Impacts of algorithms and data structures on multicore efficiency

(cache/memory optimization study using ThreadSpotter)

Marwa Al-Shandawely

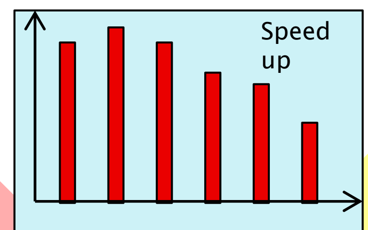
Agenda

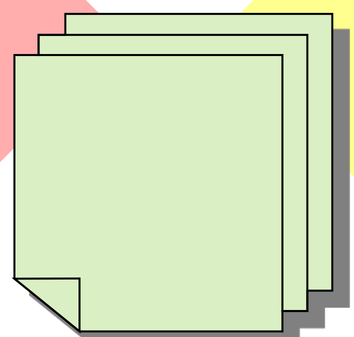
- Introduction
- Optimization process
- Memory perfomance problems
- Case study: GMRES for block toeplitz matrices
- Case study: Gaussian elimination

Introduction

- The gap between cores and memory speeds.
- Caches: fast and limited capacity.
- So keep it cached to get it fast!

Memory bandwidth
Memory latency
Data locality
Thread communication/ interaction





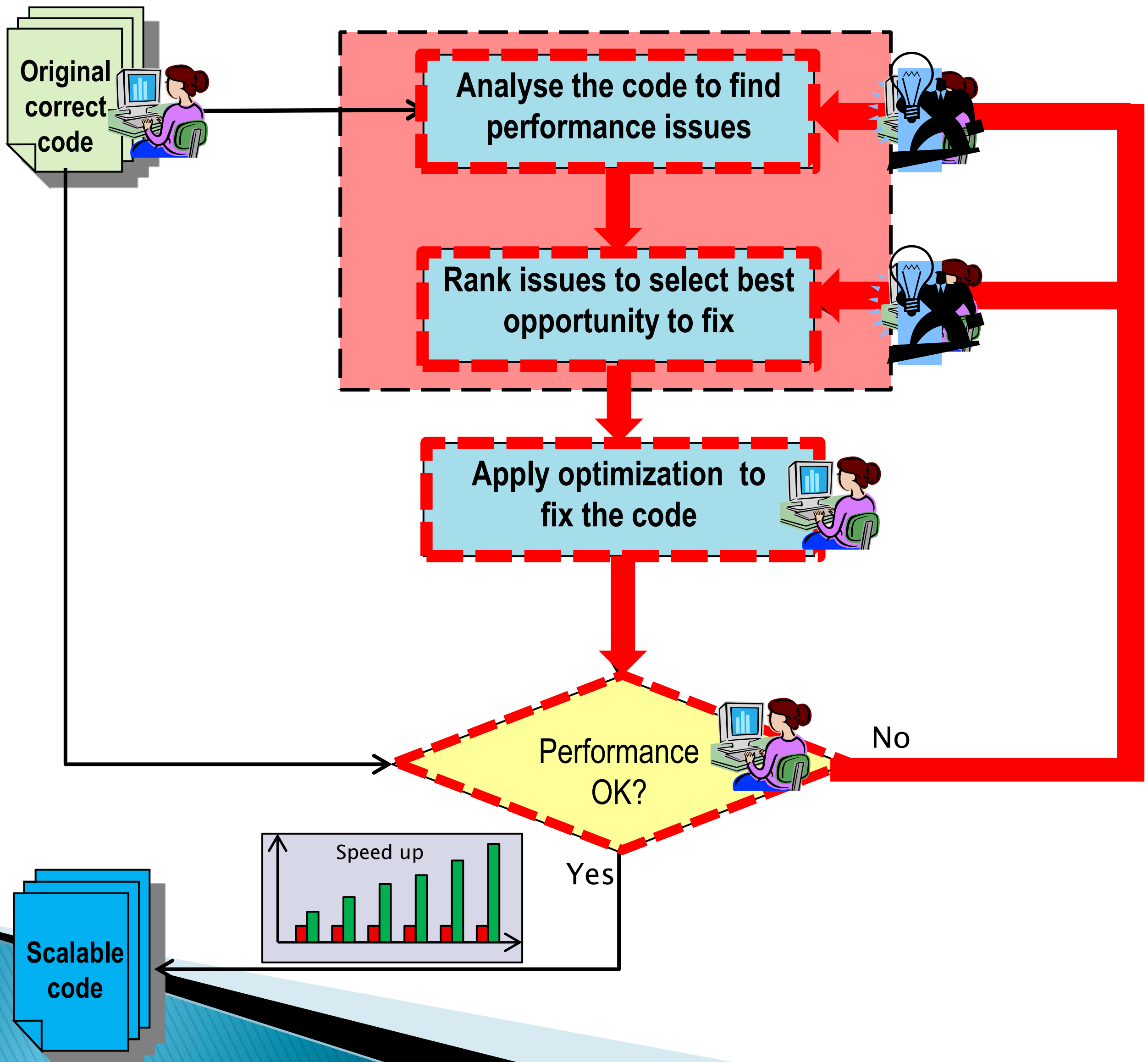
Algorithm
Time/ space compexity
Correctness
Lock-free
Parallel tasks

ThreadSpotter

- Cache memory optimization tool
- Analyzes memory bandwidth and latency, data locality and thread communications
- Pinpoints troublesome areas in source code
- Provides guidance towards a resolution
- Two steps: sampling application then report generation.

http://www.roguewave.com/products/threadspotter/resources/videos.aspx

Improving performance on multi-cores



Memory performance problems

- Data layout problems
  - Partially used structures.
  - Alignment problems.
  - Dynamic memory allocation.
- Data access pattern problems
  - Inefficient loop nesting.
  - Random access patterns.

