Report on analysis of running time of explicit Euler method for different containers.

Contents

List of Figures

1 Problem

We test the speed of time iteration for explicit Euler method when solving the diffusion equation. Using different types may result in different performance, due to double indexing, size of the problem and cache-friendliness. A detailed analysis of the topic can be seen in below attached report.

Basic code being tested is[1]:

```
for (int tt = 0; tt < t_steps; ++tt) {
    for (int c : interior) {
        double Lap = 0;
        for (int j = 0; j < n; ++j) {
            Lap += SL[c][j] * T1[SD[c][j]];
        }
        T2[c] = T1[c] + dt * Lap;
    }
    T1.swap(T2);
}
```

where `T1`, `T2`, `SL` are (one or two dimensional) arrays of integers, and `SD` is a 2D array of int's.

The outer loop performs time step iterations. The middle loop iterates over all nodes in the domain. For each node value of Laplace operator is calculated by looping over the support nodes (inner loop). Temperature of nodes is then updated, and after the calculations for all nodes are complete, we swap the vectors of previous and newly calculated temperatures.

Other parameters are:

- End time: 0.2
- Time step: 10^{-5}
- Number of time iterations: $tsteps = 20000$
- Basis: 5 Gaussians
- Domain: $[0,1]^2$

We mark node count by N and support size by n. Total number of iterations performed is:

$$I = tsteps \times n \times (N - 4\sqrt{N} + 4)$$

We tested 7 different types of containers, which will appear in the same order throughout the report.

1. our own types: Range, Vec
2. our own operators
3. pure Eigen types
4. C arrays (1 dimensional)
5. std::vector (2 dimensional)
6. std::vector (1 dimensional)
7. our types (1 dimensional)

[1]Full code is avaliable at our repository (link).

2 Environment

For mesuring time standard C++ high resolution clock was used. For measuring number of cache accesses Intel PCM was used.

All tests were run on our clusters computers separately and independently. Hardware and software specifications:

- Intel(R) Xeon(R) CPU E5520 @ 2.27GHz processor
- 256 kB of L2 cache and 8 MB of L3 cache.
- 6 GB of DDR3 RAM
- system: `Linux k5 3.2.0-26-generic #41-Ubuntu x86_64 GNU/Linux`
- `g++` version: `g++ (Ubuntu 4.9.2-0ubuntu1~12.04) 4.9.2`
- `cmake` version: 3.3.2

All tests were ran using the following compile flags:

```
g++ -O3 -DNDEBUG -DEIGEN_NO_DEBUG -std=c++14 -Wall
```

3 Initial measurements

Initially our types and Eigen were much slower that the rest. However, after disabling the debug modes we gained first relevant measurements on figures 1 to 5.

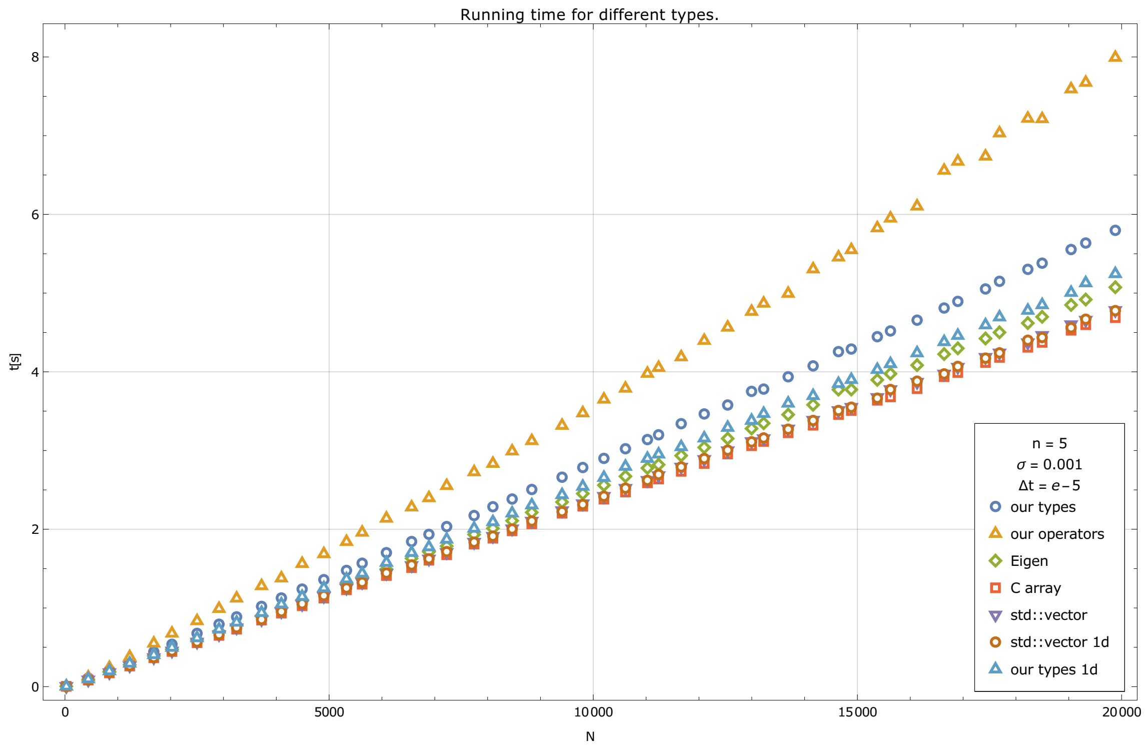

Figure 1: Initial running time for different types.

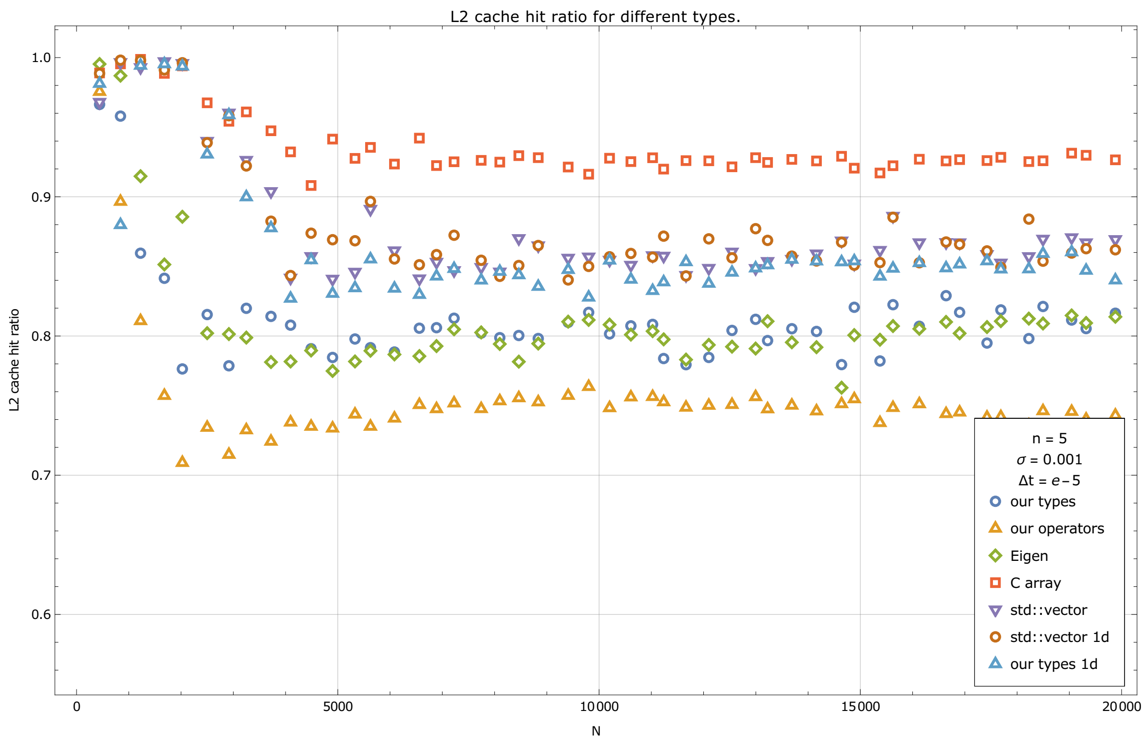

Figure 2: Initial L2 cache hit ratio for different types.

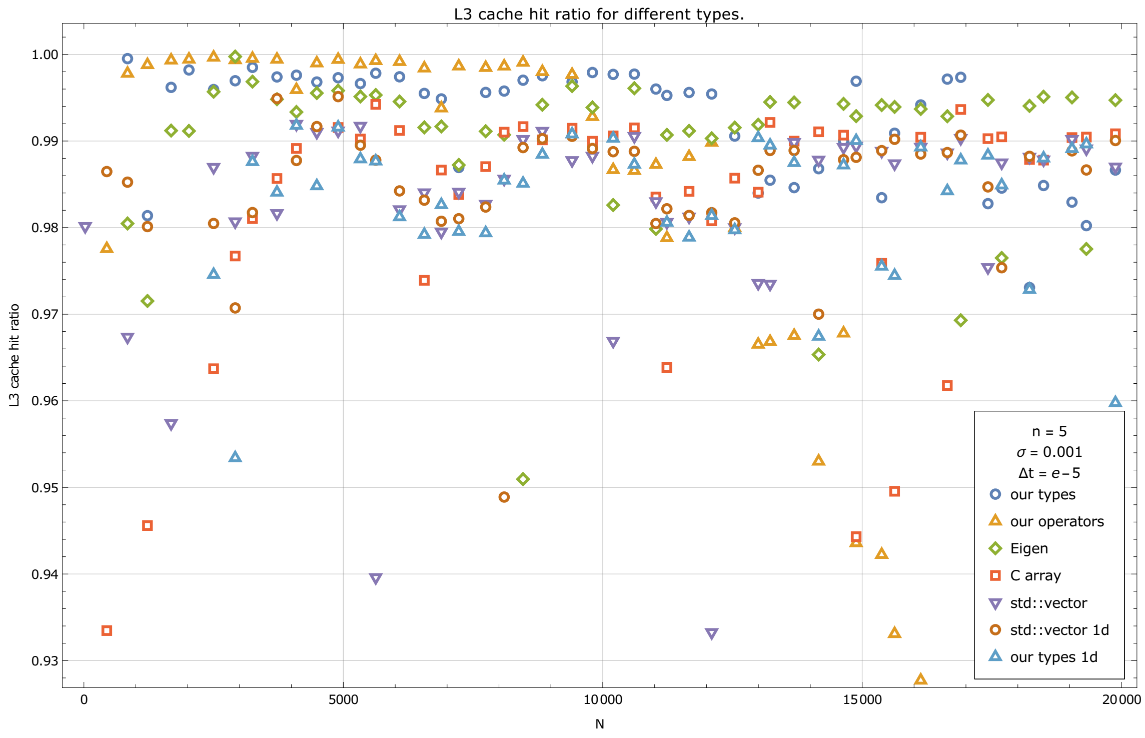

Figure 3: Initial L3 cache hit ratio for different types.

