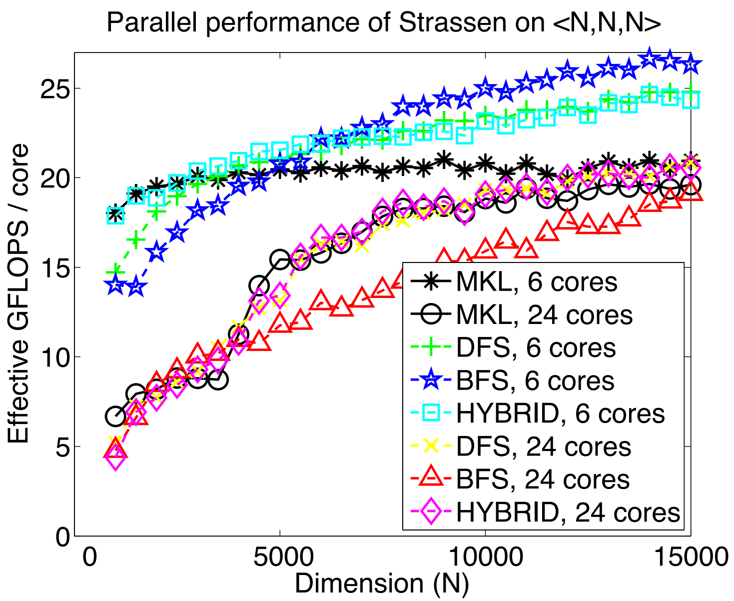
A FRAMEWORK FOR PRACTICAL FAST MATRIX MULTIPLICATION

arXiv: 1409.2908



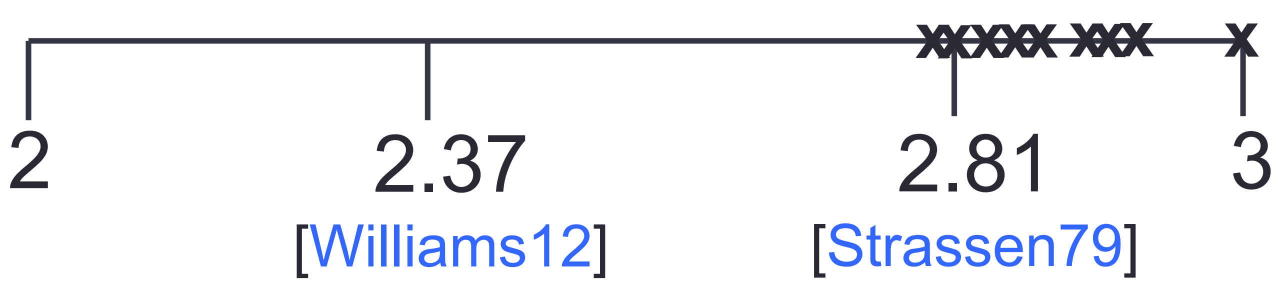
Austin Benson (arbenson@stanford.edu), ICME, Stanford

Grey Ballard, Sandia National Laboratories

BLIS Retreat, September 26, 2014

Fast matrix multiplication: bridging theory and practice

- There are a number of Strassen-like algorithms for matrix multiplication that have only been “discovered” recently. [Smirnov13], [Benson&Ballard14]
- We show that they can achieve higher performance with respect to MKL (sequential and sometimes in parallel).
- We use code generation to do extensive prototyping. There are several practical issues, and there is plenty of room for improvement (lots of expertise at UT to help here!)



Strassen's algorithm

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$S_1 = A_{11} + A_{22} \qquad T_1 = B_{11} + B_{22}$$
$$S_2 = A_{21} + A_{22} \qquad T_2 = B_{11}$$
$$S_3 = A_{11} \qquad T_3 = B_{12} - B_{22}$$
$$S_4 = A_{22} \qquad T_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{12} \qquad T_5 = B_{22}$$
$$S_6 = A_{21} - A_{11} \qquad T_6 = B_{11} + B_{12}$$
$$S_7 = A_{12} - A_{22} \qquad T_7 = B_{21} + B_{22}$$

$$M_r = S_r \cdot T_r$$
$$1 \leq r \leq 7$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Key ingredients of Strassen's algorithm

- 1. Block partitioning of matrices (**<2, 2, 2>**)
- 2. **Seven** linear combinations of sub-blocks of A
- 3. **Seven** linear combinations of sub-blocks of B
- 4. **Seven** matrix multiplies to form M_r (recursive)
- 5. Linear combinations of M_r to form C_{ij}

Key ingredients of fast matmul algorithms

- 1. Block partitioning of matrices (**<M, K, N>**)
- 2. **R** linear combinations of sub-blocks of A
- 3. **R** linear combinations of sub-blocks of B
- 4. **R** matrix multiplies to form M_r (recursive)

  R < MKN → faster than classical
- 5. Linear combinations of M_r to form C_{ij}

“Outer product” fast algorithm

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \end{bmatrix}$$

- **<4, 2, 4>** partitioning
- **R = 26** multiplies (**< 4 * 2 * 4 = 32**)

  → 23% speedup per recursive step (if everything else free)
- Linear combinations of A_{ij} to form S_r: 68 terms
- Linear combinations of B_{ij} to form T_r: 52 terms
- Linear combinations of M_r to form C_{ij}: 69 terms

Discovering fast algorithms is a numerical challenge

- Low-rank tensor decompositions lead to fast algorithms
- Tensors are small, but we need exact decompositions
  → NP-hard
- Use alternating least squares with regularization and rounding tricks [Smirnov13], [Benson&Ballard14]
- We have around 10 fast algorithms for <M, K, N> decompositions. Also have permutations, e.g., <K, M, N>.

Algorithm base case	Multiples (fast)	Multiplies (classical)	speedup per recursive step	exponent
$\langle 2,2,3\rangle$	11	12	9%	2.89
$\langle 2,2,5\rangle$	18	20	11%	2.89
$\langle 2,2,2\rangle$ [Strassen69]	7	8	14%	2.81
$\langle 2,2,4\rangle$	14	16	14%	2.85
$\langle 3,3,3\rangle$	23	26	17%	2.85
$\langle 2,3,3\rangle$	15	18	20%	2.81
$\langle 2,3,4\rangle$	20	24	20%	2.83
$\langle 2,4,4\rangle$	26	32	23%	2.82
$\langle 3,3,4\rangle$	29	36	24%	2.82
$\langle 3,4,4\rangle$	38	48	26%	2.82
$\langle 3,3,6\rangle$ [Smirnov13]	40	54	35%	2.77

Code generation lets us prototype algorithms quickly

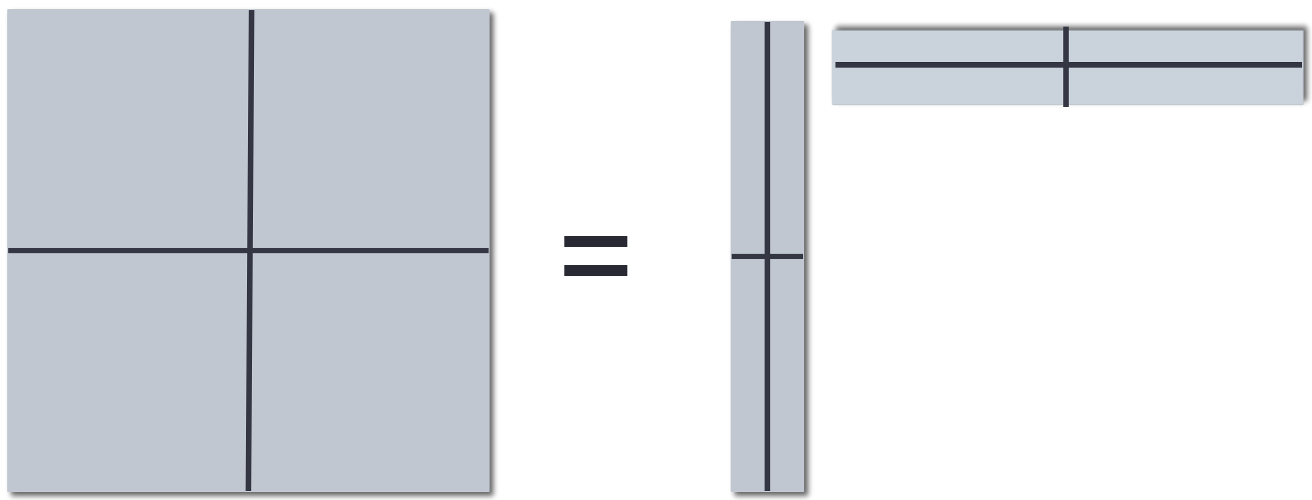
- We have compact representation of many fast algorithms:
  1. dimensions of block partitioning (<M, K, N>)
  2. linear combinations of sub-blocks (S_r, T_r)
  3. linear combinations of M_r to form C_{ij}

- We use code generation to rapidly prototype fast algorithms

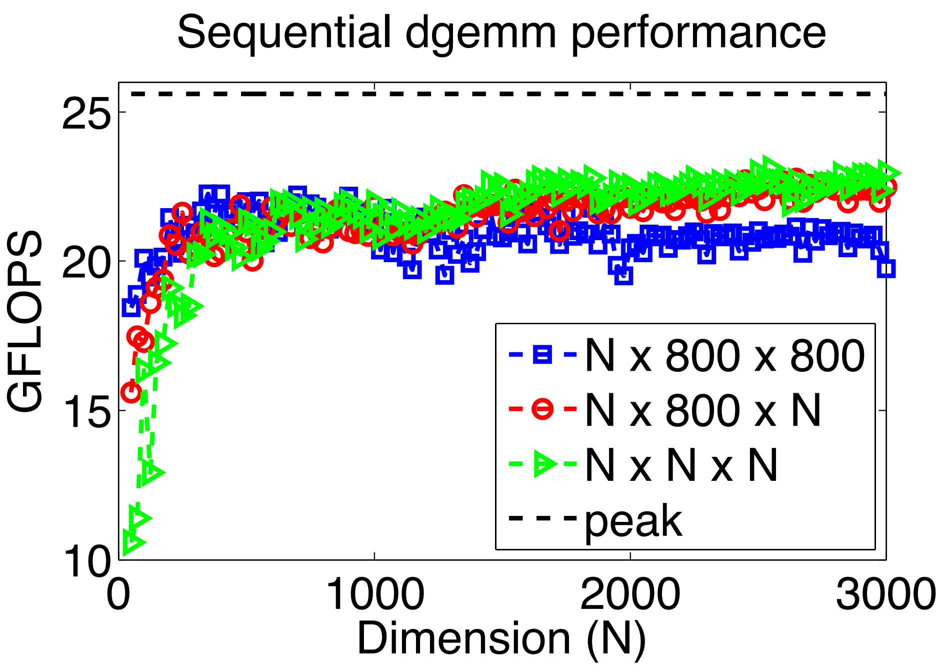
- Our approach: test all algorithms on a bunch of different problem sizes and look for patterns

