COMPUTATIONAL LINEAR ALGEBRA

- **Matrix –Vector Multiplication**
- **Matrix – matrix Multiplication**
- **Slides from UCSD and USB**
- **Directed Acyclic Graph Approach Jack Dongarra**
- **A new approach using Strassen`s algorithm Jim Demmel**

How do we optimize performance ?

Using a Simpler Model of Memory to Optimize

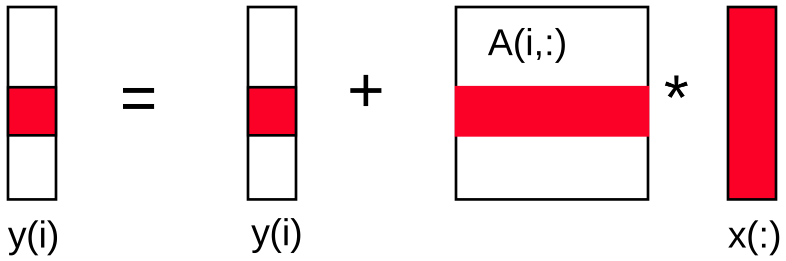
- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation << t_m
  - $q = f / m$ average number of flops per slow element access
- Min. possible time = $f * t_f$ when all data in fast memory
- Actual time

$$= f \cdot t_f + m \cdot t_m = f \cdot t_f \cdot \left(1 + \frac{t_m}{t_f} \frac{1}{q} \right)$$

- Larger q means Time closer to minimum $f * t_f$

Warm up: Matrix-vector multiplication

```
{implements y = y + A*x}
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



Warm up: Matrix-vector multiplication

```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
    {read row i of A into fast memory}
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

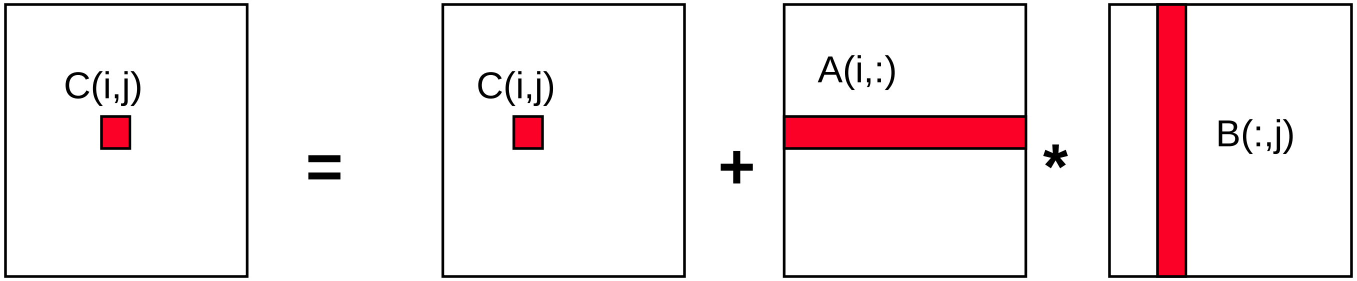
- m = number of slow memory refs = $3n + n^2$
- f = number of arithmetic operations = $2n^2$
- $q = f / m \sim= 2$