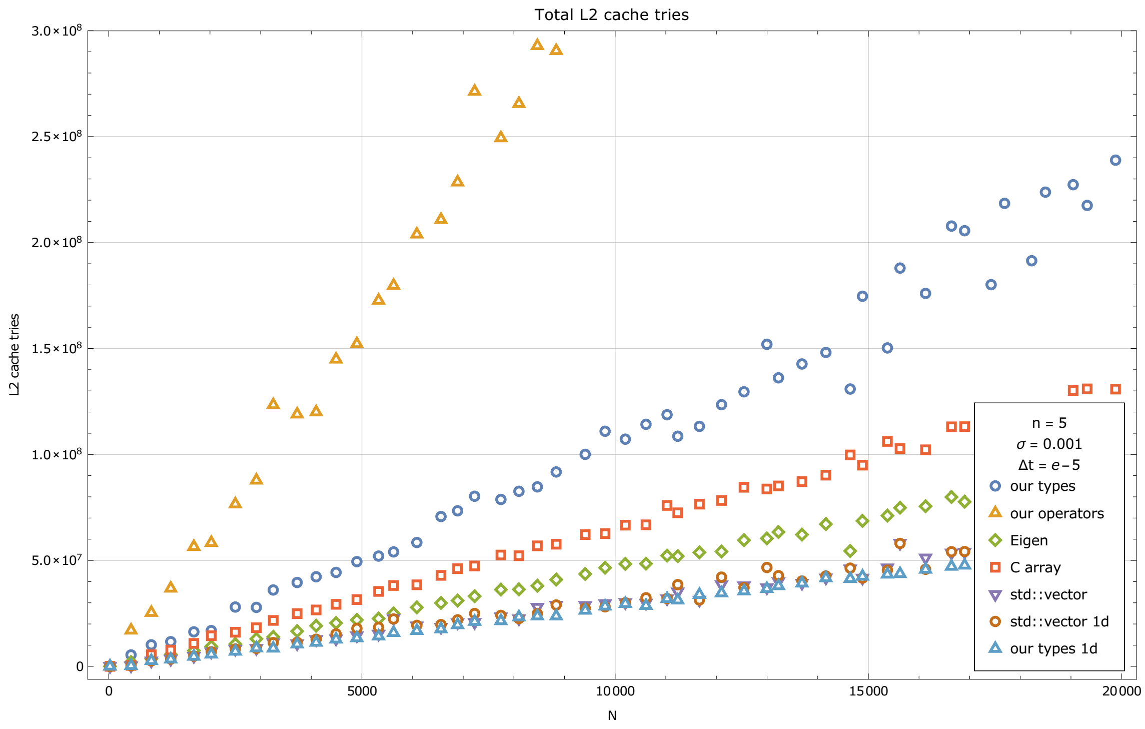

Figure 4: Initial L2 cache total access count for different types.

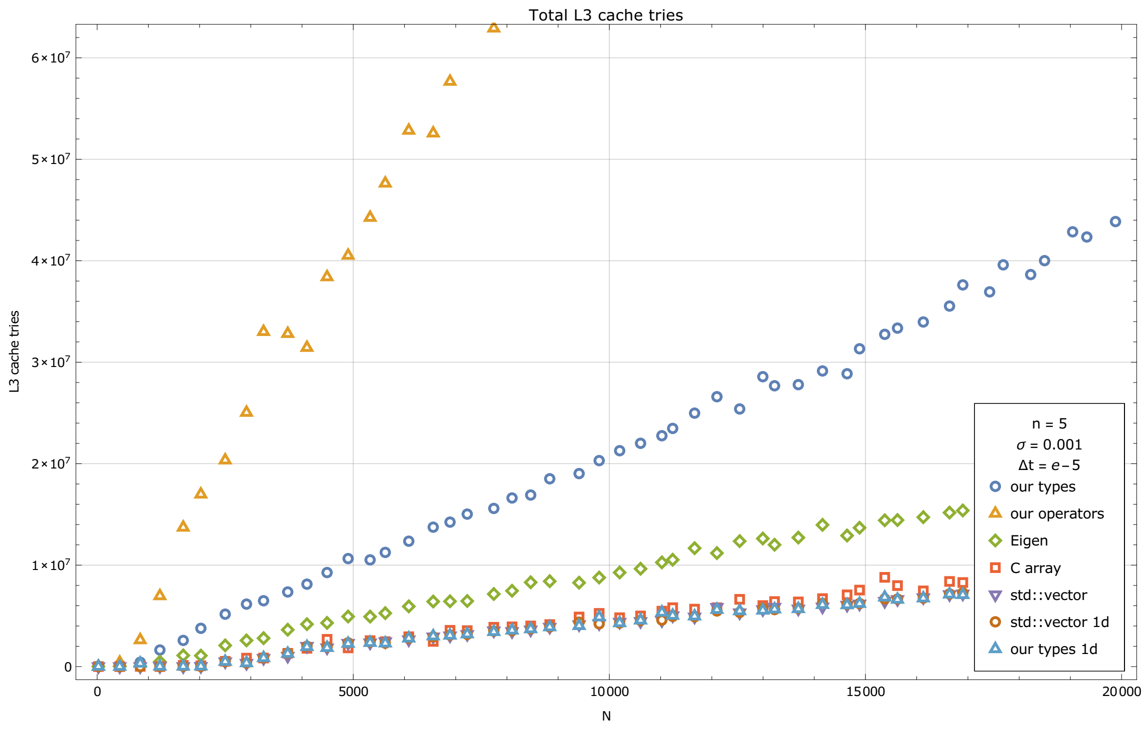

Figure 5: Initial L3 cache total access count for different types.

As the problem size is quite small, L2 cache hit ratio is important. Figure 6 plots correlation between average cache hit ratio and running time growth coefficient, showing strong correlation (except for Eigen).

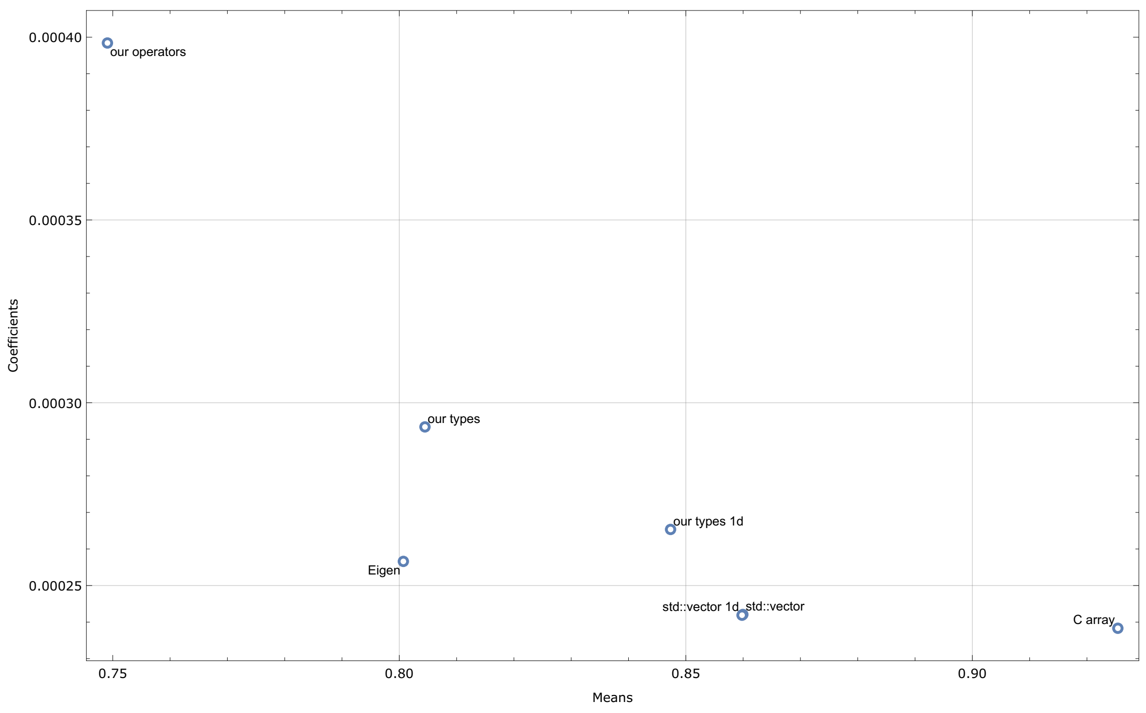

Figure 6: Correlation between running time and L2 cache hit ratio.

Looking at L2 total accesses graph, we see that C array uses twice as many accesses as 1 dimensional types, and our types use 4 times as many. Operators, which call a function in their own class, are by far the worst, which is blamed on their general structure, as they do not assume all supports have the same size.