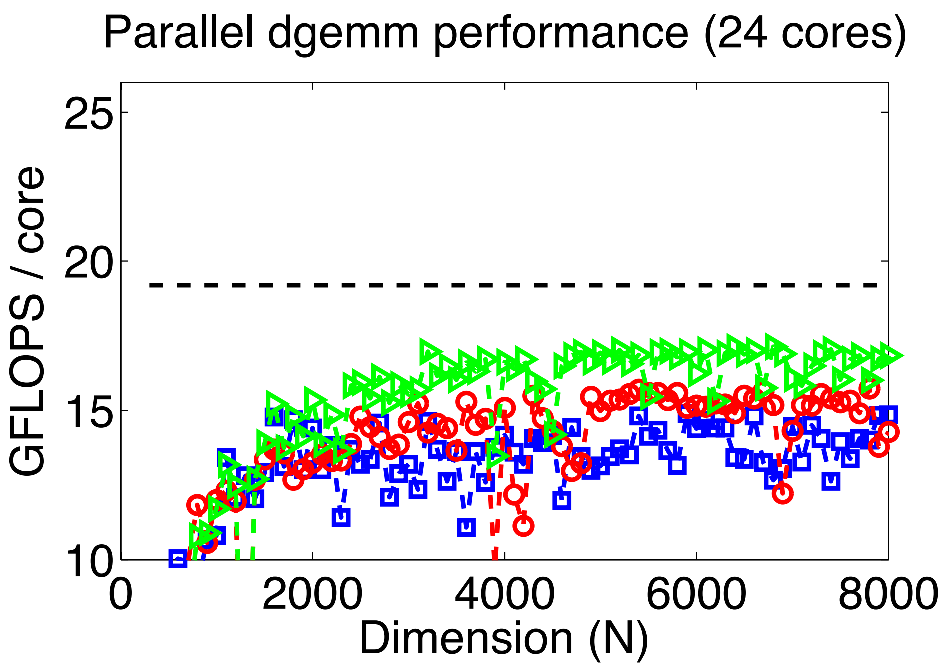
Practical issues

- Best way to do matrix additions? (in paper)
- Can we eliminate redundant linear combinations? (in paper)
- Different problem shapes other than square (this talk)
- When to stop recursion? (this talk)
- How to parallelize? (this talk)

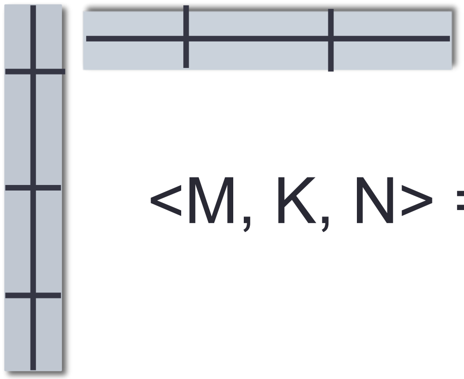


Recursion cutoff: look at gemm curve





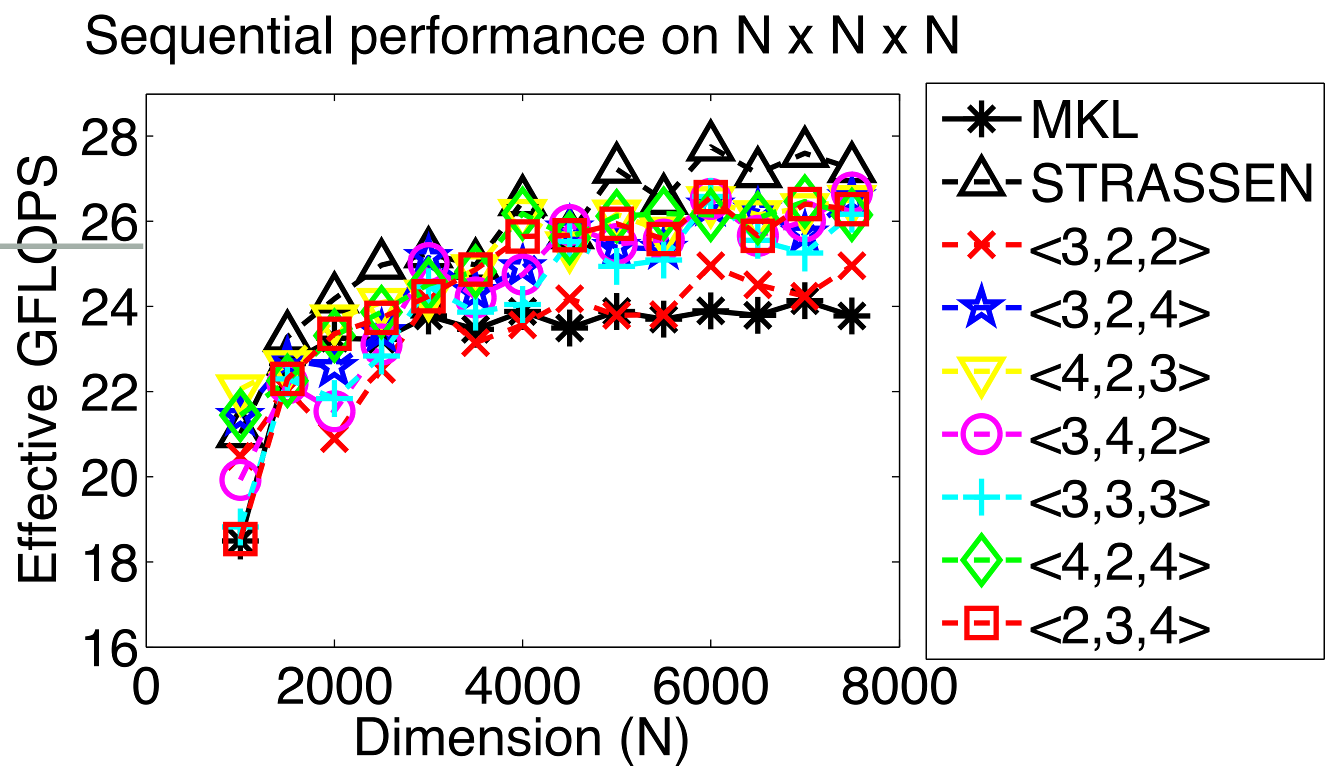
Basic idea: take another recursive step if the sub-problems will still operate at high performance



<M, K, N> = <4, 2, 3>

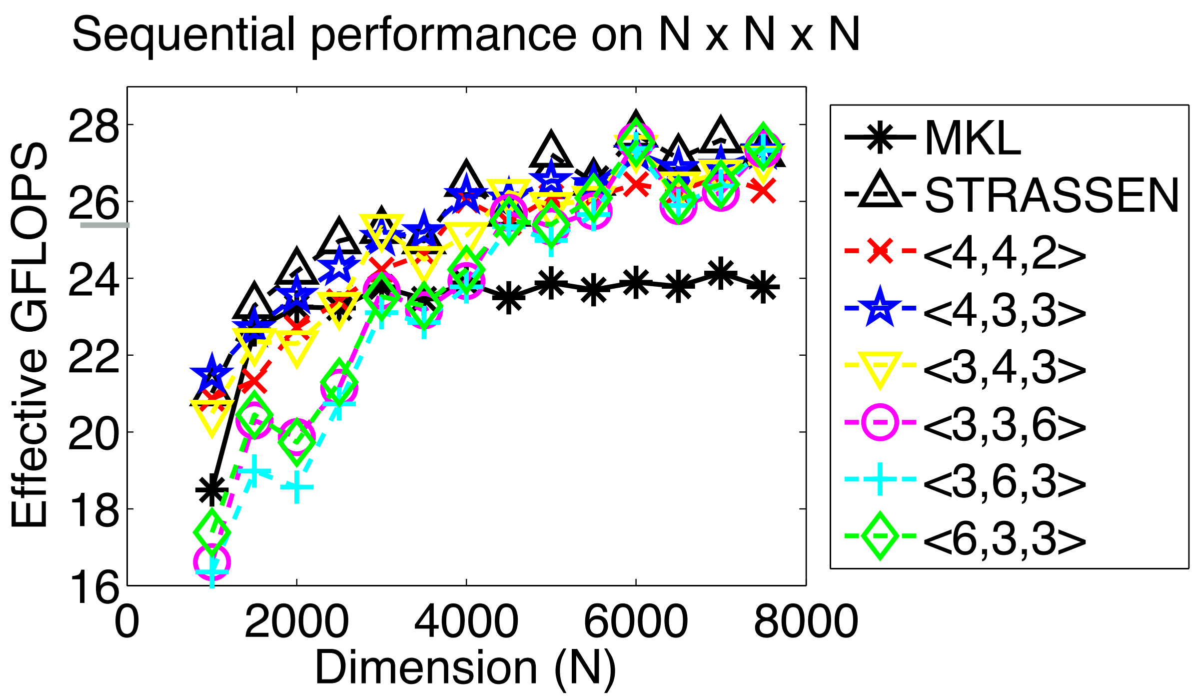
Sequential performance

True peak



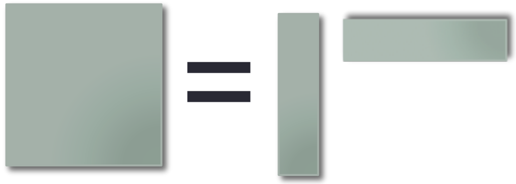
Effective GFLOPS for M x K x N multiplies
= 1e-9 * 2 * MKN / time in seconds