Memory performance problems

- Data reuse opportunities
  - Blocking
  - Loop fusion.

- Multi-threading problems
  - False sharing
  - Poor communication problems.

Case study

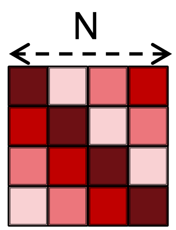
GMRES for block-toeplitz matrices

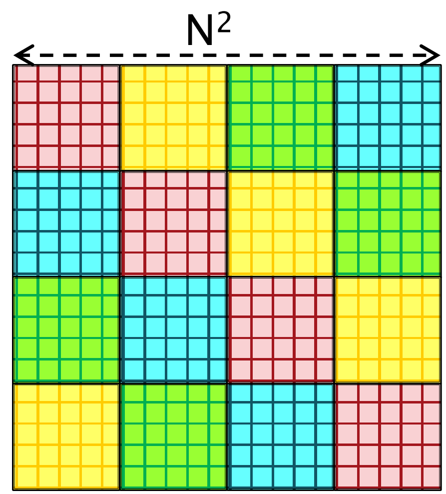
GMRES: *Iterative matrix inversion method*

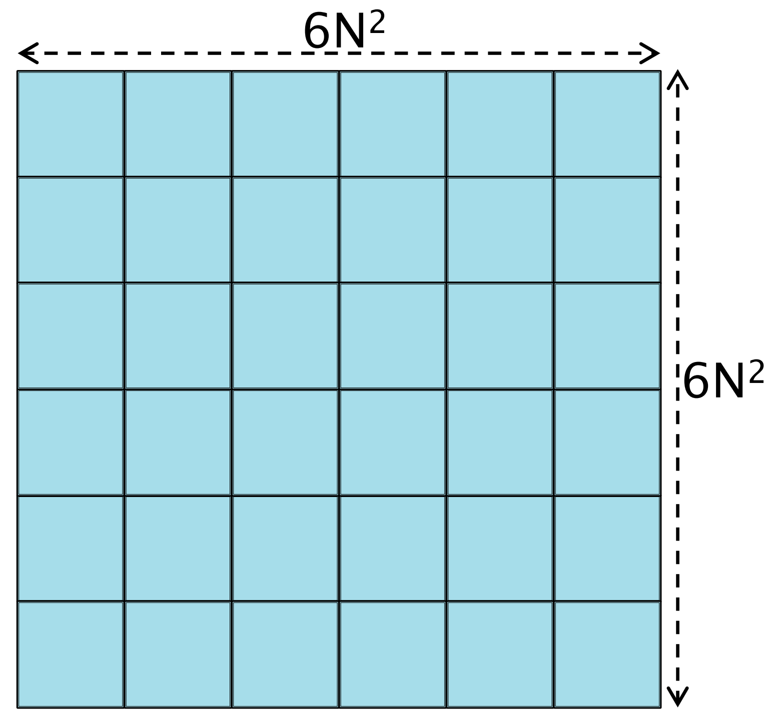
- Find $x_k \approx x$ that solves $Ax = b$ and hence minimizes the residual
  $$\|r_k\| = \|Ax_k - b\| \leq TOL$$

- The core calculations at each iteration k of GMRES require a matrix-vector multiplication $A.c_k$