Running the tests with bigger support size revealed even more behaviour. The same types of plots as before are shown on figures 7 to 11.

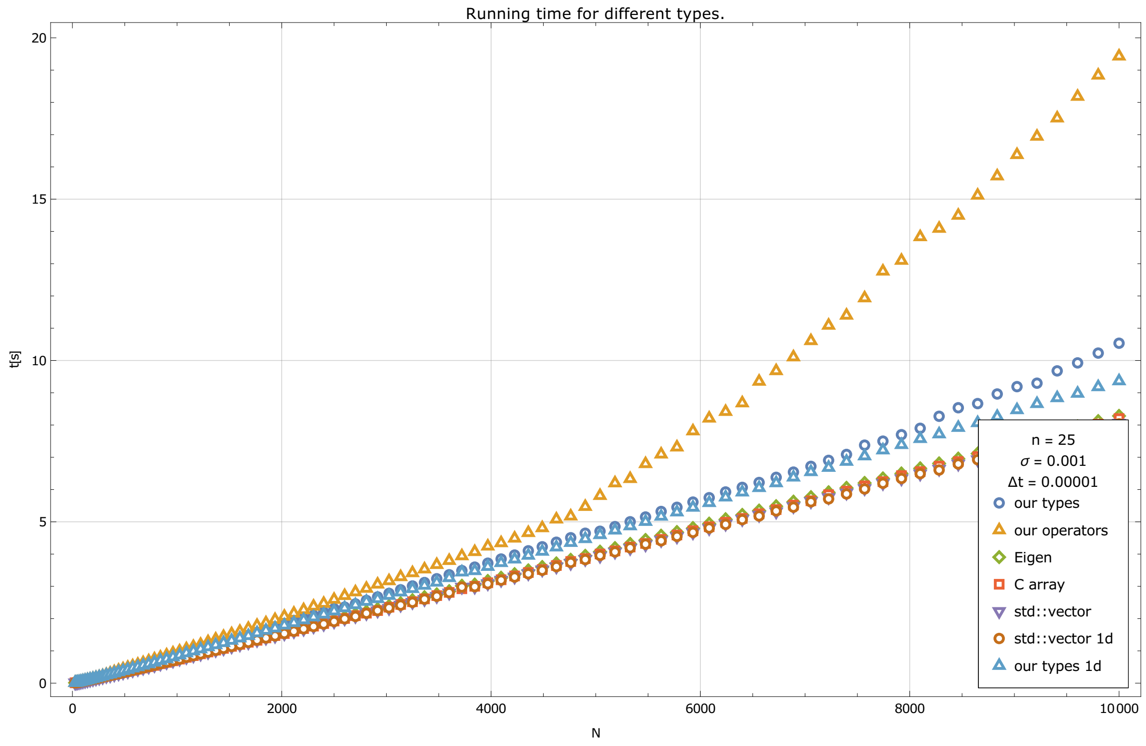

Figure 7: Initial running time with larger support size.

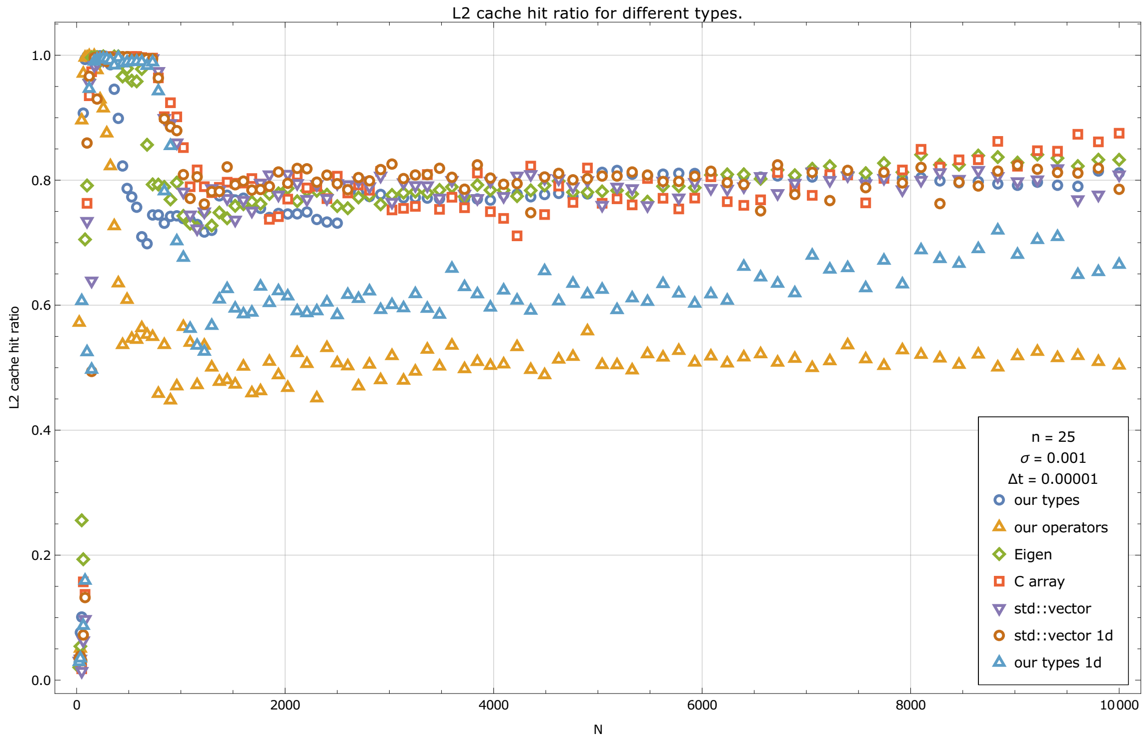

Figure 8: Initial L2 cache hit ratio with larger support size.

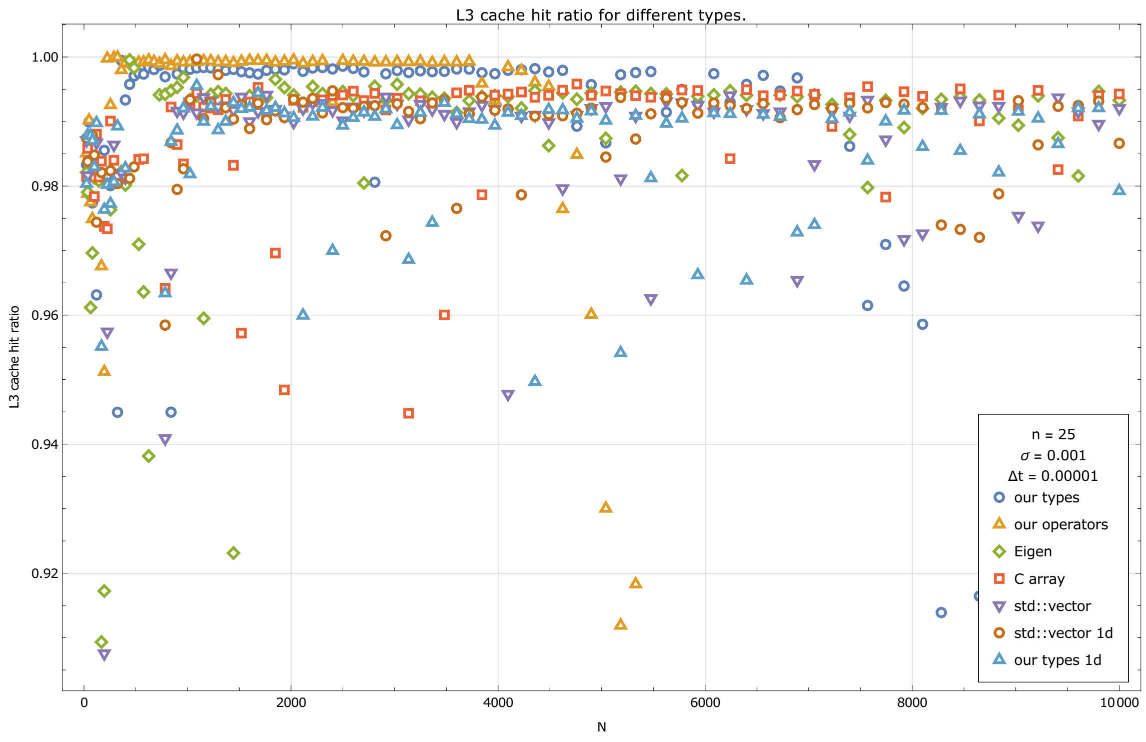

Figure 9: Initial L3 cache hit ratio with larger support size.

