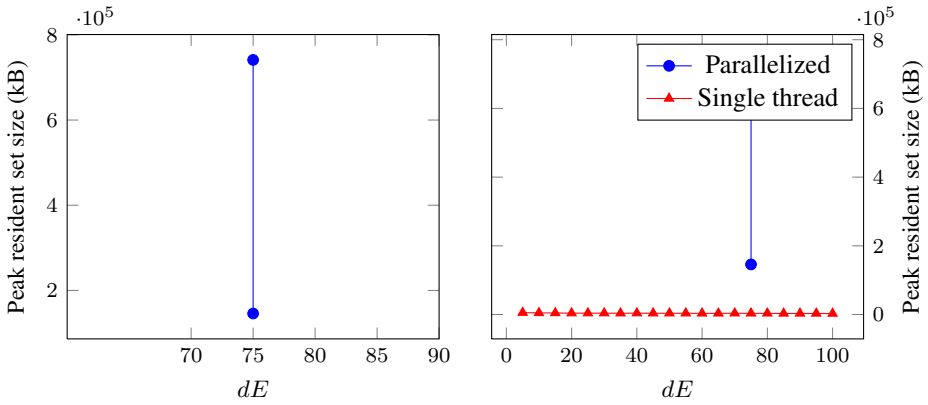
4.2.3 Increasing density

So, we've seen that the density of the graph is important to the performance of the algorithm. But how much falloff can we expect? And are there any certain points of extra interest?

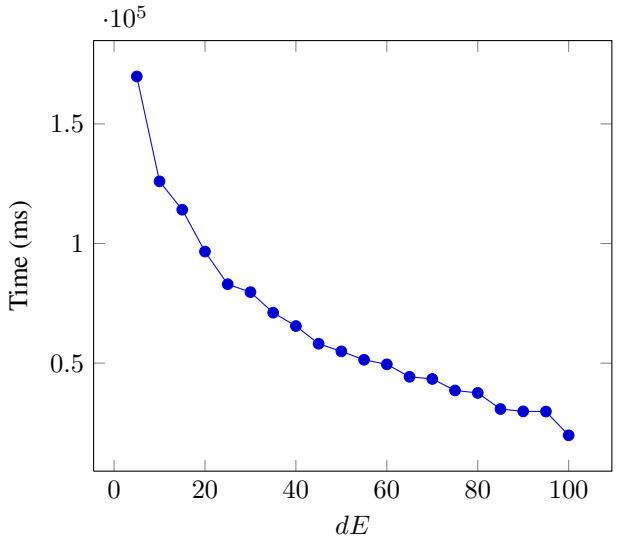


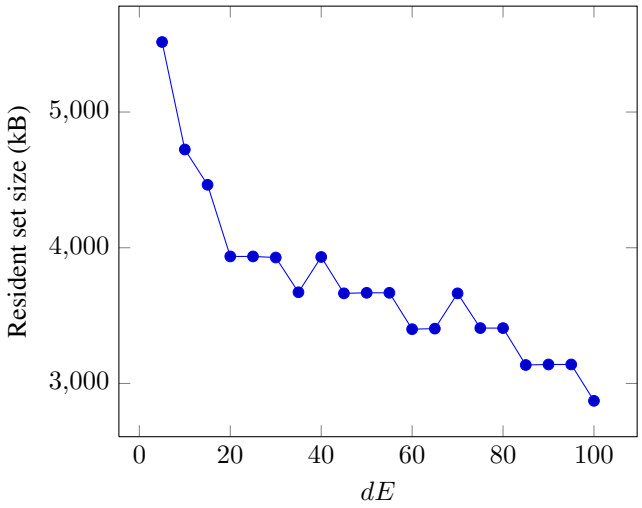
The time falloff follows the same pattern in user time and in real time. The pattern is also the same if we compare to the single thread version.

Memory as a function of density:



Single threaded version:





As we can see, both time and space consumption falls off the most as we add the first edges. At 20% density we use 30% less space and over 40% less time as compared to 5% density. Our best performance is on complete graphs, even though we do not implement any ”special treatment” for graphs of certain density.

4.2.4 ”Large” instance

Single thread: (gets beat) on m_25_40 (I think!)