Block-toeplitz matrices



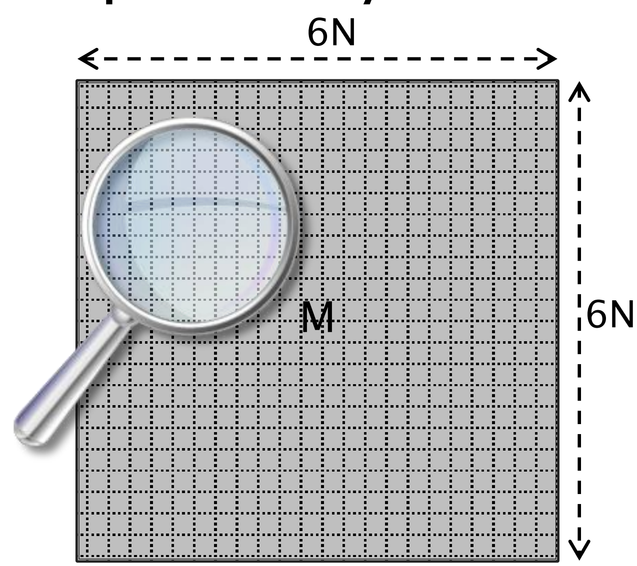




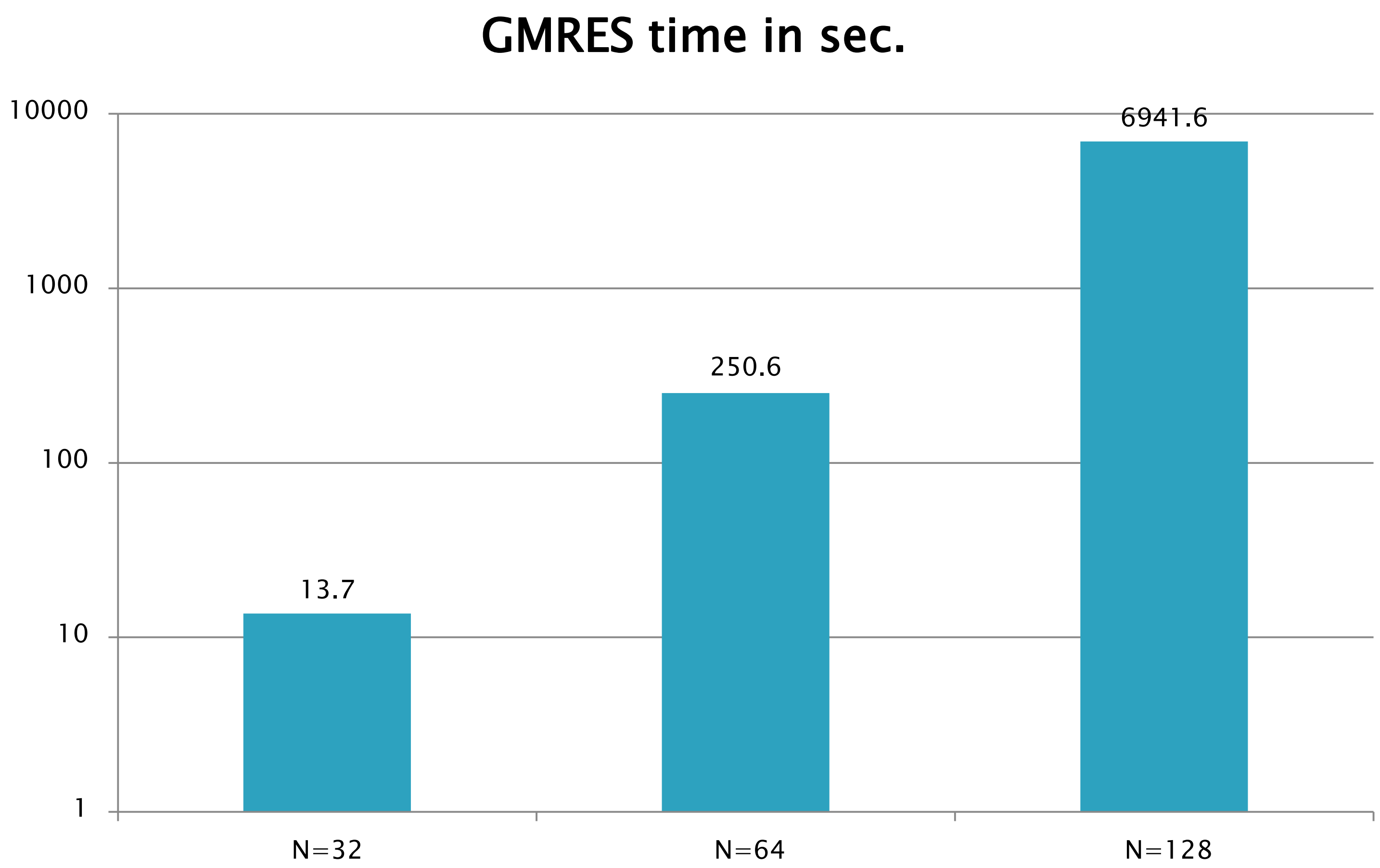
- Total size of A $=(6N^2)^2$
- A has $(6)^2$ different blocks.
- Typically max. N=128 and hence A is about 10 million elements
- Using the mask matrix (storing unique elements of A). M size = $(6N)^2$

First approach: The mask matrix M

- For the full matrix $\mathbf{A}_{(6N^2 \times 6N^2)}$, only a mask $\mathbf{M}_{(6N \times 6N)}$ is needed to be stored. If N=128 → A is $(98304)^2$ while M is $(768)^2$.
- Each element in **M** will appear N^2 times in **A** due to the circular block-toeplitz structure.
- Elements of **A** can be looked up in **M** by index calculations.



Why using M is not fast enough!