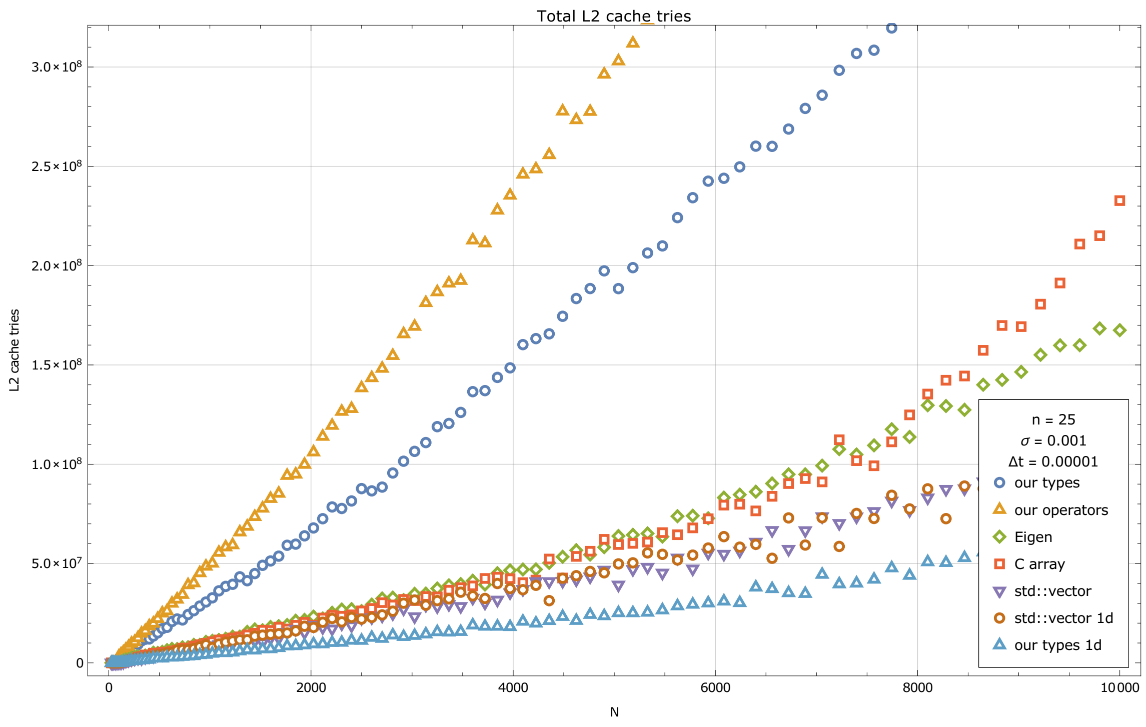

Figure 10: Initial L2 cache total access count with larger support size.

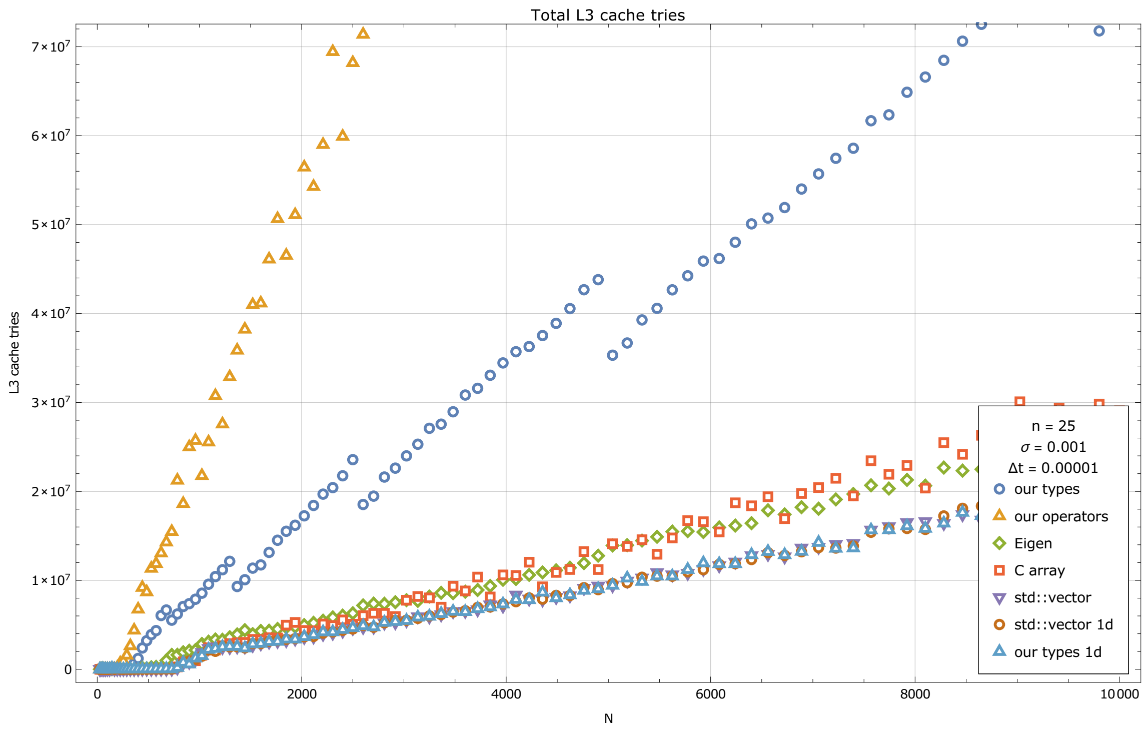

Figure 11: Initial L3 cache total access count with larger support size.

Of special interest is the break of time for our operators on figure 7 at around $N \approx 5000$. The time increase is believed to be the consequence of running out of L3 cache space. This also happens for some of the other types later on the graph. This slowdowns coincide with the declines of the L3 cache hit ratio curve on figure 9.

We have to answer a few questions:

1. Why do we have significantly different times for different types with 1d structure?
2. Why do we have significantly different times for different types with 2d structure?
3. What accounts for the difference between 1d and 2d types and how relevant is it?
4. Do slowdowns happen at expected node counts and are they well correlated with cache size?
5. Why do different types experience slowdowns at smaller node counts than others?
6. Why the hell are operators so much slower?

4 Fixes and second round of measurements

Measurements helped us bind a few inconsistencies in our code. C-array was using a wrong swap technique, which accounted for more memory accesses than necessary. Some types used `size_t` for indexes and other used `int`. This was one of the causes for different slowdowns, as array of `size_t`-s takes twice more space than an array of `int`-s. We fixed the whole code to use ints (and plan to do so throughout the whole library).

We also used Eigen suboptimally, not using their own `ArrayXii` class, which is implemented as 1d array. We also changed all matrices used to be row-major, to ensure cache friendliness, as we only do row traversals.

For every time, we copied support domain data `SD` and shape functions `SL` to local variables of the same type.

We believe the above accounts for differences among the similar types and answers questions 1 and 2.

Assuming that slowdowns are caused because we ran out of cache space, we looks at out memory use. Our fixes included using consistent types for indexes and decimal numbers, copying all data to locally variables, and always allocating arrays of the same size on the heap. This helped smooth the differences in slowdown starts for different types, answering question 5.

We currently have around 10 local `int` or `double` variables, two 1d arrays of N `double`-s, one 2d $N \times n$ array of `int`-s and one 2d array of `double`-s. This sums up to approximately

$$[N \times n \times (8+4) + N \times 8 + 64] \text{ bytes}$$

To answer question 4, we check at which node count do we fill the L2 or L3 cache memory and we expect that the slowdown will appear around the same node count. The expected results are calculated in table 1.

n	L2	L3
5	3 350	105 000
13	1 500	46 500
25	800	23 300

Table 1: Expected node counts at which slowdowns should happen.

To answer question 6 we need to look at operators structure a bit closer. Operators are a class, repsresenting a collestion of operators (of which we only use laplace). The compute and store all first derivatives and laplace shape functions. When called for a certain node, they perform the innermost for loop in code above. As they were planned to be as general as possible, we assumed that we do not know the size of the shape functions and used our 2d types for storage. We changed that to 1d heap allocated C-arrays, storing only pointers, changed all methods to be constant and make appropriate variables static. With some loss of generality (that we never even had elsewhere), we have made them comparable to the other types.

Measurements after changes above for 3 different support sizes, can be seen on figures 12 to 26.

We see that L3 and L2 cahce declines coincide very nicely with slowdowns on time graphs. The decline of L3 cahce hit ration in the $n = 25$ case in figure 24 happens at approximately the same node cont as we predicted. Similarly, our predictions match in the $n = 13$ case as well (fig. 19). All 1d types are grouped together and 2d types are grouped together, with only alight difference between them. We believe that this difference is due to an additional work of another lookup / dereference, which is large, compared to simple addition and multiplication used in 1d arrays. Operators are now the fastest (which is surprising. . .)

All solutions have also been tested and they return identical and correct results.

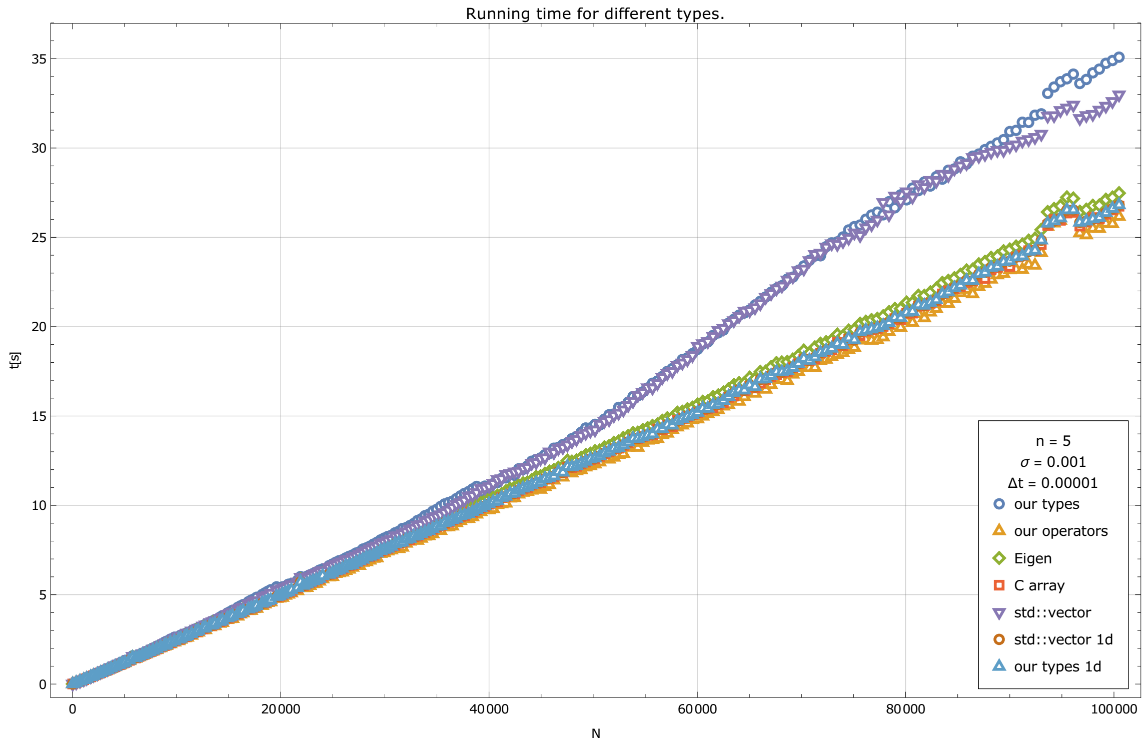

Figure 12: Final running time for $n = 5$.