Program	Time (s)	Peak resident set size (MB)
`chr_pol_pari`	10870	33.93
`tutte`	4906	20129.5

12 threads: (pwns) on m_30_75

Program	CU time (s)	Real time (s)	Peak resident set size (MB)
`chr_pol_pari`	559,841	49,651	2,584
`tutte`	> 94,000	> 95,000	40,135

`tutte` did not finish and was stopped after 95,000 seconds (over 26 hours).

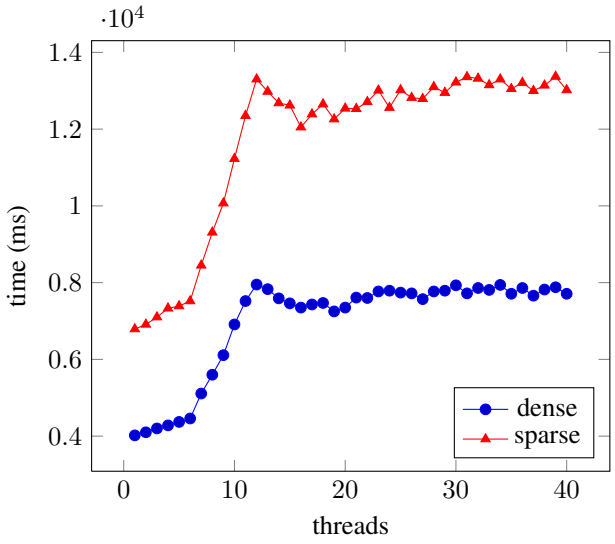
4.2.5 Evaluation step

4.3 NTL

The NTL implementation was the first I incorporated, but its performance did not live up to my expectations. For reference, and to clearly show the importance of the actual implementation of polynomial arithmetics, I’ve let `chr_pol_ntl` execute all the same instances as previously measured with my PARI implementation.

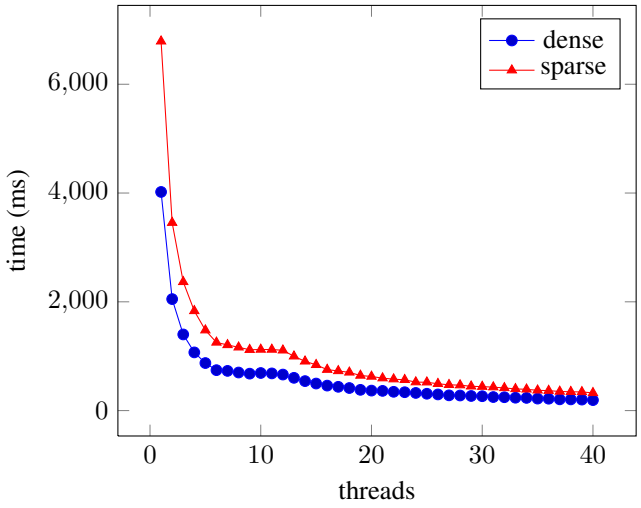
4.3.1 Parallelization

To see how much time we gain on parallelizing, we increase the number of threads used and measure the time consumption.



Time usage goes up drastically and peaks as we reach 12 threads. Note that this is a measure of time spent running in the CPU, the test machine has 12 concurrent CPUs and so we can spend 12 units of time in the CPUs in only 1 unit of real time (theoretically). The conclusion we can draw from this result is that it doesn't seem to be worthwhile to run more than 12 threads, as the effect (whatever it is) of adding more threads is insignificant after the 12th thread.

To get an estimate[3]on the average time spent in each of the CPUs, we can divide the measured time by the number of threads used, as follows:

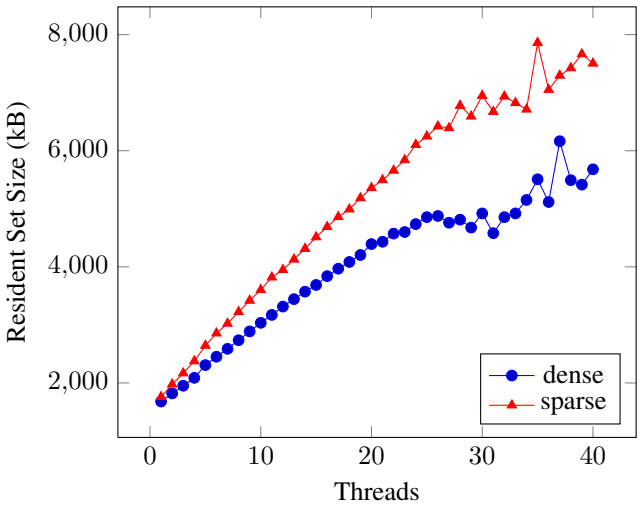


The figure (and the theory [1]) suggests that the average time used by each thread will decrease until we reach our maximum of 2^{n_1} threads. This may not be a decrease