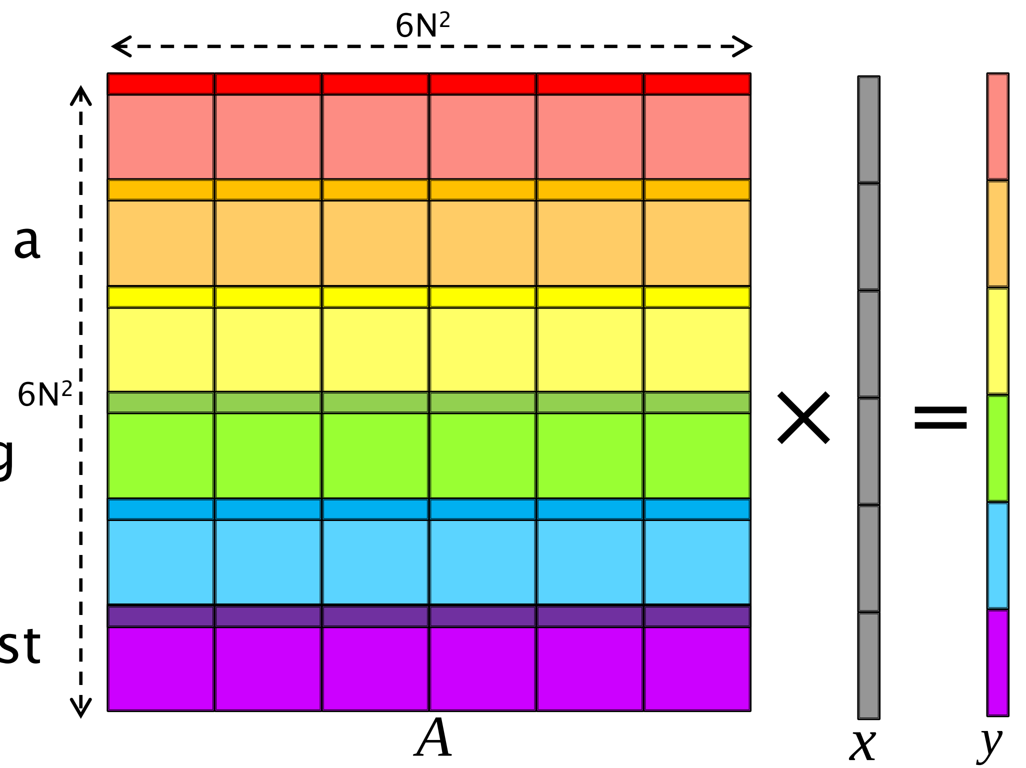


Report conclusions

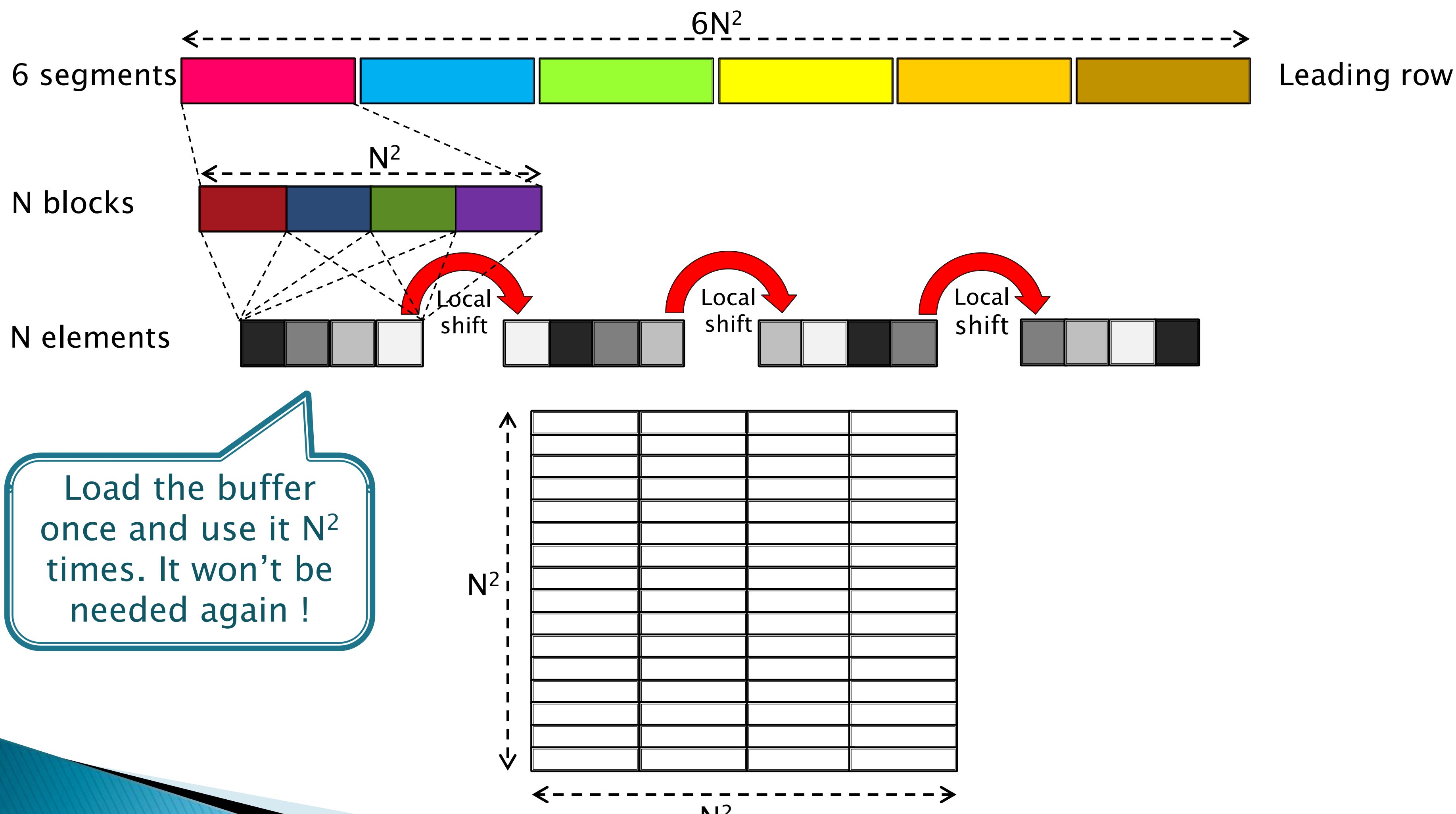
- Avoid random access pattern.
- Increase data reusability.
- Apply blocking mechanism.

Alternative representation of A

- Each segment $\mathbf{S_i}$ of **A**'s six-horizontal segments updates a distinct $\mathbf{Y_i}$ segment in **Y**.
- Due to **A**'s structure, knowing the first (leading) row of each $\mathbf{S_i}$ is sufficient to be stored and used to determine the rest of its rows.