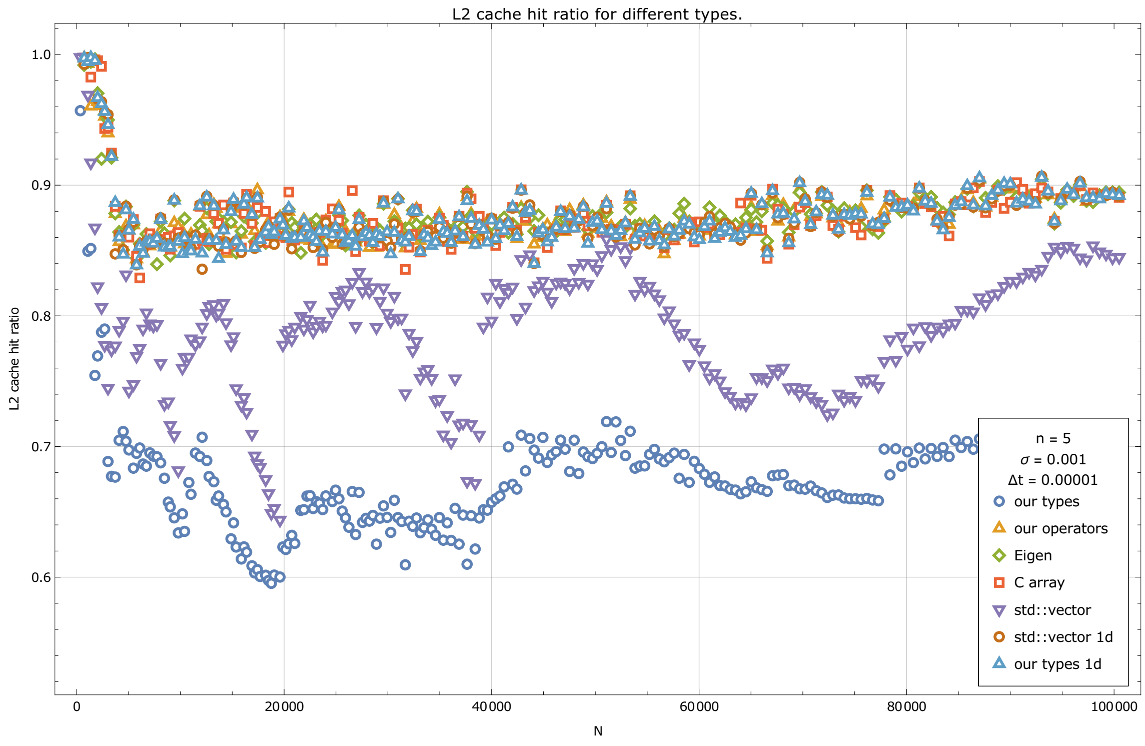

Figure 13: Final L2 cache hit ratio for $n = 5$.

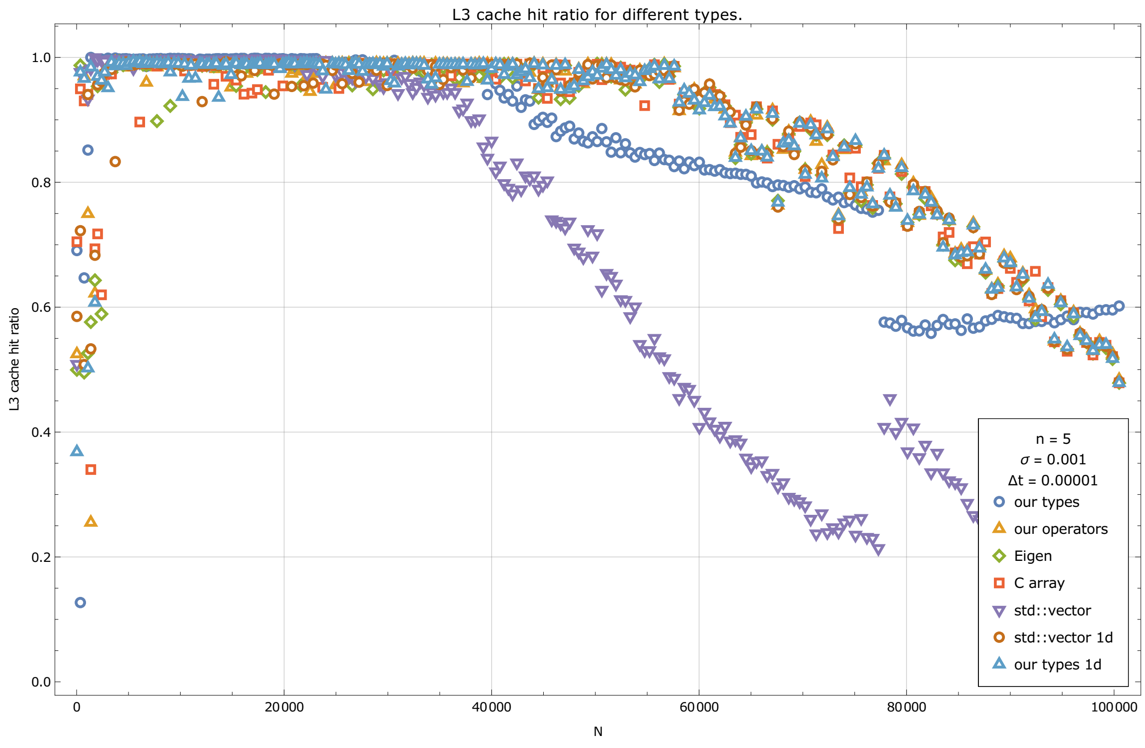

Figure 14: Final L3 cache hit ratio for $n = 5$.

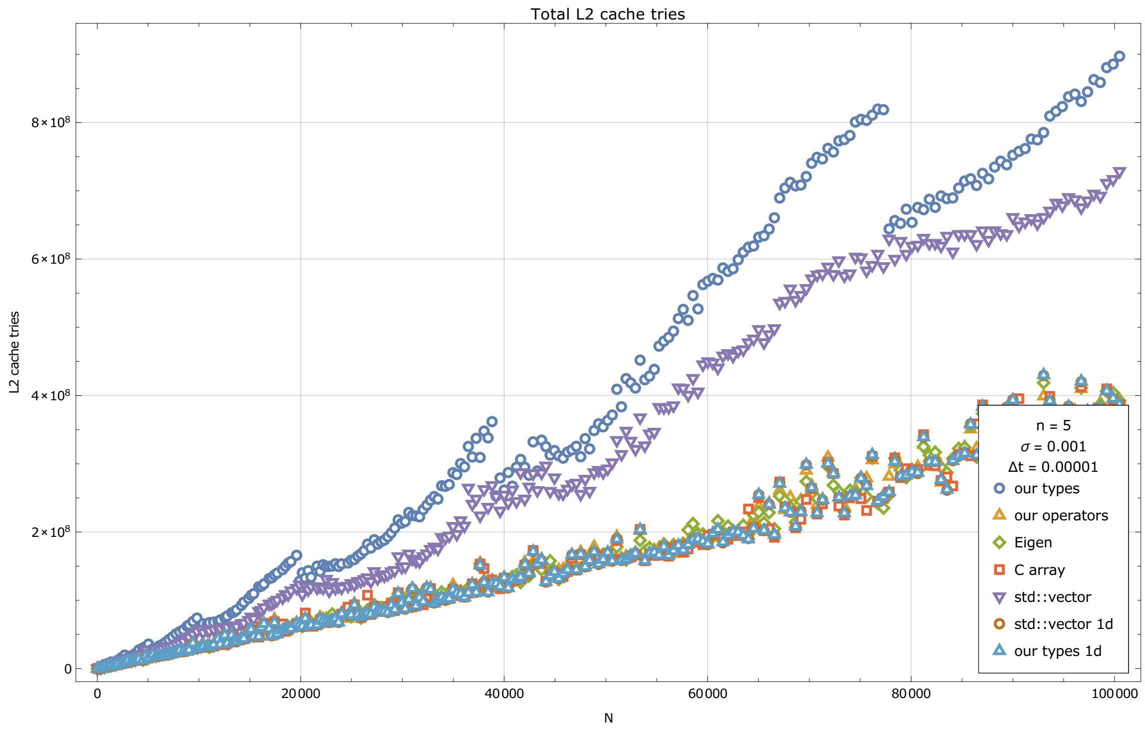

Figure 15: Final L2 cache total access count for $n = 5$.