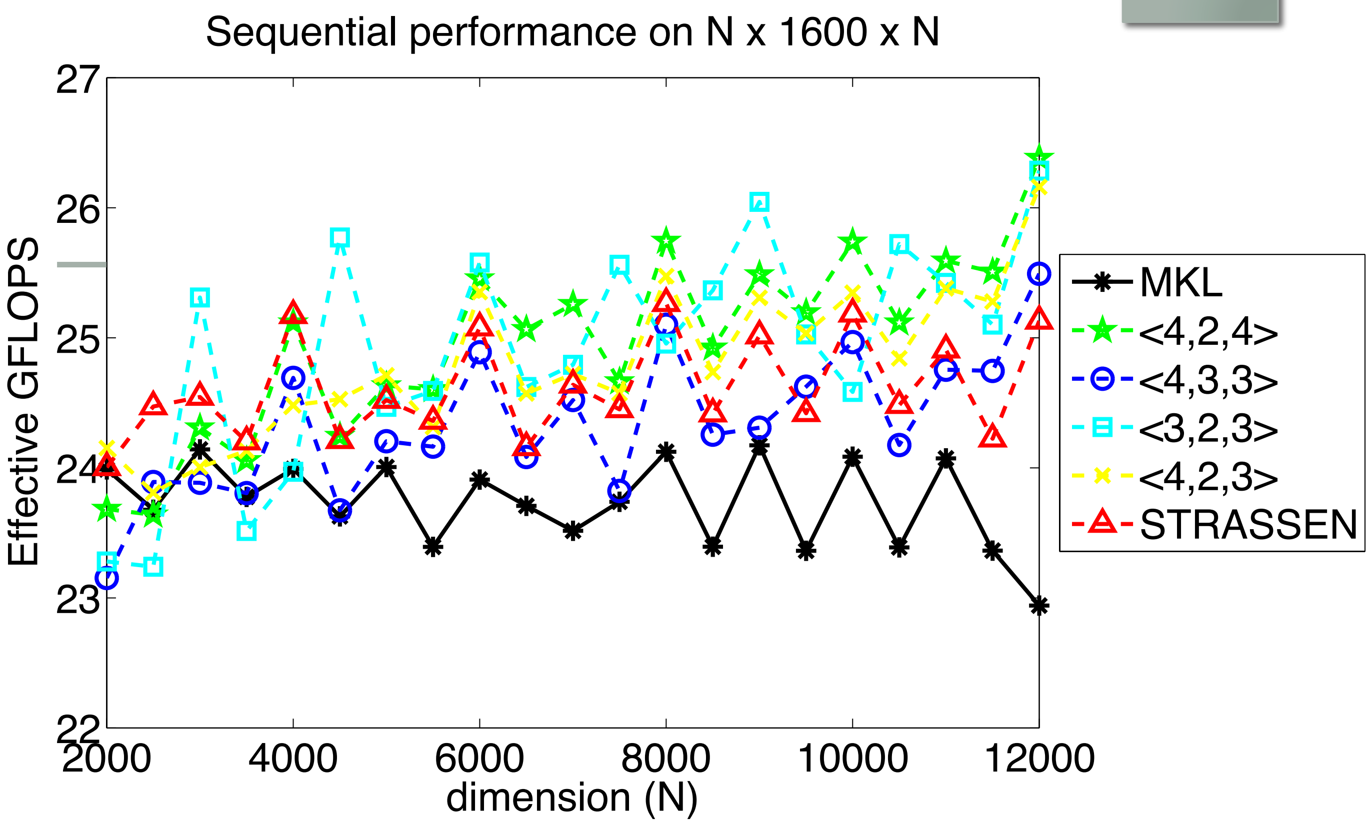
Sequential performance



- All algorithms beat MKL on large problems
- Strassen's algorithm is hard to beat

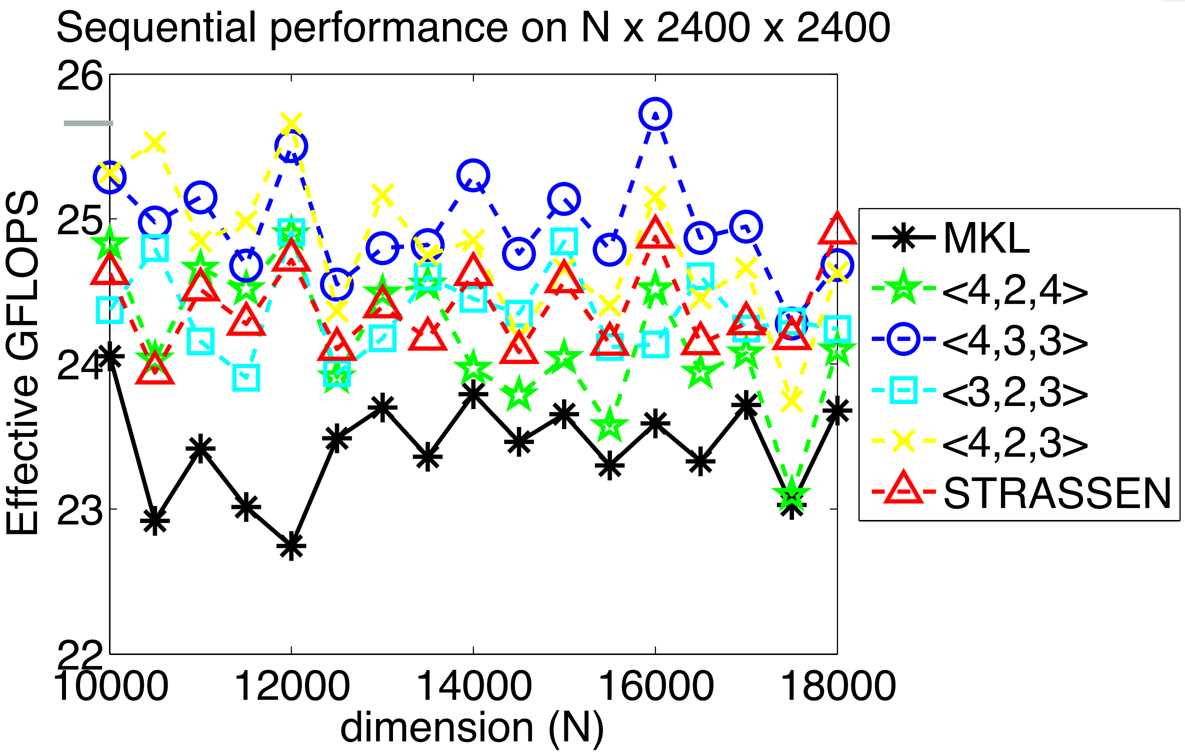
Sequential performance





- Almost all algorithms beat MKL
- <4, 2, 4> and <3, 2, 3> tend to perform the best

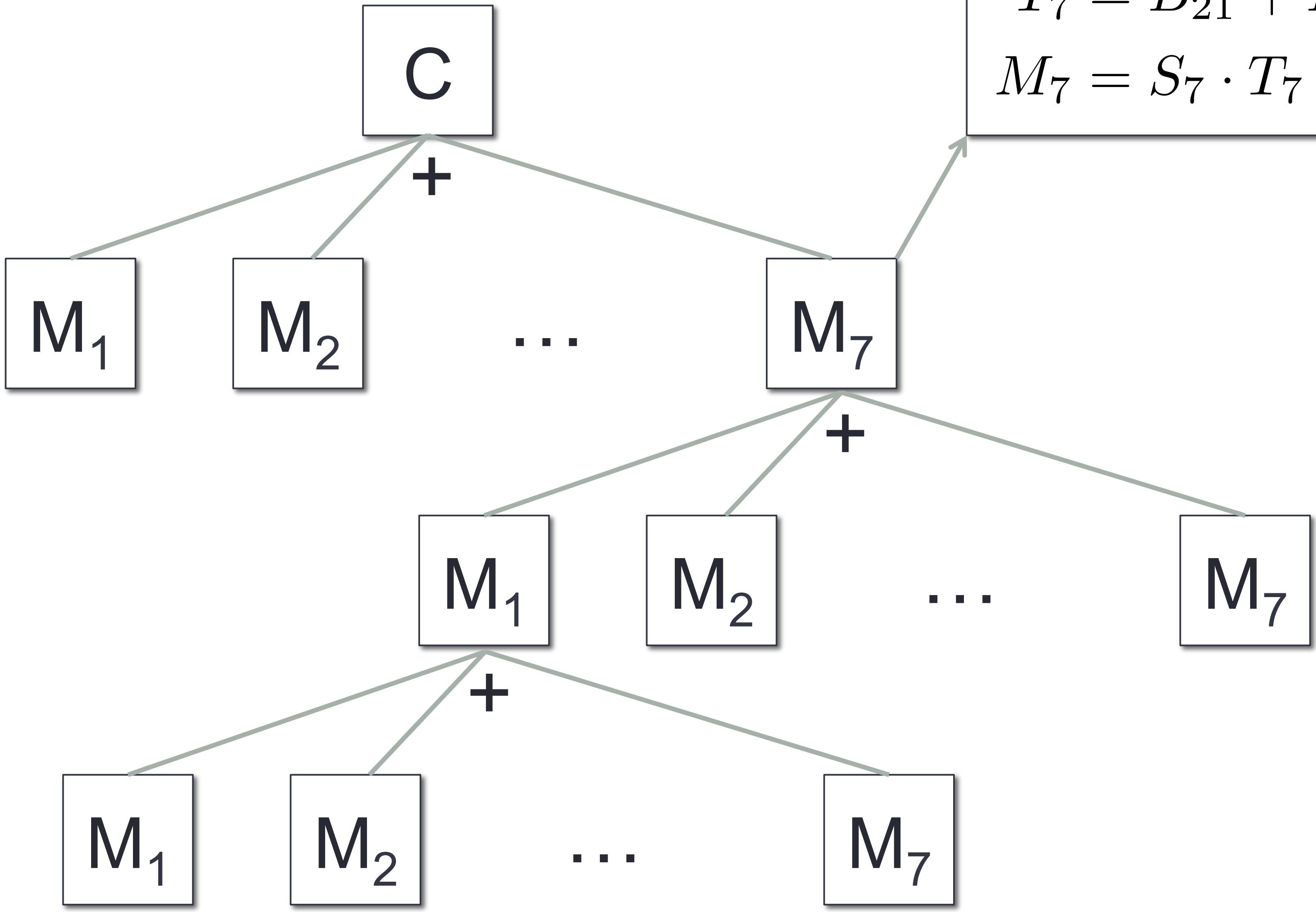
Sequential performance



- Almost all algorithms beat MKL
- <4, 3, 3> and <4, 2, 3> tend to perform the best