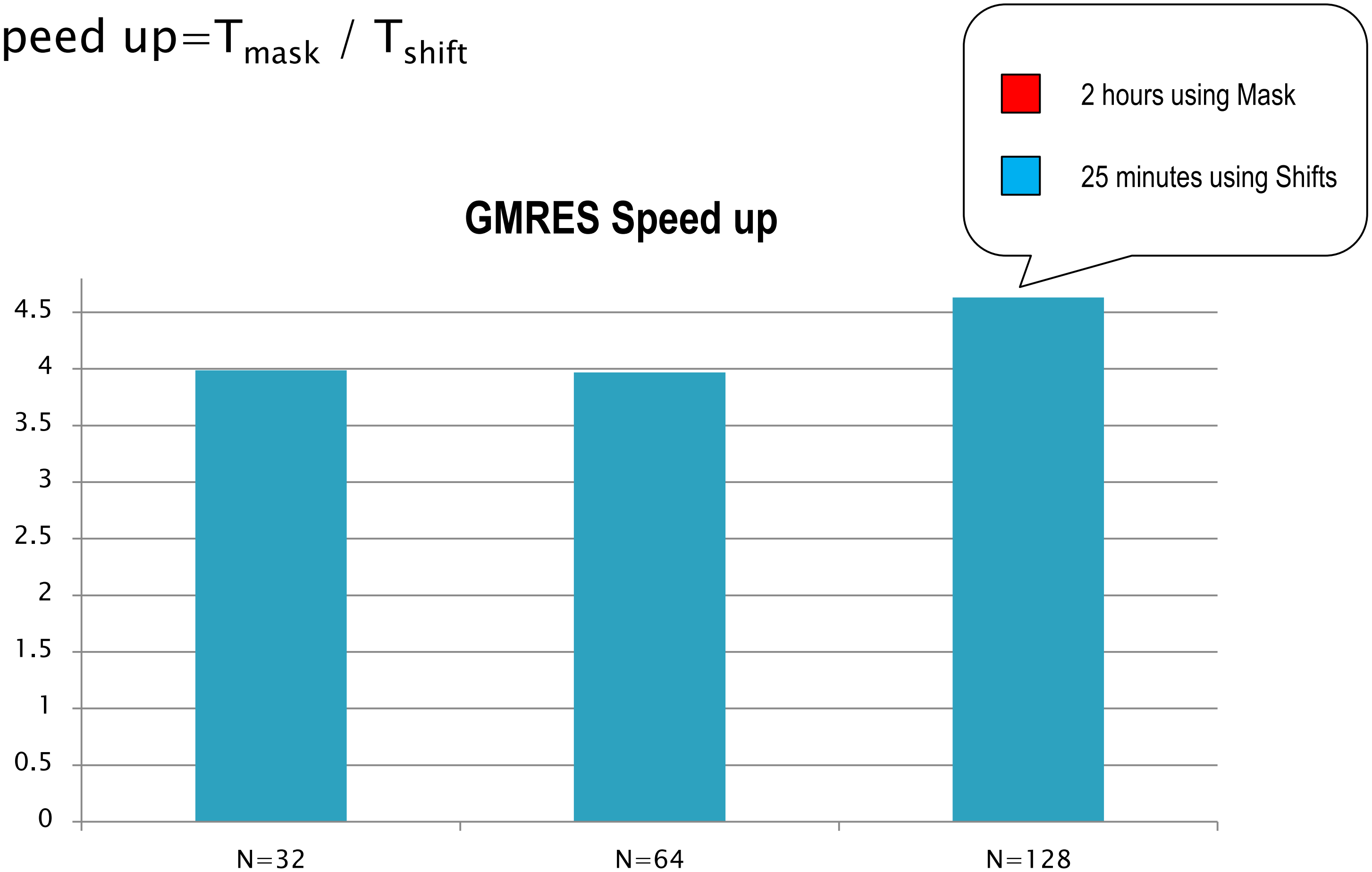
Data reusability through shifting



Load the buffer once and use it N^2 times. It won't be needed again !