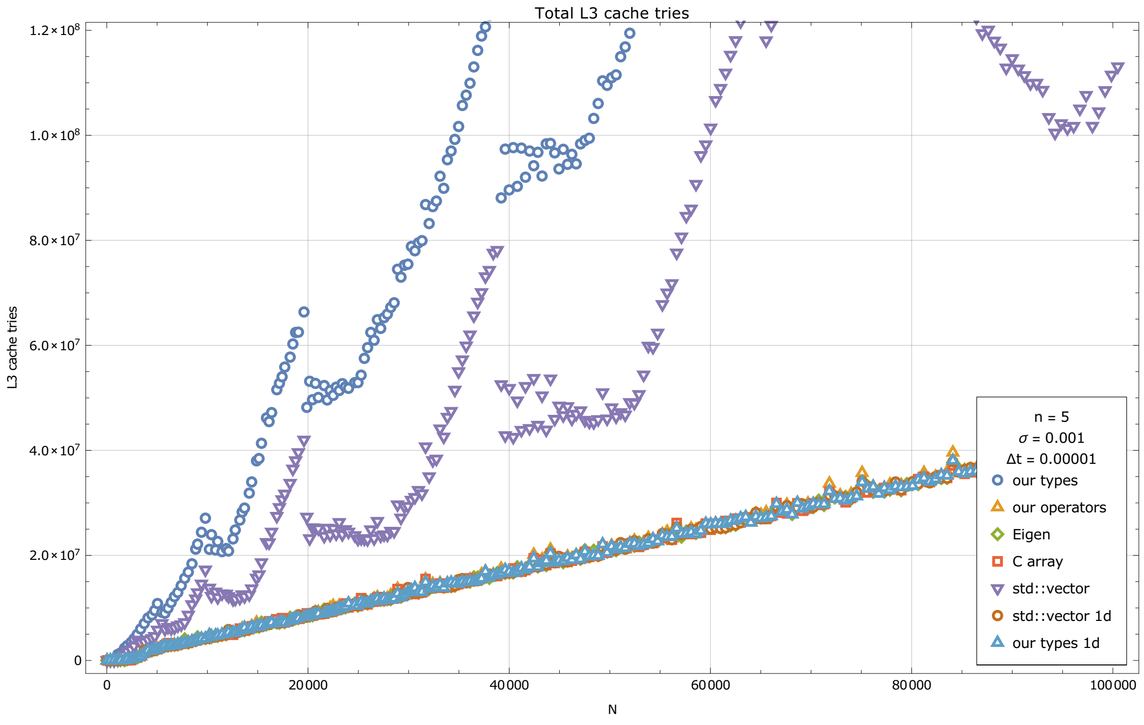

Figure 16: Final L3 cache total access count for $n = 5$.

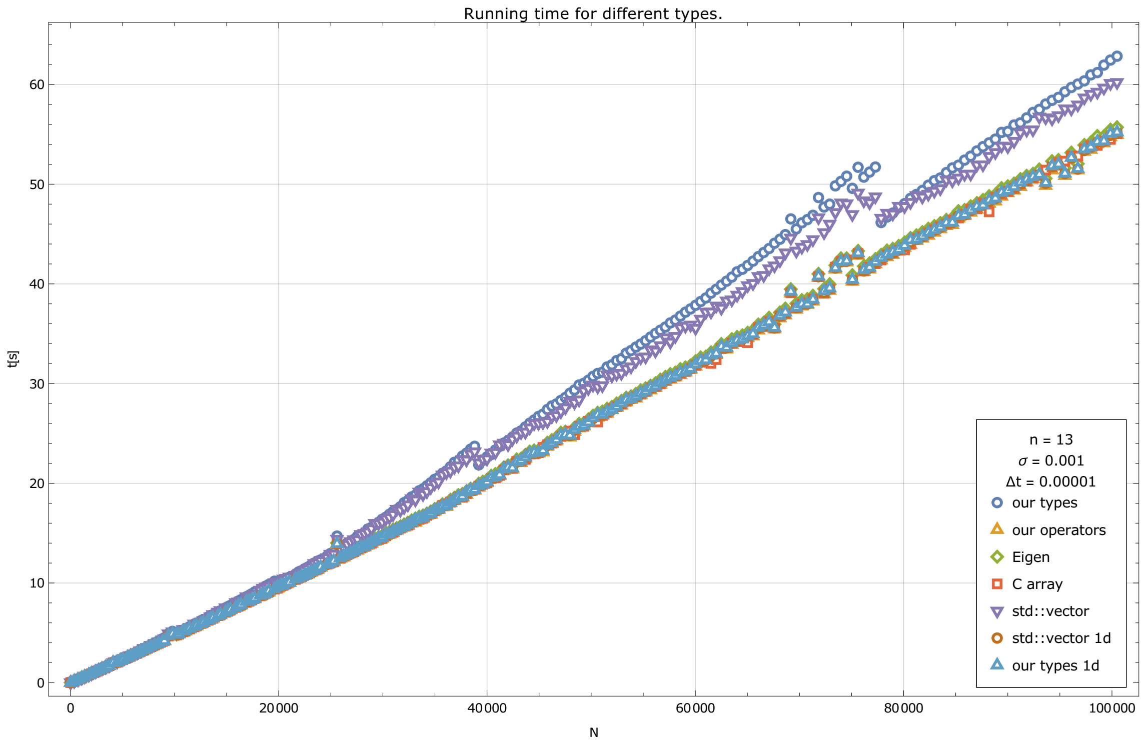

Figure 17: Final running time for $n = 13$.

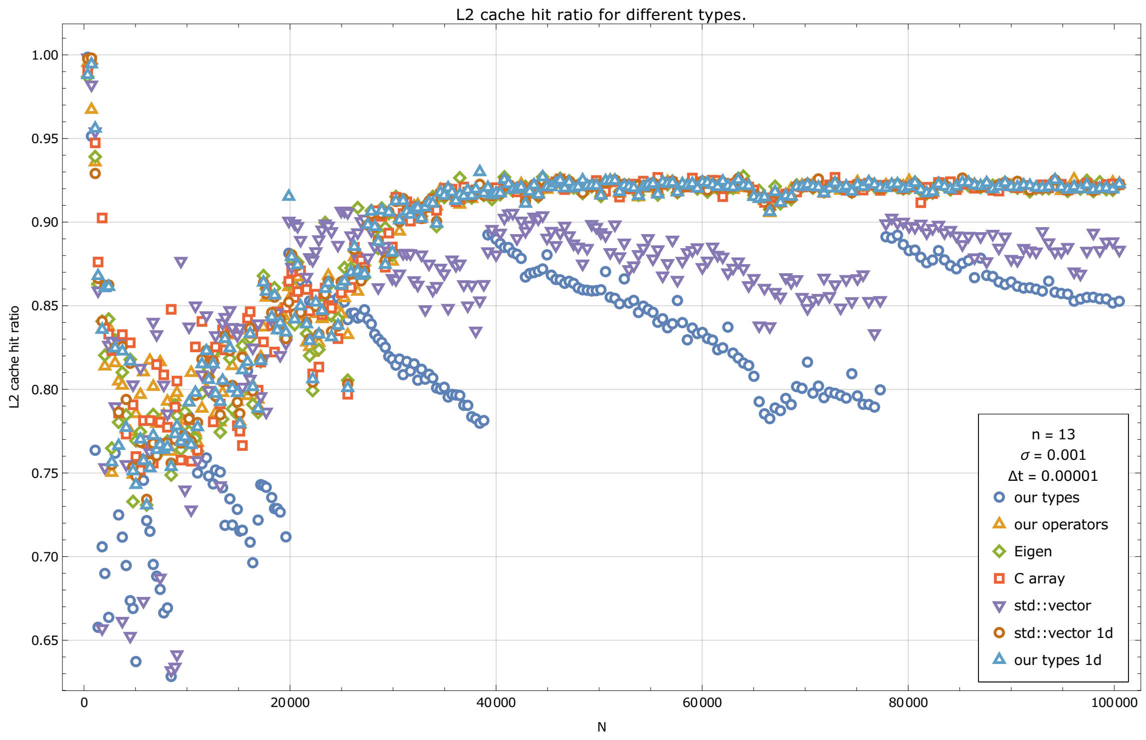

Figure 18: Final L2 cache hit ratio for $n = 13$.

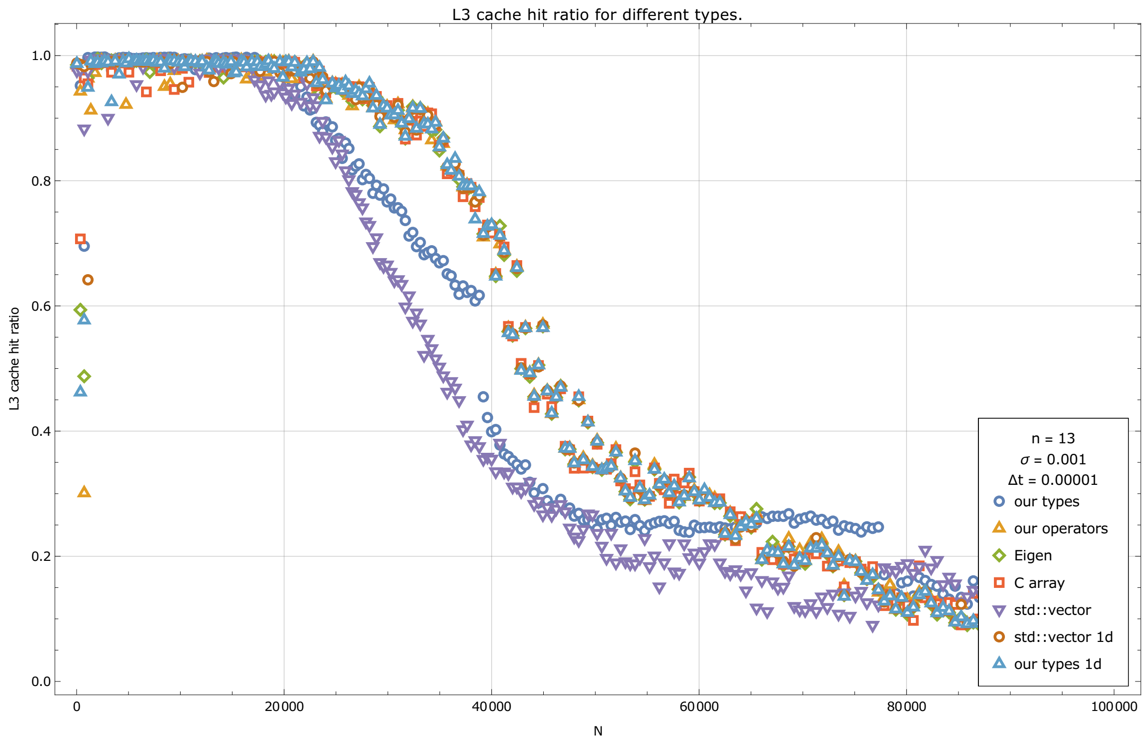

Figure 19: Final L3 cache hit ratio for $n = 13$.