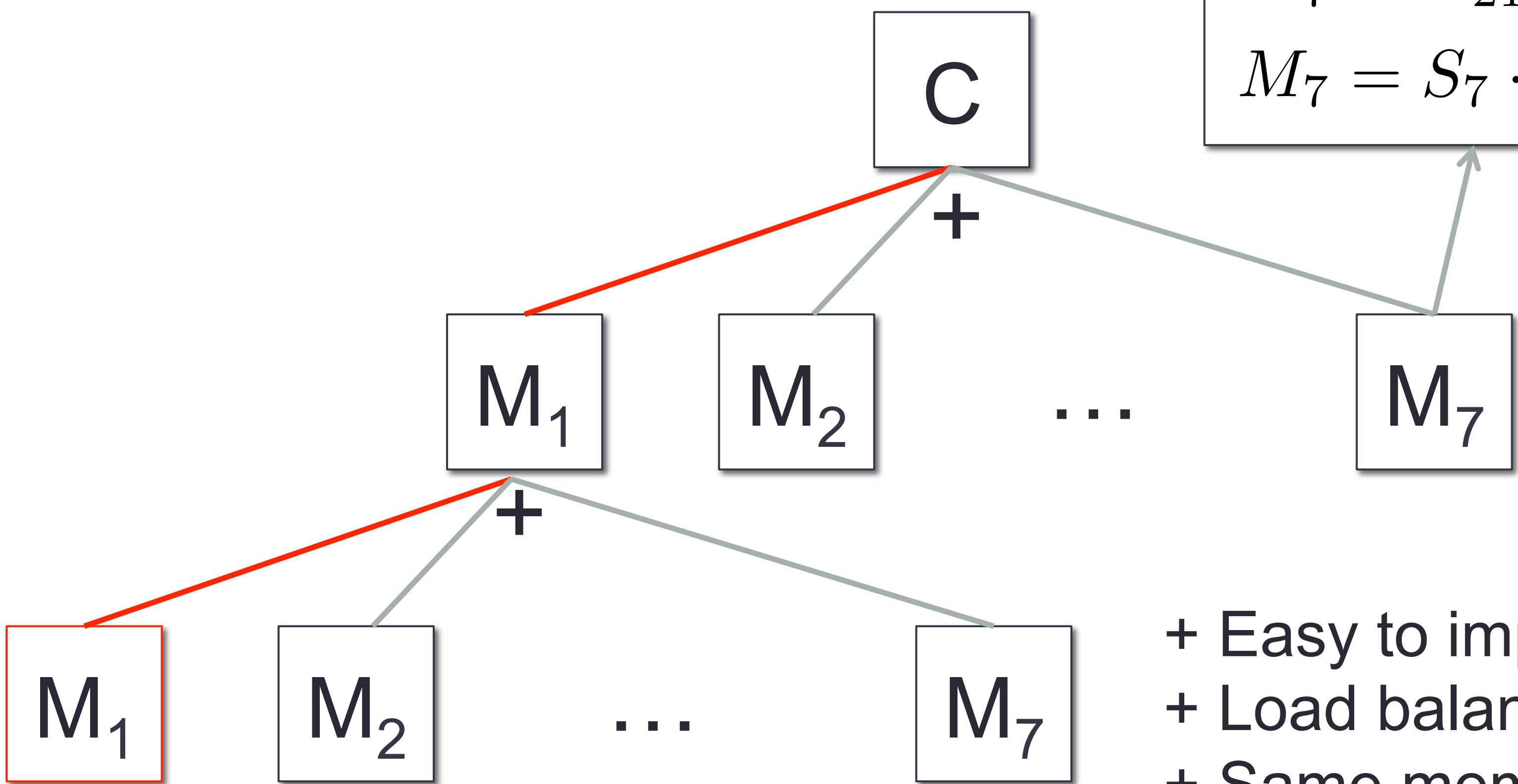
Parallelization

$$S_7 = A_{12} - A_{22}$$
$$T_7 = B_{21} + B_{22}$$
$$M_7 = S_7 \cdot T_7$$



DFS Parallelization

$$S_7 = A_{12} - A_{22}$$
$$T_7 = B_{21} + B_{22}$$
$$M_7 = S_7 \cdot T_7$$

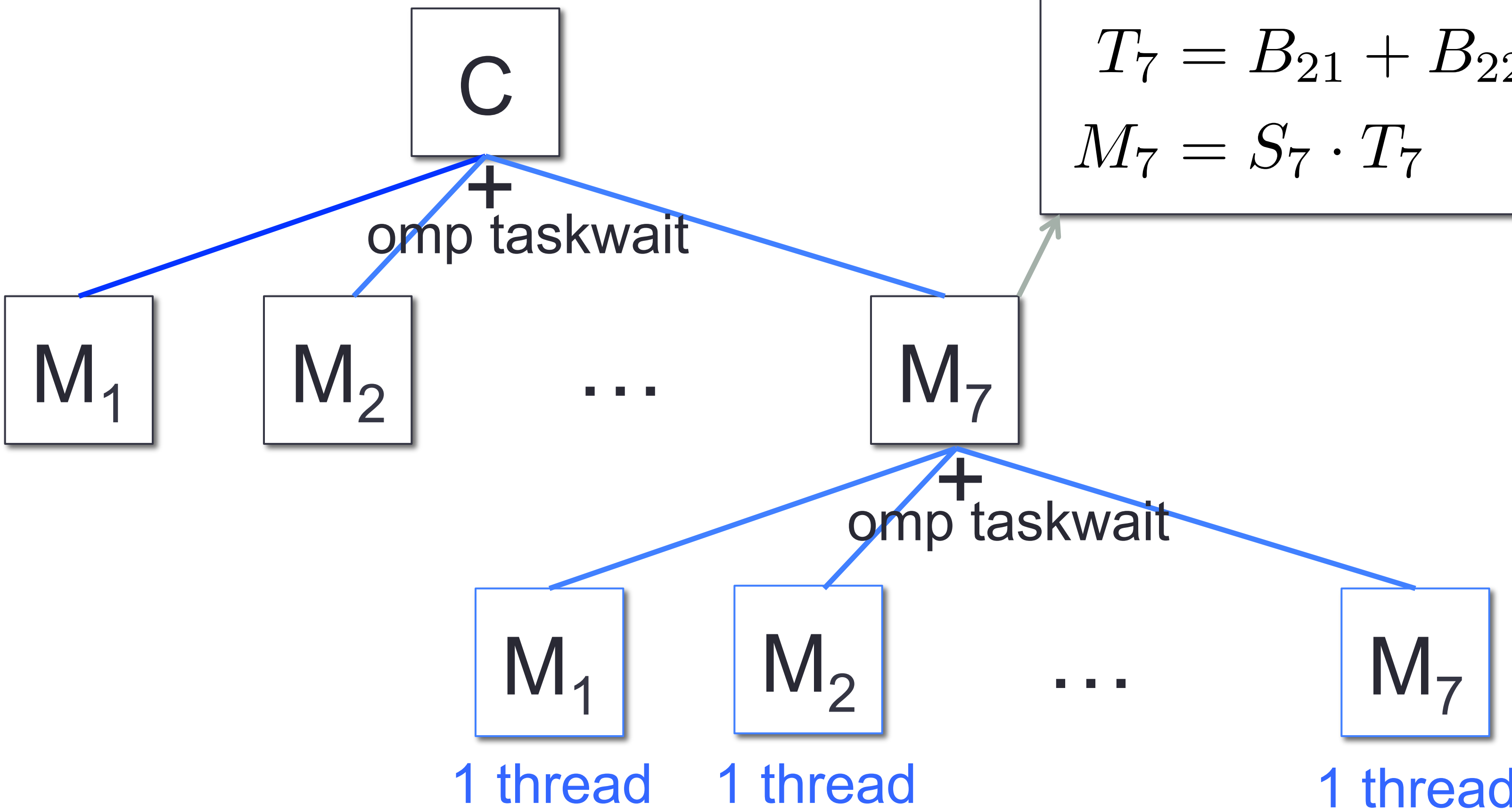


All threads

Use parallel MKL

+ Easy to implement
+ Load balanced
+ Same memory footprint as sequential
- Need large base cases for high performance