Overall improvement

Speed up $= T_{mask} \,/\, T_{shift}$



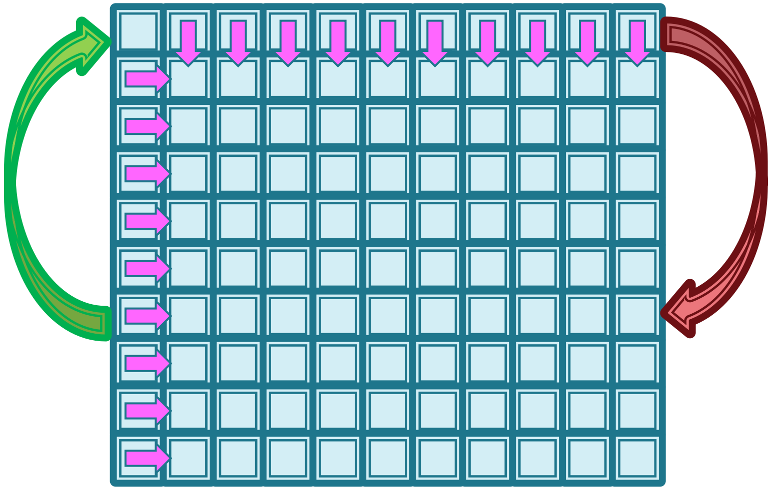
Case study

Gaussian Elimination with pivoting

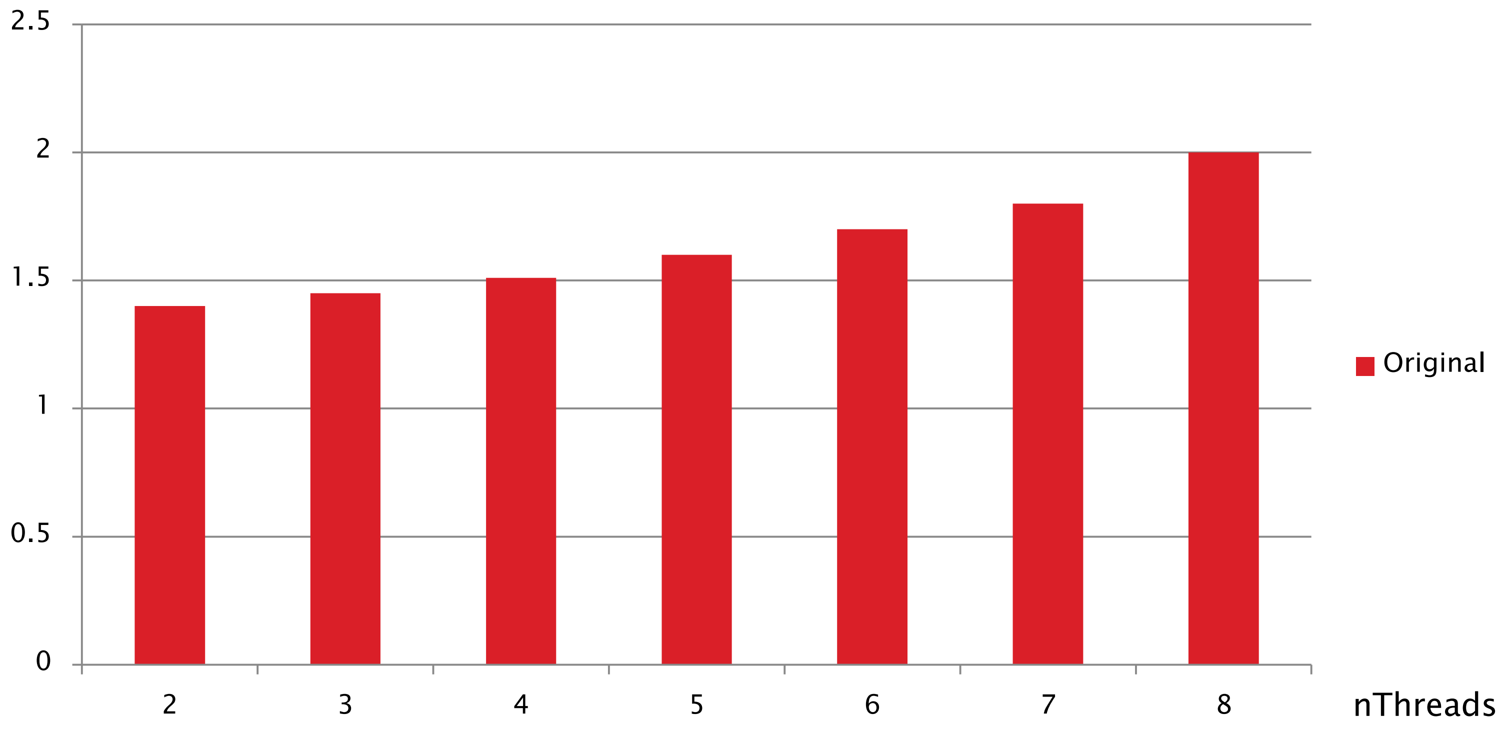
Overview of forward elimination

```
for i=1 to n-1
  find pivotPos in column i
  if pivotPos ≠ i
    exchange rows(pivotPos,i)
  end if

  for j=i+1 to n
    A(i,j) = A(i,j)/A(i,i)
  end for j
 !$omp parallel do private ( i ,j )
  for j=i+1 to n+1
    for k=i+1 to n
      A(k,j)=A(k,j)-A(k,i)×A(i,j)
    end for k
  end for j
end for i
```



First approach speed up



What went wrong?!

- The original algorithm requires pivot columns to be prepared in order while the whole matrix is accessed for each pivot column.

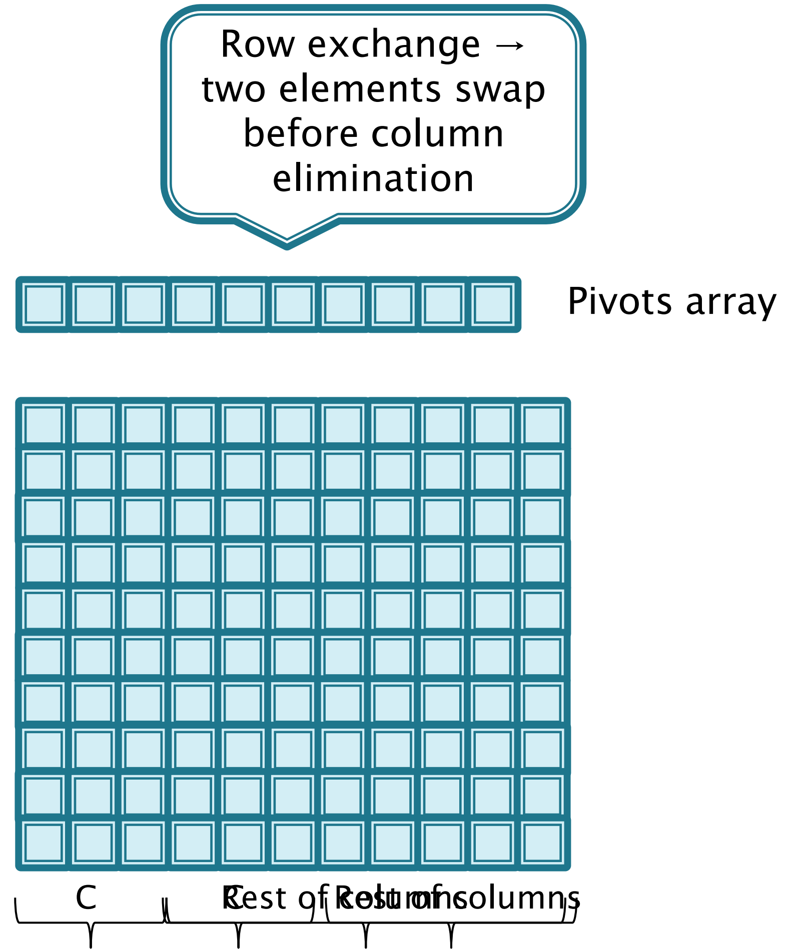
- The cache is evicted many times for each iteration and there is no reuse of data in the cache.

Making things right!