- Matrix-vector multiplication limited by slow memory speed

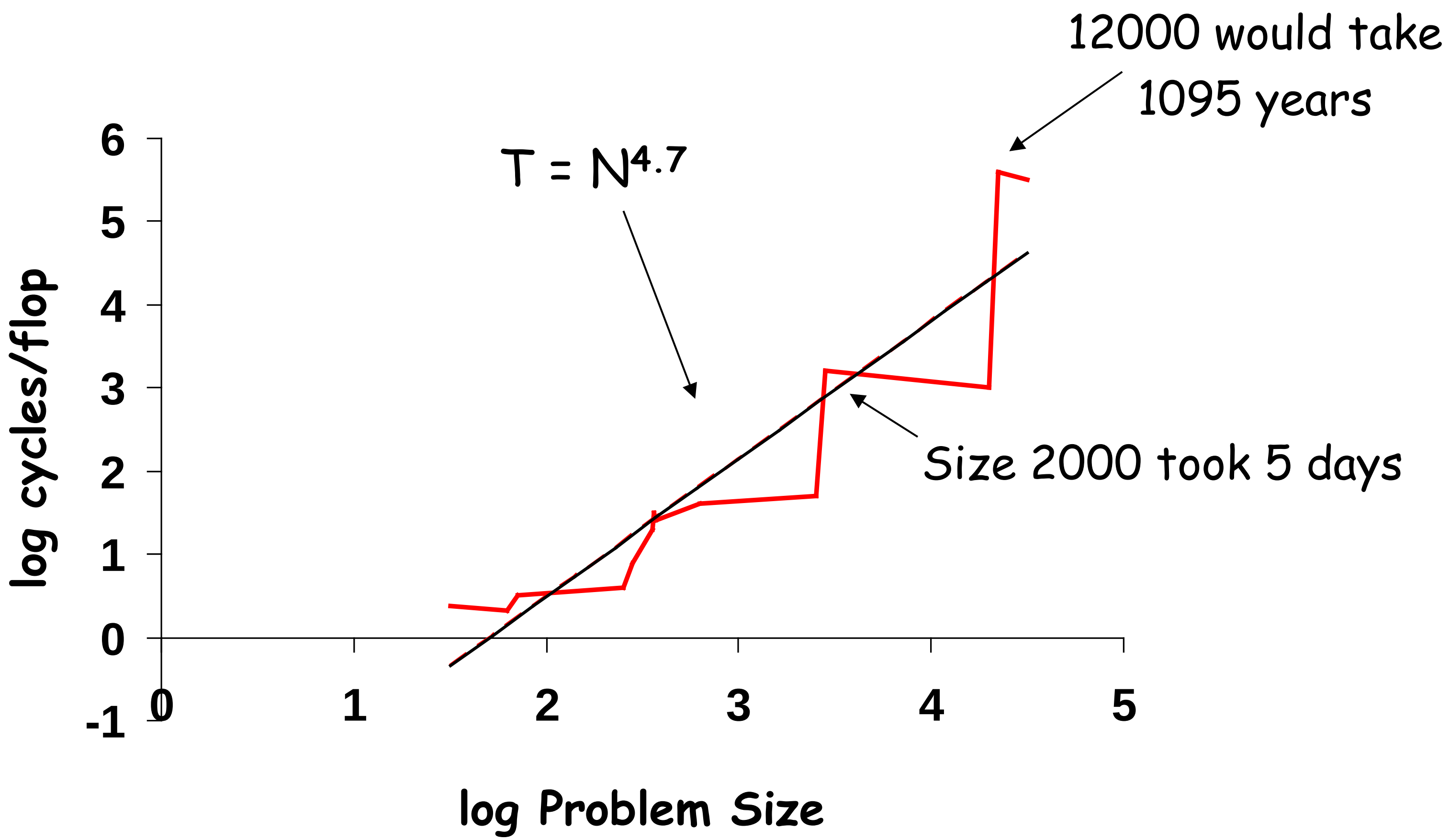
“Naïve” Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Algorithm has $2*n^3 = O(n^3)$ Flops and operates on $3*n^2$ words of memory



Matrix Multiply on RS/6000



$O(N^3)$ performance would have constant cycles/flop

Performance looks much closer to $O(N^5)$

“Naïve” Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Reuse value from a register