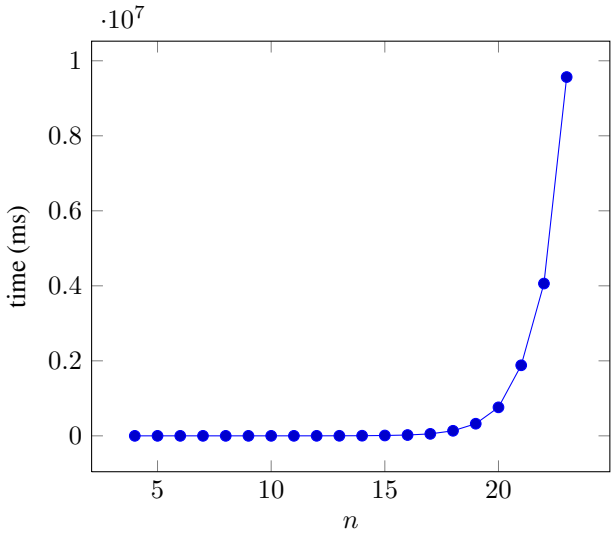
[3]This is an estimate for several reasons. First, we are measuring user time allocated to the process for *all* its computations, not just those done in parallel. Second, we are only running *some* threads in parallel if we have more than 12 threads (max 12). That is, when the fastest thread has completed execution, a 13th thread can occupy its CPU, but when only 11 threads remain, our parallelization width will decrease.

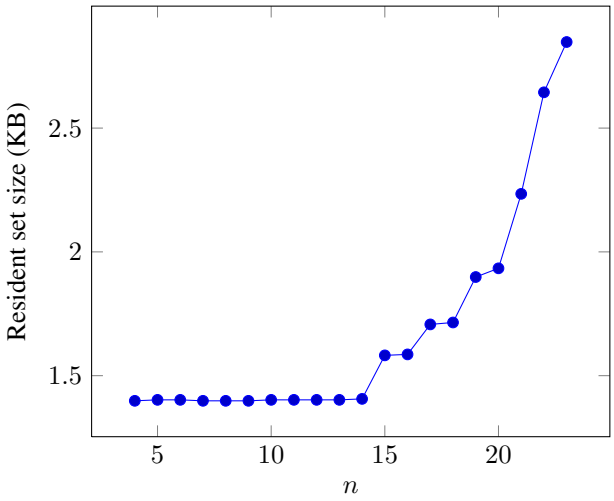
of *actual* time consumption however (unless we have 2^{n_1} CPUs), if we recall the conclusion for the non-weighted graph above. What this graph does tell us is that we dramatically decrease the time used for each thread when adding the first few parallel channels. The effect quickly diminishes though.

The general conclusion is that we should use 12 threads, as was intuitively assumed. This was not an obvious fact, however, as using some waiting threads could speed up overall time consumption, as the fastest threads do not block parallelization after their termination. From the experimental results it is hard to see that kind of an effect have any real impact. Also, the reasoning is the same for both dense and sparse graphs.