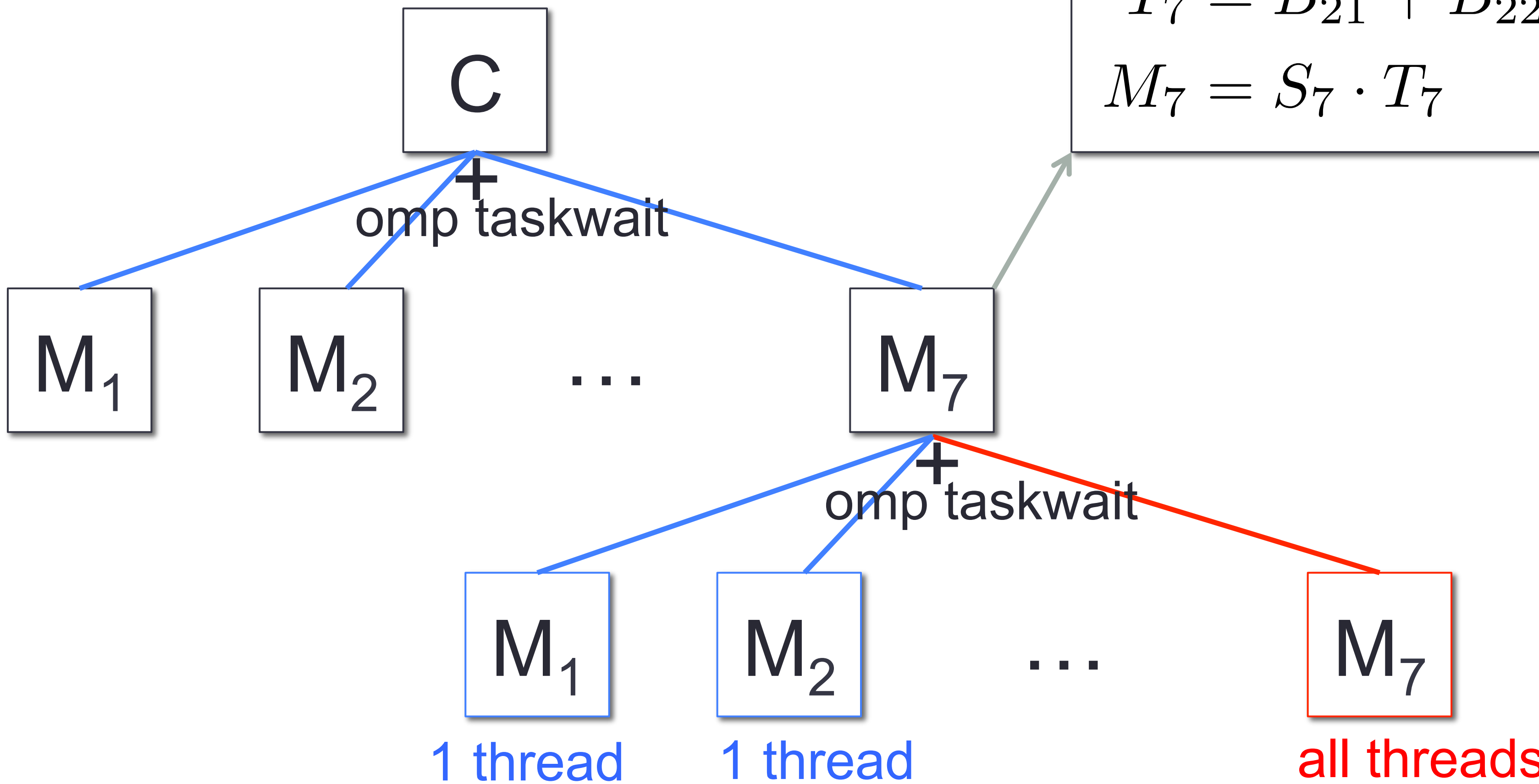
BFS Parallelization



$$S_7 = A_{12} - A_{22}$$
$$T_7 = B_{21} + B_{22}$$
$$M_7 = S_7 \cdot T_7$$

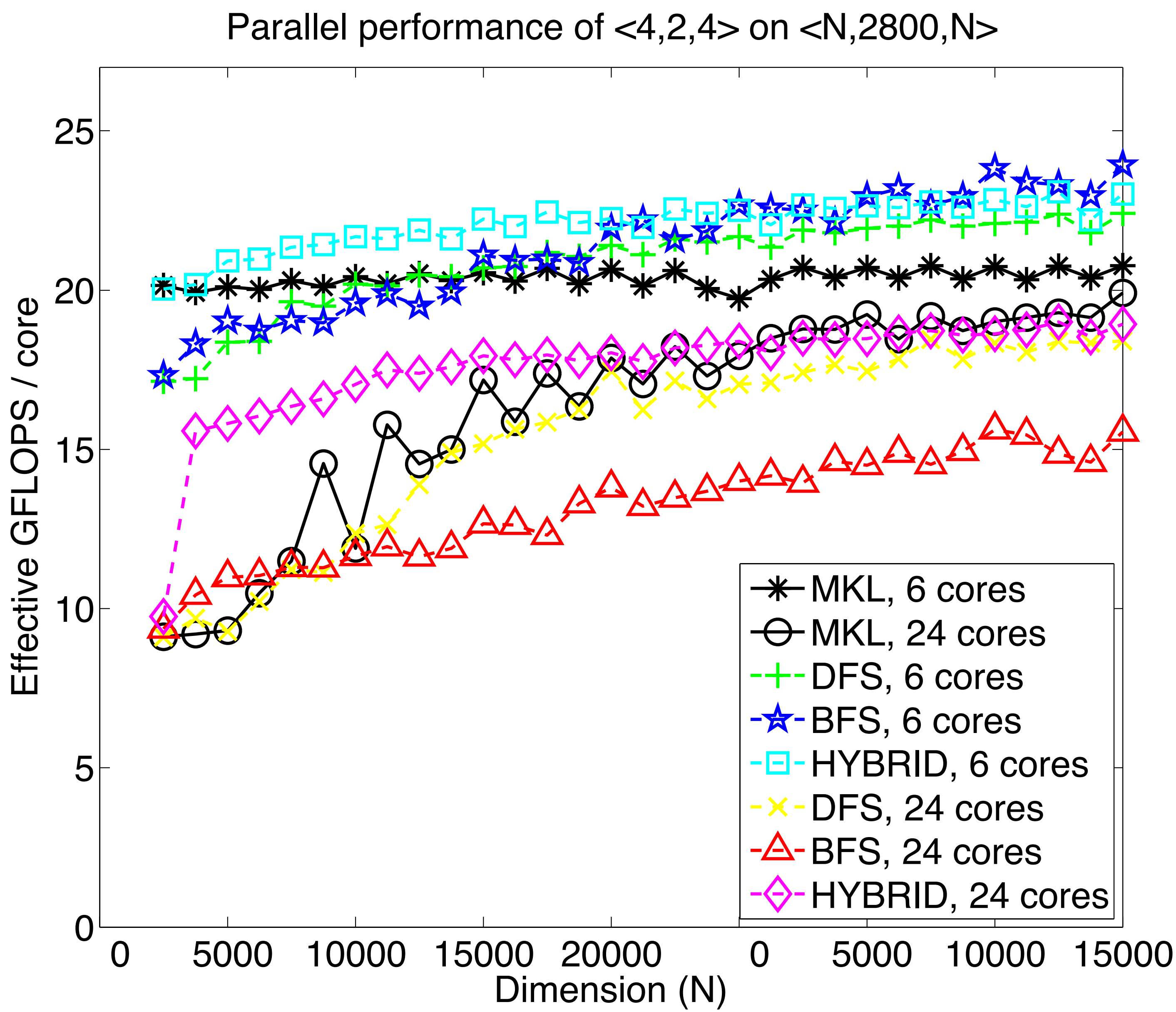
- \+ High performance for smaller base cases
- \- Sometimes harder to load balance: 24 threads, 49 subproblems
- \- More memory

HYBRID parallelization



$$S_7 = A_{12} - A_{22}$$
$$T_7 = B_{21} + B_{22}$$
$$M_7 = S_7 \cdot T_7$$

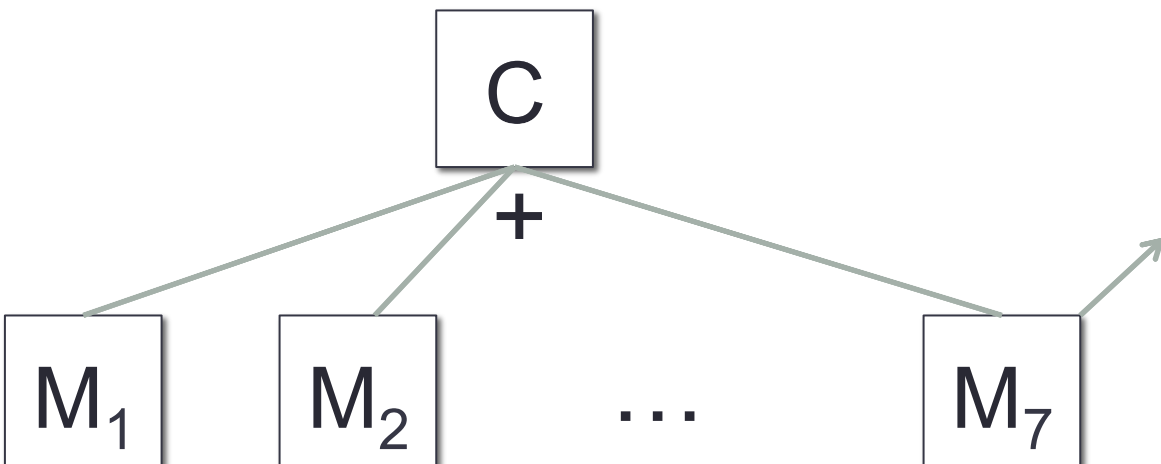
+ Better load balancing
- Explicit synchronization or else we can over-subscribe threads



Parallel performance of <4,2,4> on <N,2800,N>
Effective GFLOPS / core
Dimension (N)
0
5
10
15
20
25
0
5000
10000
15000
20000
0
5000
10000
15000
MKL, 6 cores
MKL, 24 cores
DFS, 6 cores
BFS, 6 cores
HYBRID, 6 cores
DFS, 24 cores
BFS, 24 cores
HYBRID, 24 cores

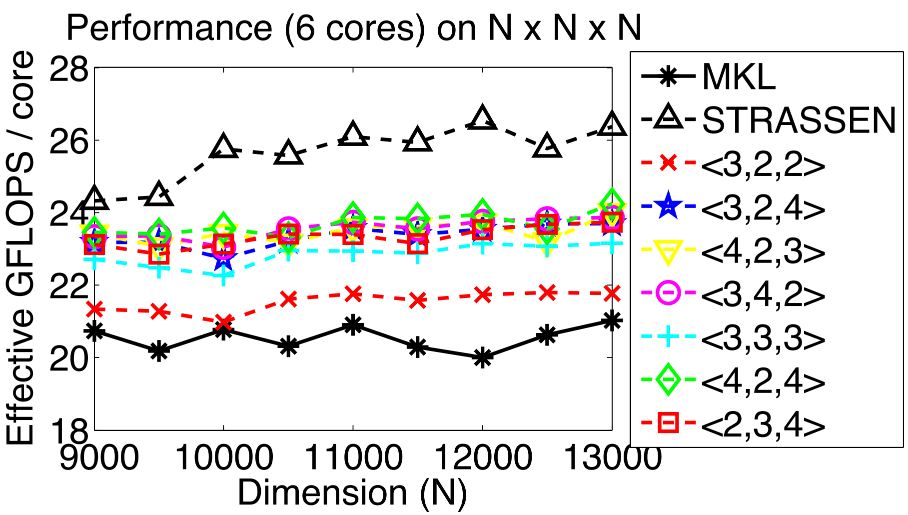
Bandwidth problems

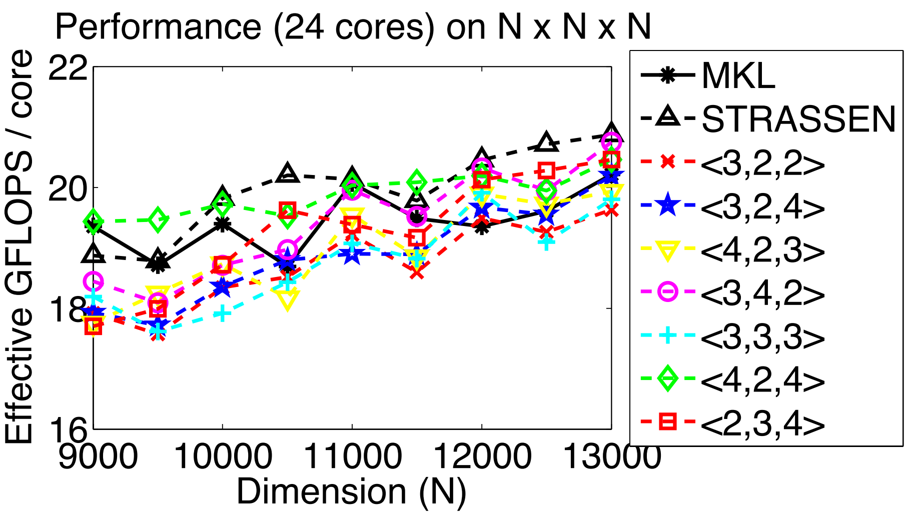
- We rely on the cost of matrix multiplications to be much more expensive than the cost of matrix additions
- Parallel dgemm on 24 cores: easily get 50-75% of peak
- STREAM benchmark: < 6x speedup in read/write performance on 24 cores