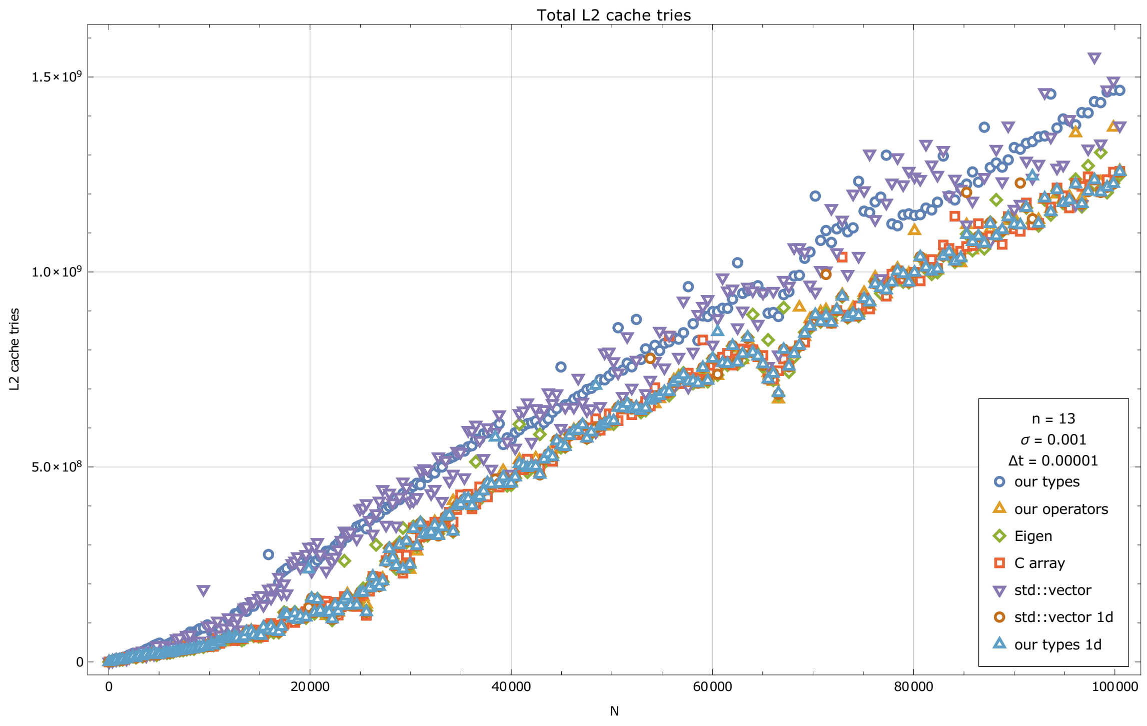

Figure 20: Final L2 cache total access count for $n = 13$.

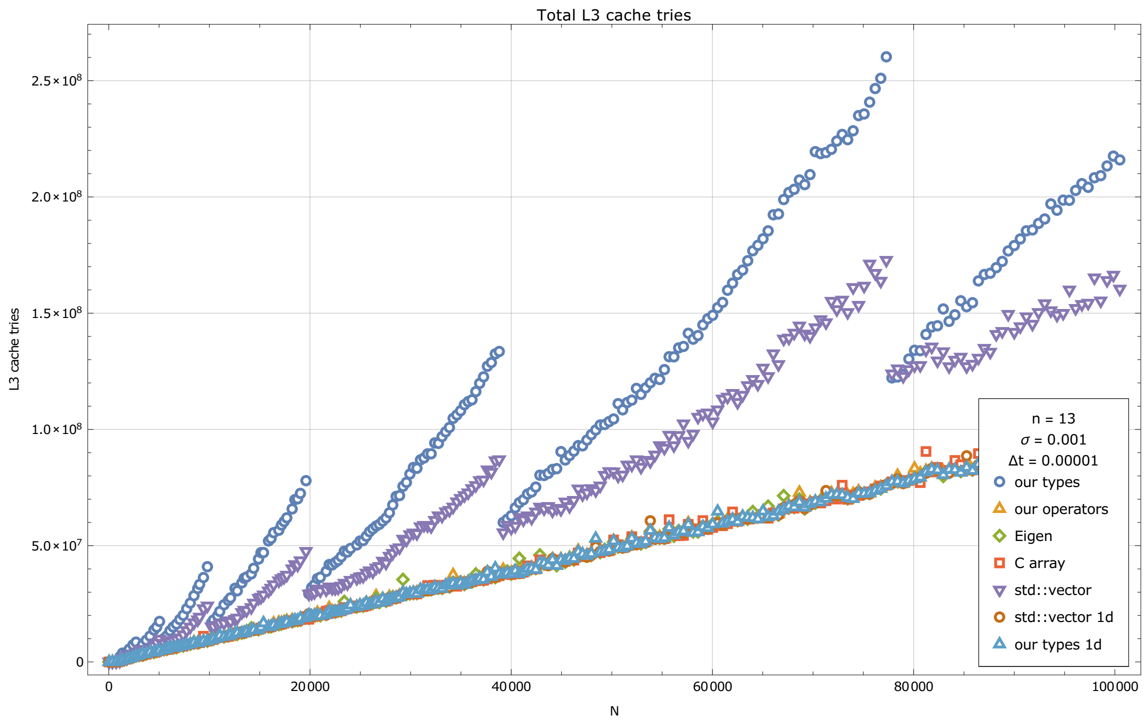

Figure 21: Final L3 cache total access count for $n = 13$.

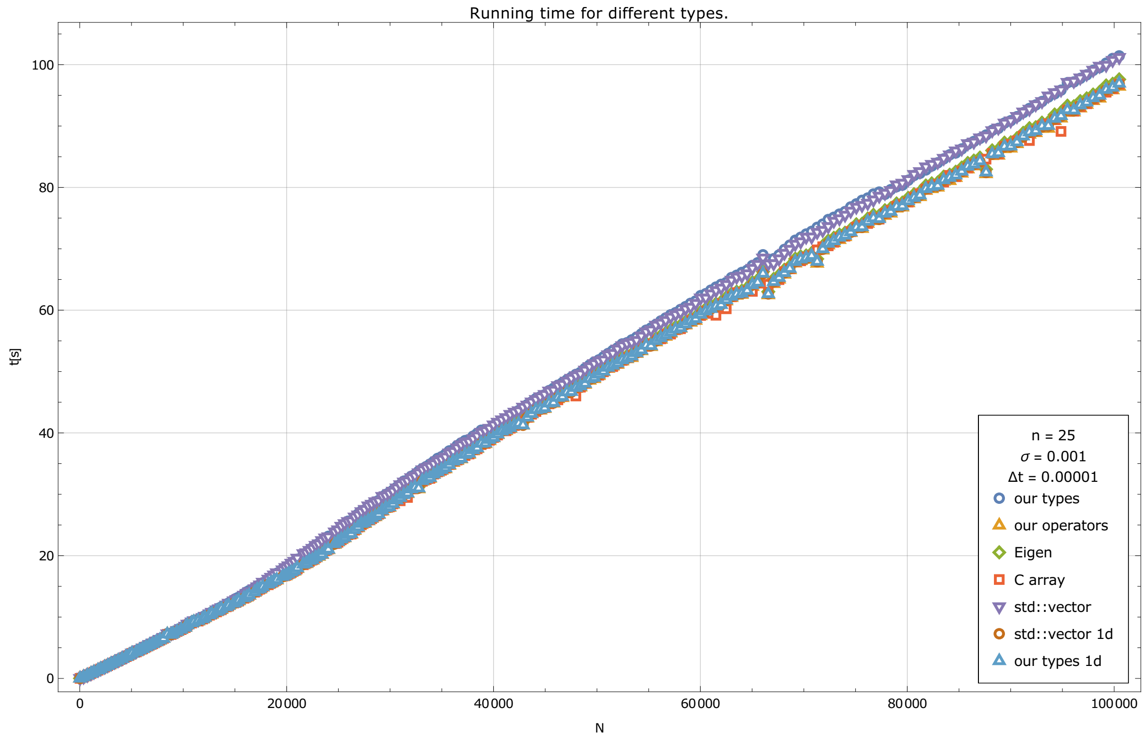

Figure 22: Final running time for $n = 25$.