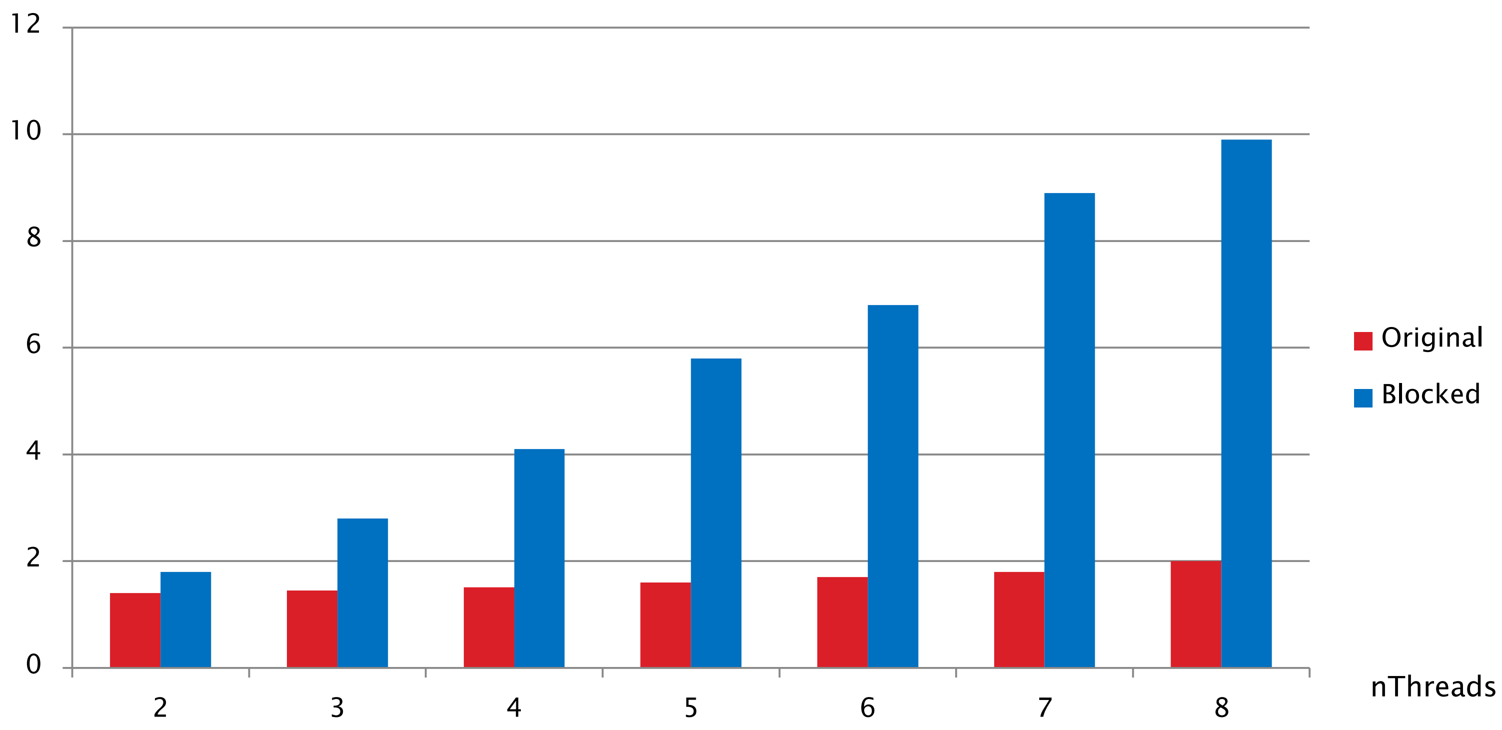
- Each column is an accumulation of eliminations using previous columns!
- Make more pivots available each step and eliminate each column using several pivots while it is in the cache.

Blocking GE

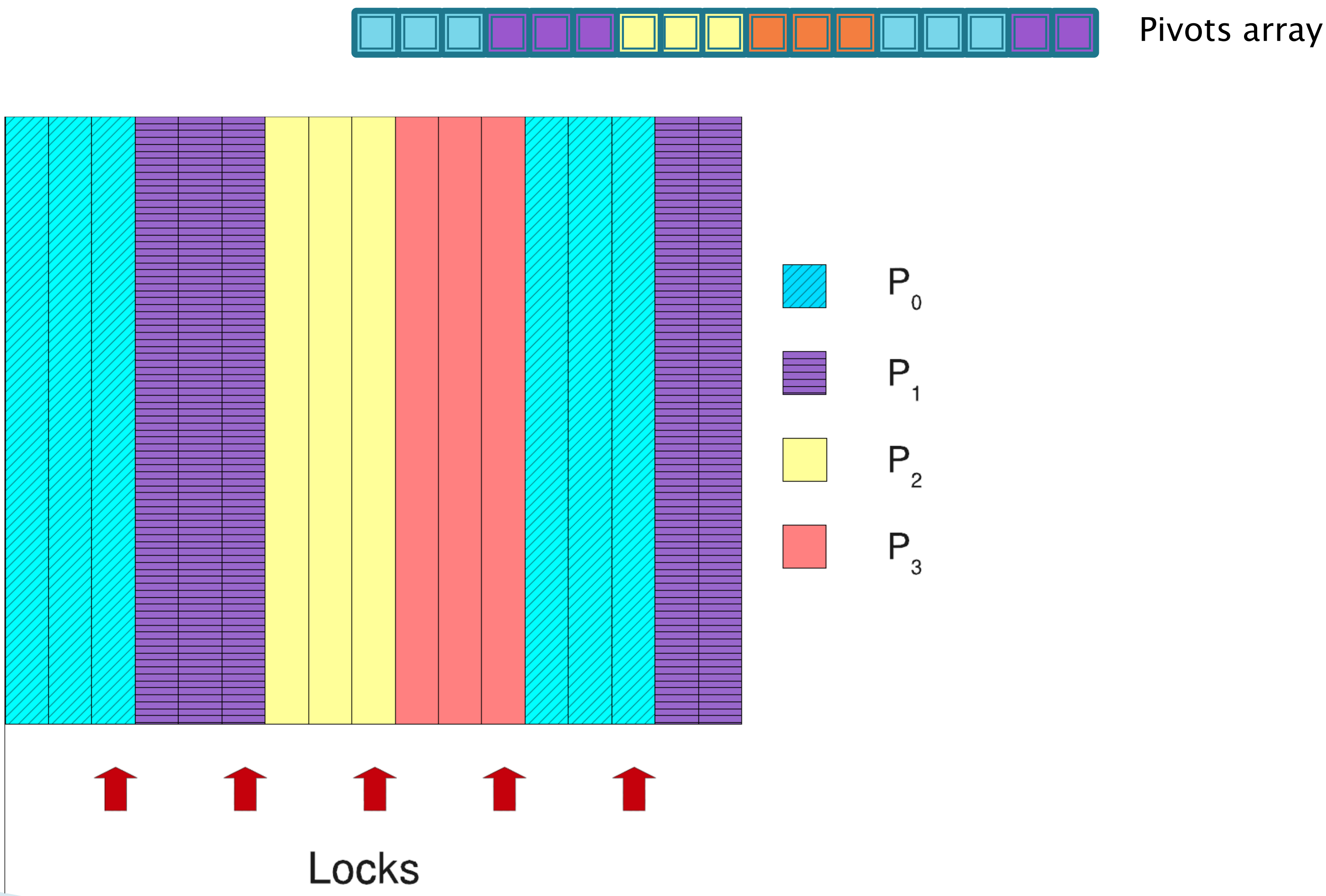
```
for k=1 to n-1, step C
    BlockEnd=min(k+C-1,n)
    GE on A(k:n,k:BlockEnd) &
    Store C pivots' positions
  !$omp parallel do private ( i ,j )
    for each column j after BlockEnd
        for i=k to BlockEnd
            swap using pivots(i)
            elimination i on j
        end for i
    end for each j
End for k
```