$$
\begin{aligned}
S_7 &= A_{12} - A_{22} \\
T_7 &= B_{21} + B_{22} \\
M_7 &= S_7 \cdot T_7
\end{aligned}
$$

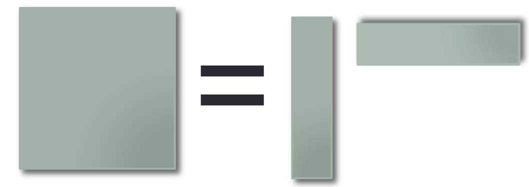
Parallel performance

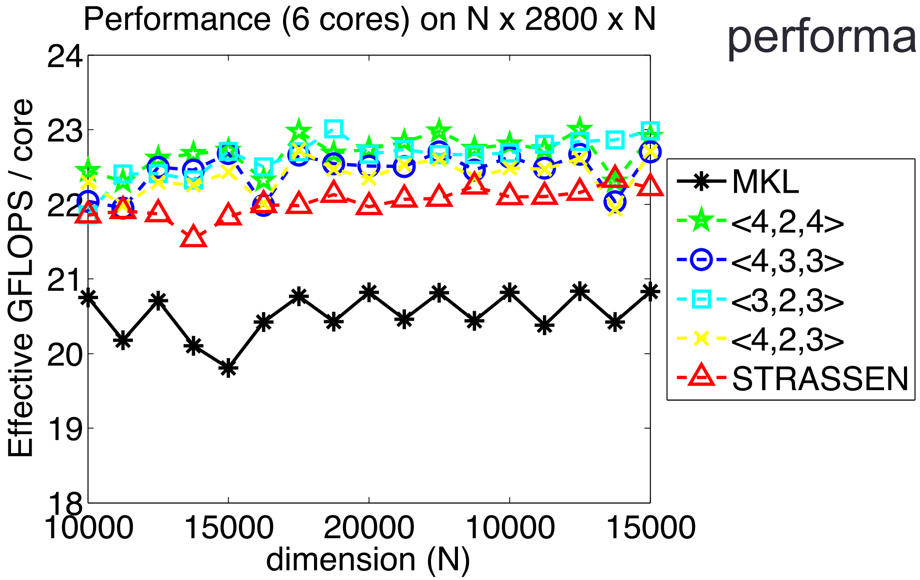




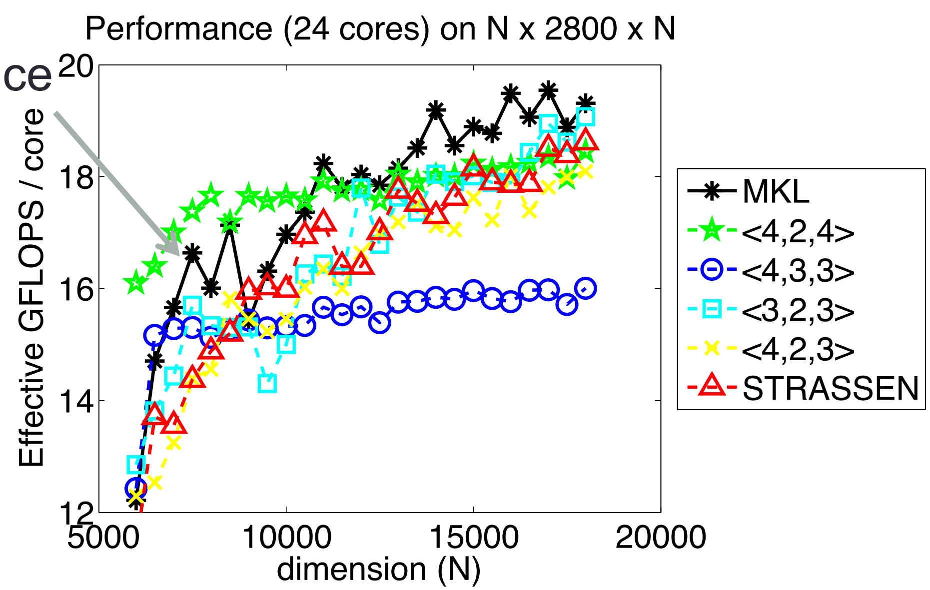
- 6 cores: similar performance to sequential
- 24 cores: can sometimes beat MKL, but barely

Parallel performance



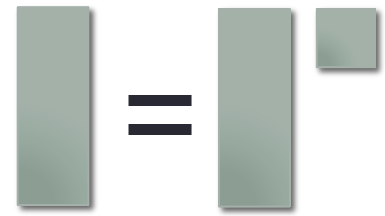


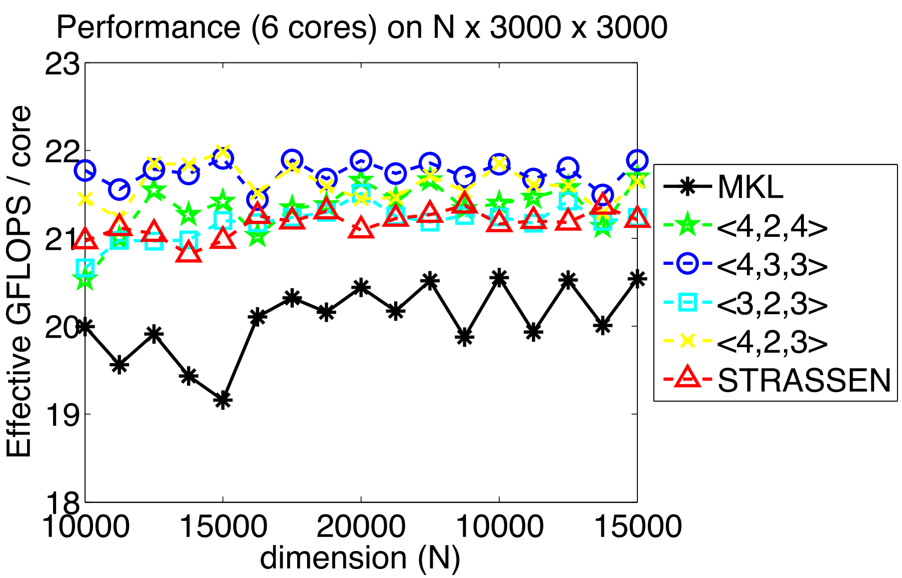
Bad MKL performance

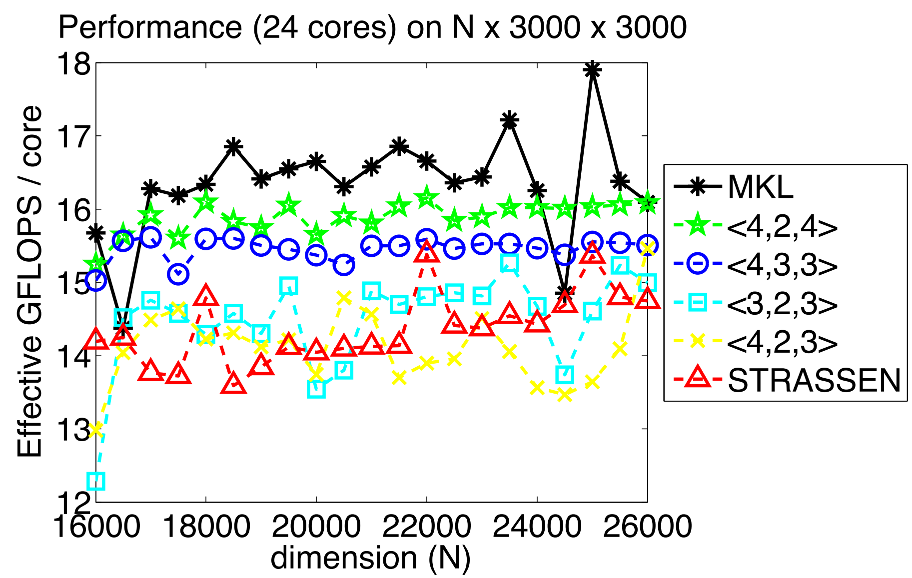


- 6 cores: similar performance to sequential
- 24 cores: MKL best for large problems