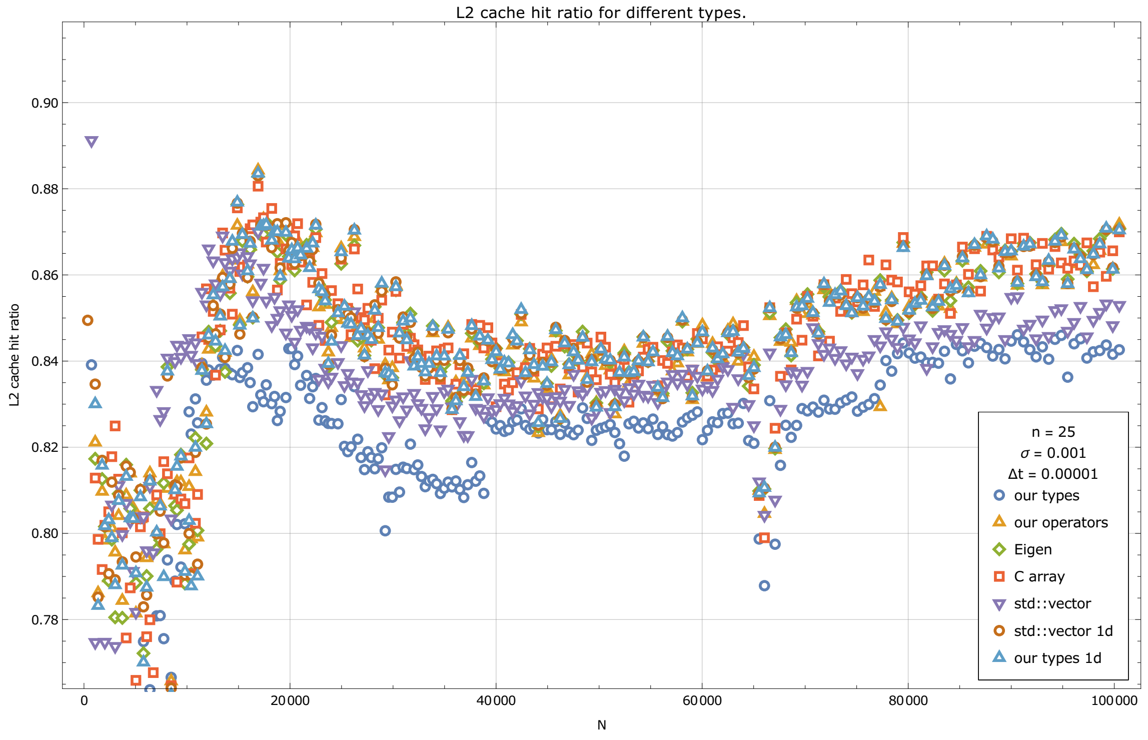

Figure 23: Final L2 cache hit ratio for $n = 25$.

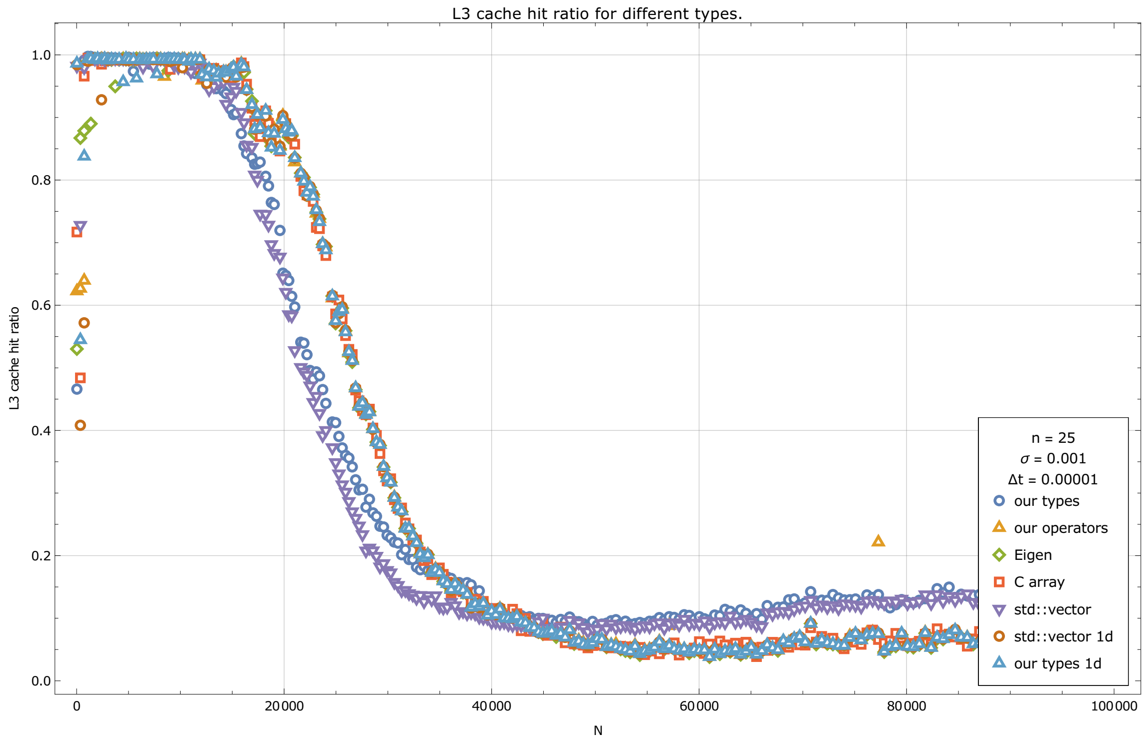

Figure 24: Final L3 cache hit ratio for $n = 25$.

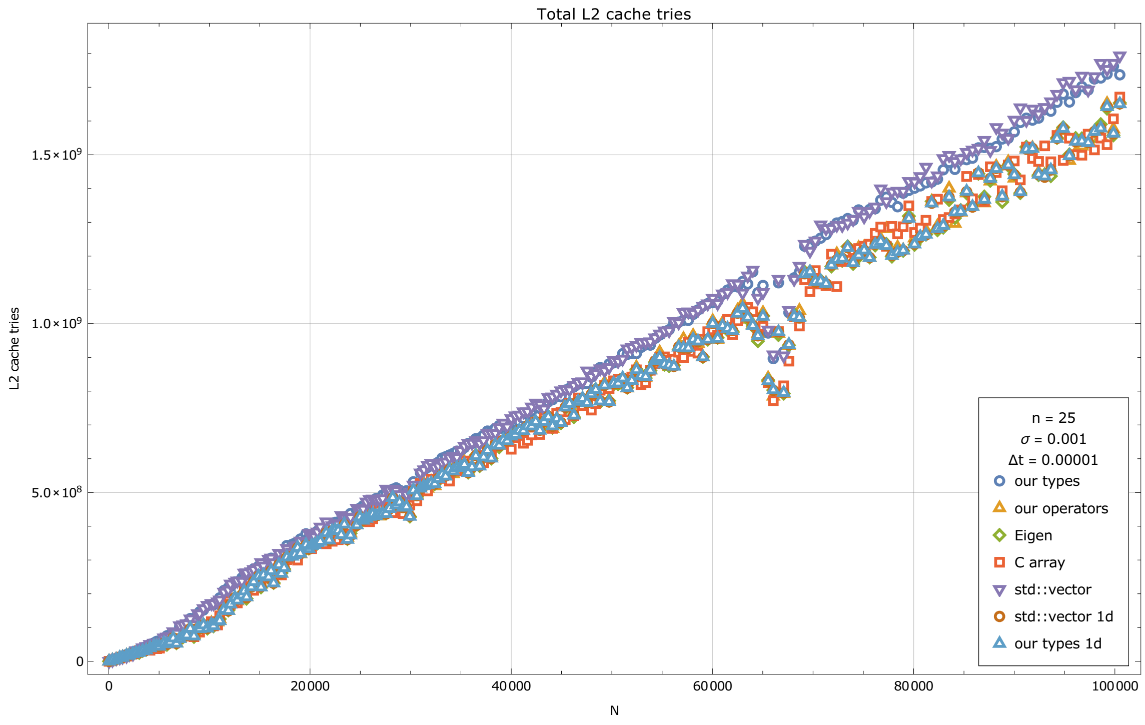

Figure 25: Final L2 cache total access count for $n = 25$.

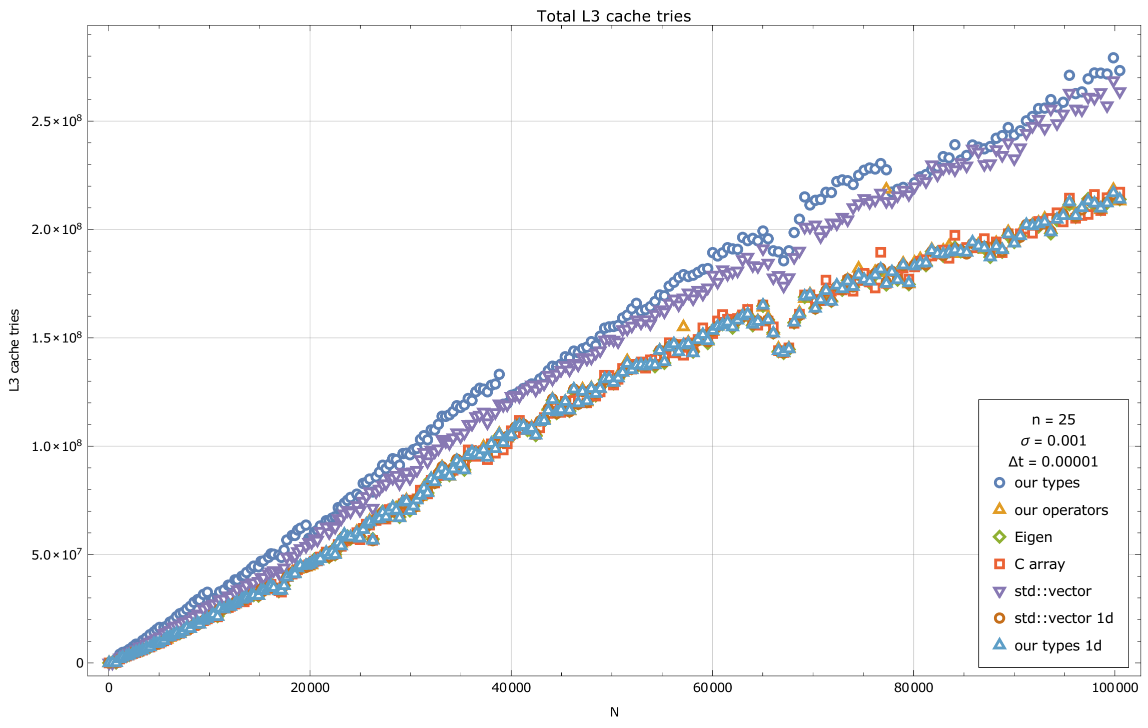

Figure 26: Final L3 cache total access count for $n = 25$.

5 Conclusions

Overall, we are satisfied with the results. We learned some things. Details matter. Caches are important.