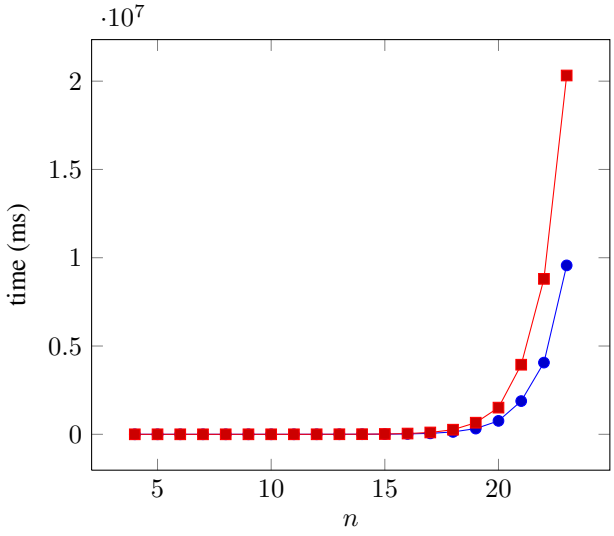


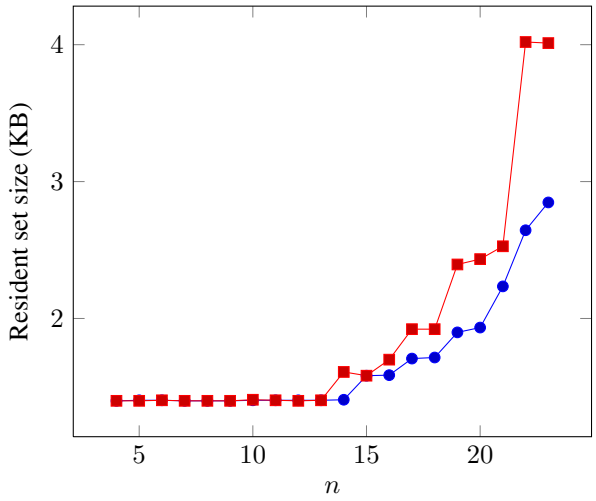
4.3.2 ”Dense” graphs



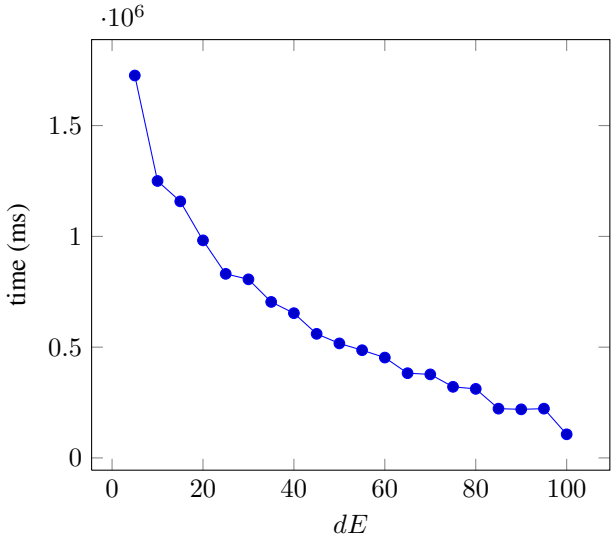


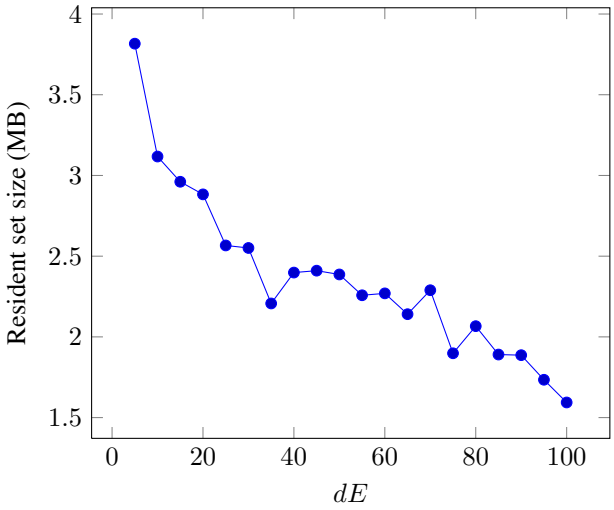
4.3.3 Comparing ”sparse” and ”dense”





4.3.4 Increasing density





4.3.5 Evaluation step

4.4 Naive

Not yet done.

4.4.1 ”Sparse” graphs

4.4.2 ”Dense” graphs

4.4.3 Increasing density

4.4.4 Evaluation step

References

[1] http://fileadmin.cs.lth.se/cs/Personal/Thore_Husfeldt/papers/lsfzt.pdf

[2] http://gmplib.org/

[3] http://man7.org/linux/man-pages/man5/proc.5.html

[4] http://www.shoup.net/ntl/index.html

[5] http://www.shoup.net/ntl/doc/ZZX.txt

[6] http://pari.math.u-bordeaux.fr/

[7] https://en.wikipedia.org/wiki/Toom%E2%80%93Cook_multiplication

[8] https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm

[9] Haggard, Pearce, Royle, 2010: *Computing Tutte Polynomials*