Parallel performance







- 6 cores: similar performance to sequential
- 24 cores: MKL usually the best

High-level conclusions

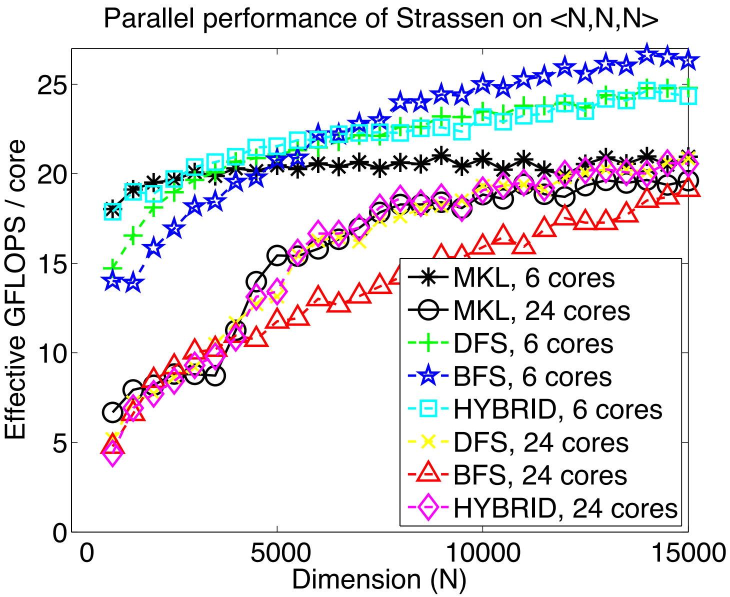
- For square matrix multiplication, Strassen's algorithm is hard to beat
- For rectangular matrix multiplication, use a fast algorithm that "matches the shape"
- Bandwidth limits the performance of shared memory parallel fast matrix multiplication
  → should be less of an issue in distributed memory

Future work:

- Numerical stability
- Using fast matmul as a kernel for other algorithms in numerical linear algebra

A FRAMEWORK FOR PRACTICAL FAST MATRIX MULTIPLICATION

arXiv: 1409.2908

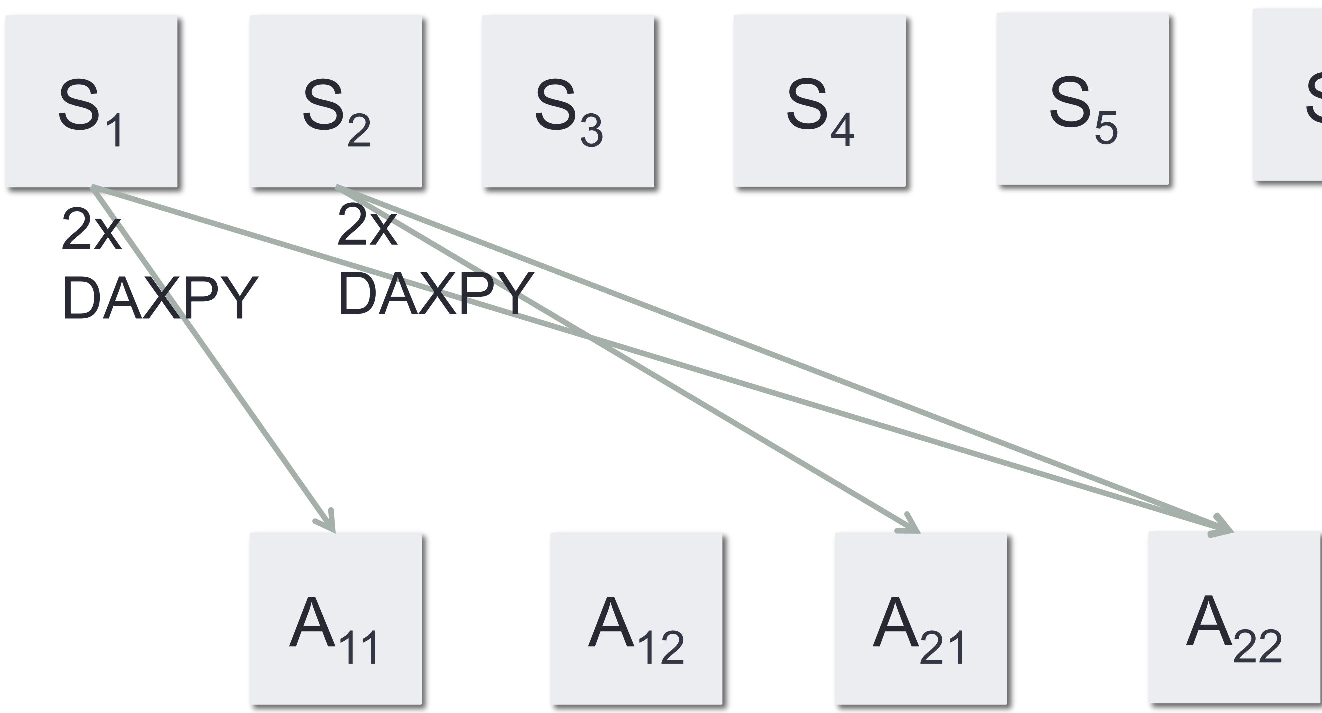


Austin Benson (arbenson@stanford.edu), ICME, Stanford

Grey Ballard, Sandia National Laboratories

BLIS Retreat, September 26, 2014

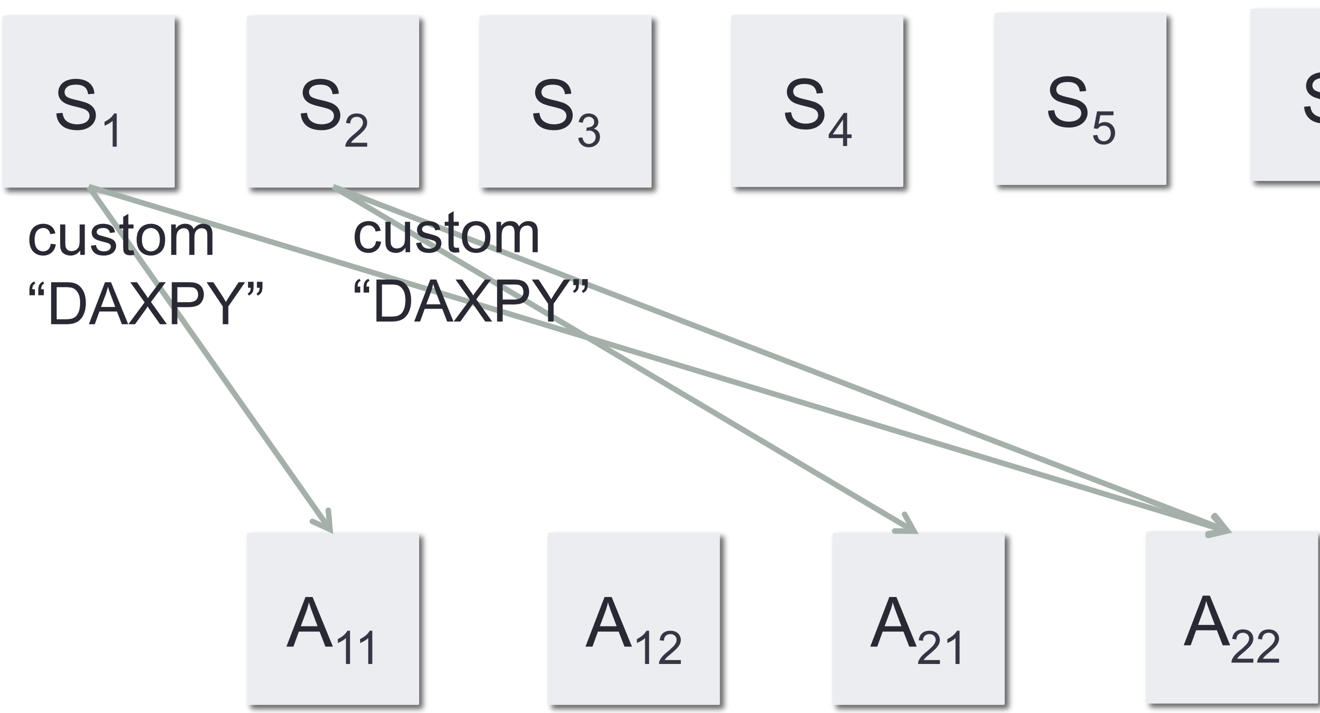
Matrix additions (linear combinations)



$$
\begin{aligned}
S_1 &= A_{11} + A_{22} \\
S_2 &= A_{21} + A_{22} \\
S_3 &= A_{11} \\
S_4 &= A_{22} \\
S_5 &= A_{11} + A_{12} \\
S_6 &= A_{21} - A_{11} \\
S_7 &= A_{12} - A_{22}
\end{aligned}
$$

“Pairwise”

Matrix additions (linear combinations)

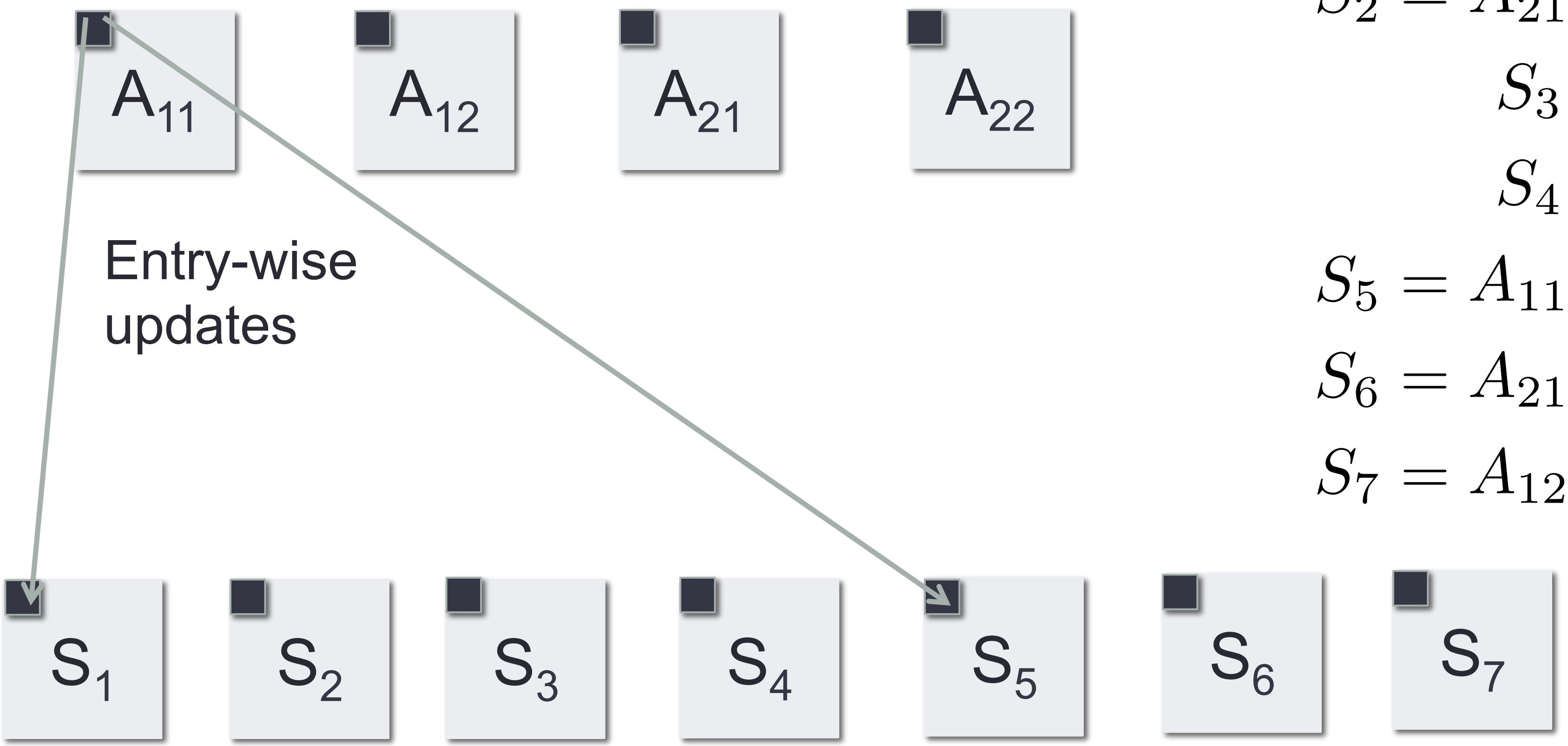


"Write once"