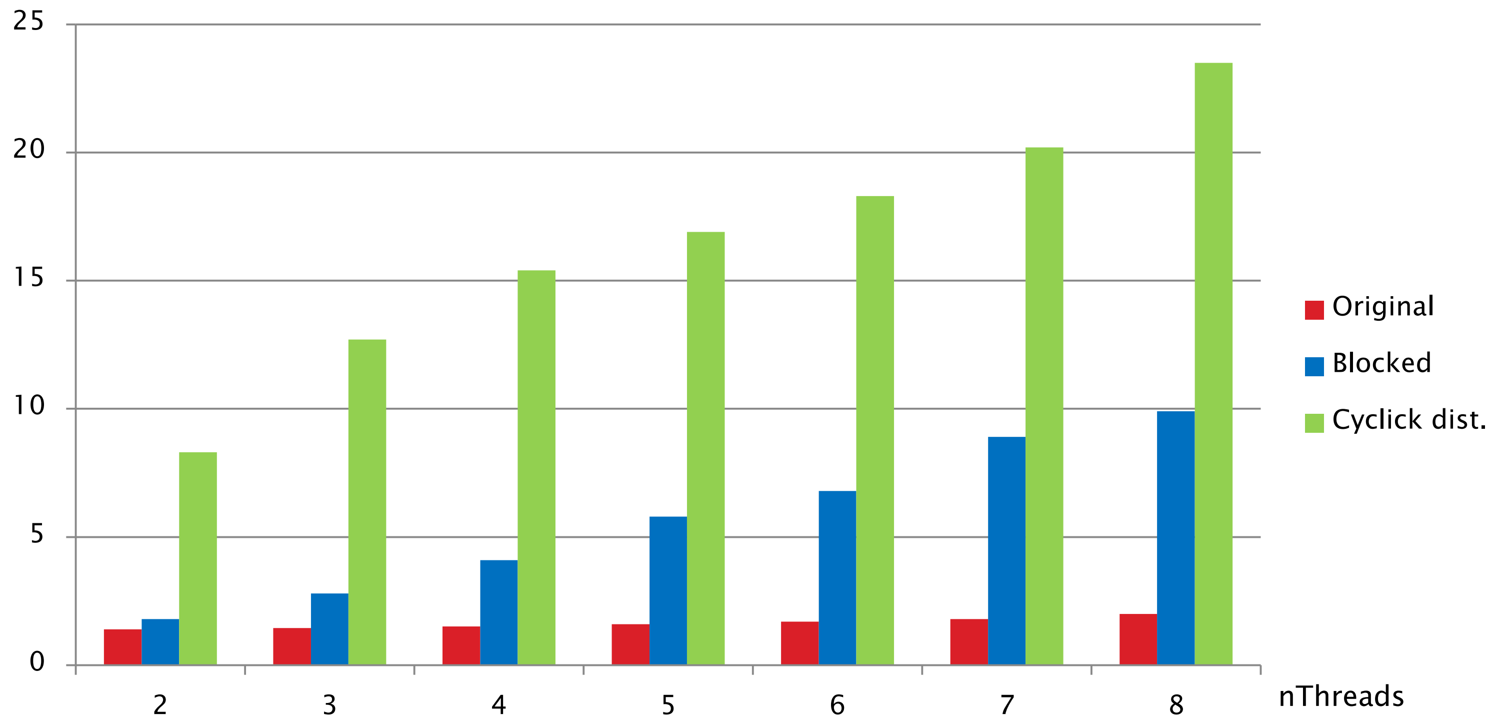
Speed up w.r.t sequential time



Cyclic column distribution



Improved Performance



Conclusions

- Scalable performance on multicores is highly dependent on application implementation, data layout and access patterns.
- The use of smart tools becomes crucial to deal with complex structures, separated code/data files and multithreaded applications.
- Cache and memory access optimization techniques is vital for performance despite the loss of readability.