$$S_1 = A_{11} + A_{22}$$
$$S_2 = A_{21} + A_{22}$$
$$S_3 = A_{11}$$
$$S_4 = A_{22}$$
$$S_5 = A_{11} + A_{12}$$
$$S_6 = A_{21} - A_{11}$$
$$S_7 = A_{12} - A_{22}$$

Matrix additions (linear combinations)

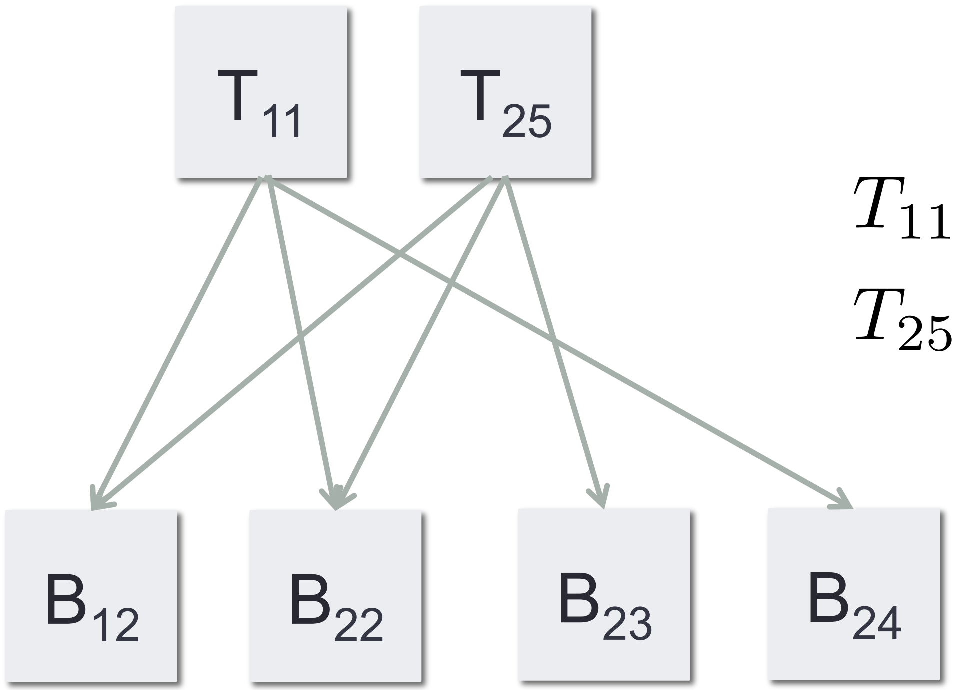
$$
\begin{aligned}
S_1 &= A_{11} + A_{22} \\
S_2 &= A_{21} + A_{22} \\
S_3 &= A_{11} \\
S_4 &= A_{22} \\
S_5 &= A_{11} + A_{12} \\
S_6 &= A_{21} - A_{11} \\
S_7 &= A_{12} - A_{22}
\end{aligned}
$$



“Streaming”

Common subexpression elimination (CSE)

- Example in <4, 2, 4> algorithm (R = 26 multiples):



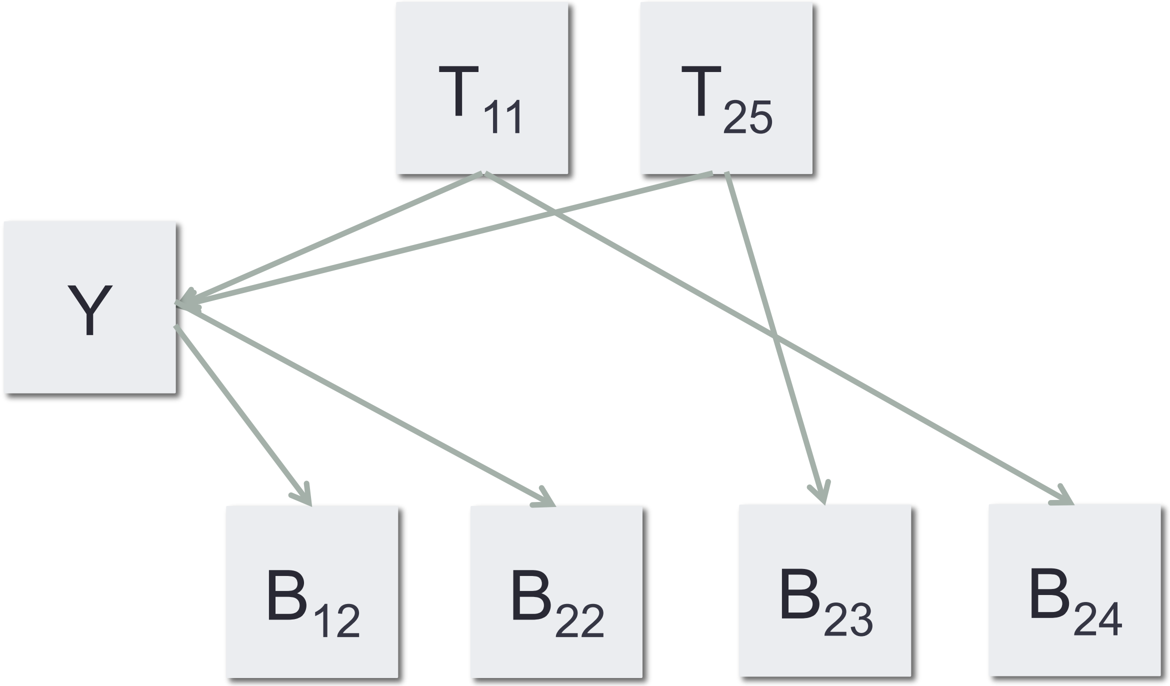
$$T_{11} = B_{24} - (B_{12} + B_{22})$$

$$T_{25} = B_{23} + B_{12} + B_{22}$$

Four additions, six reads, two writes

Common subexpression elimination (CSE)

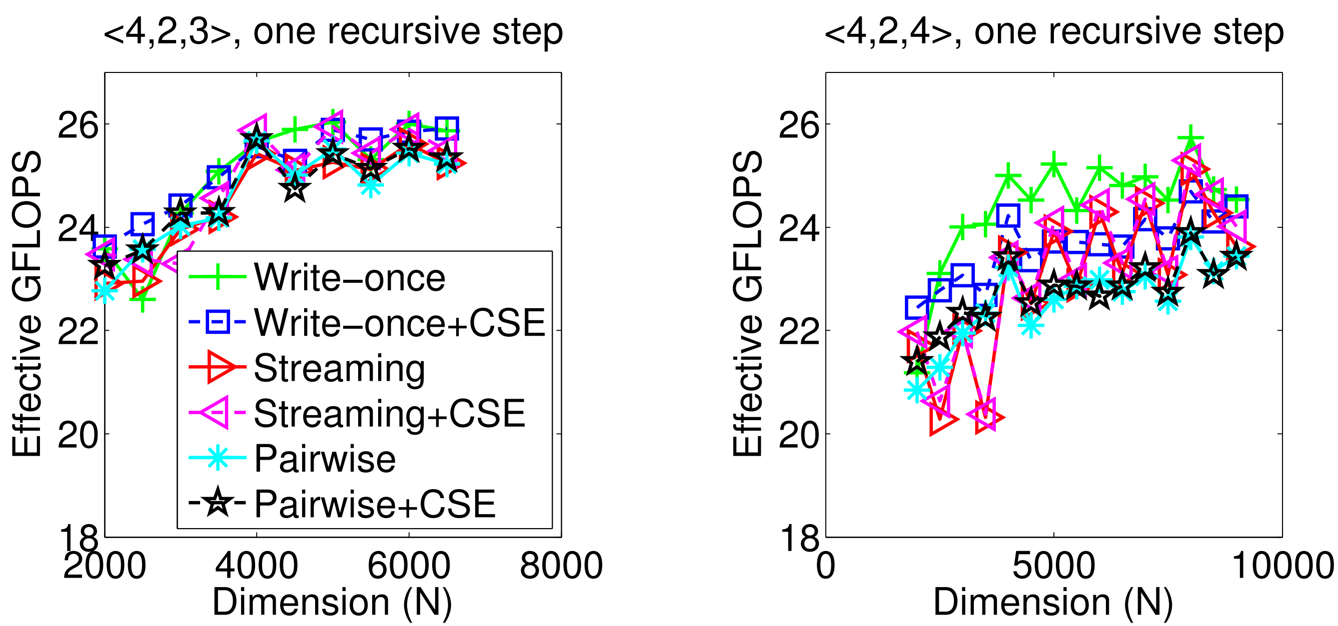
- Example in <4, 2, 4> algorithm (R = 26 multiples):



$$Y = B_{12} + B_{22}$$
$$T_{11} = B_{24} - Y$$
$$T_{25} = B_{23} + Y$$

Three additions, six reads, three writes
→ Net increase in communication!

CSE does not really help



Effective GFLOPS for M x K x N multiplies

= 1e-9 * 2 * MKN / time in seconds