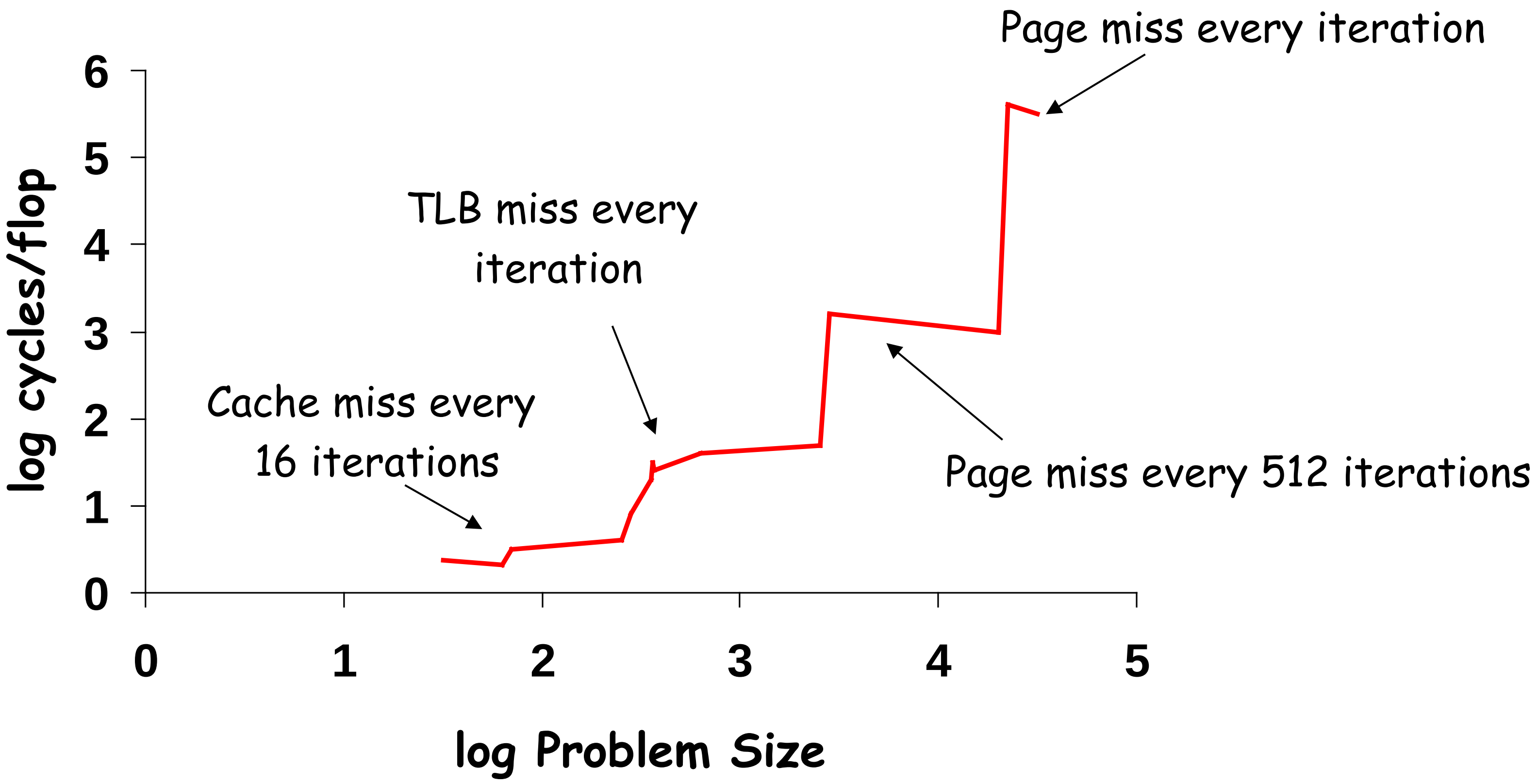
Stride-N access to one row*

Sequential access through entire matrix

- When cache (or TLB or memory) can’t hold entire B matrix, there will be a miss on every line.
- When cache (or TLB or memory) can’t hold a row of A, there will be a miss *on each access*

*Assumes column-major order

Matrix Multiply on RS/6000



Page miss every iteration

TLB miss every iteration

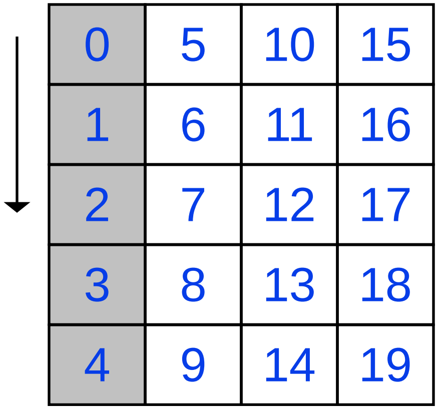
Cache miss every 16 iterations

Page miss every 512 iterations

Note on Matrix Storage

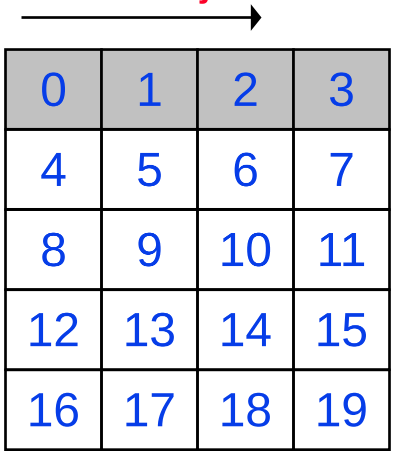
- A matrix is a 2-D array of elements, but memory addresses are “1-D”
- Conventions for matrix layout
  - by column, or “column major” (Fortran default)
  - by row, or “row major” (C default)

Column major



0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

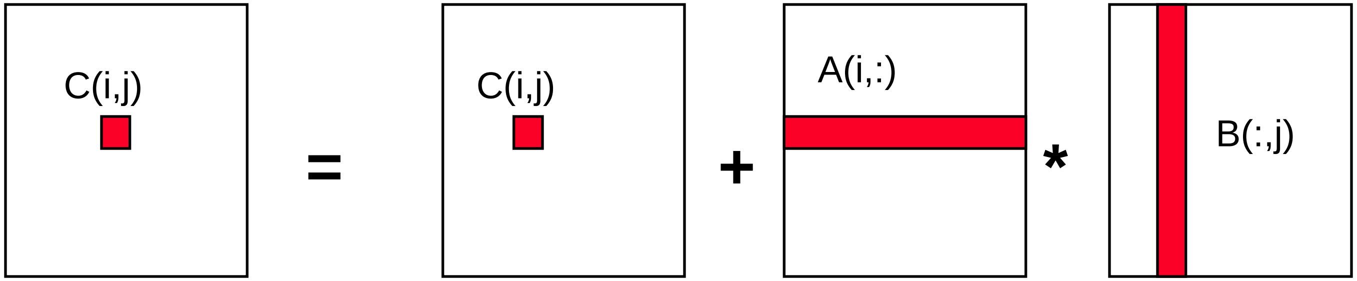
Row major



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Standard Approach to Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
       {read C(i,j) into fast memory}
       {read column j of B into fast memory}
       for k = 1 to n
           C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write C(i,j) back to slow memory}
```



Standard Approach to Matrix Multiply

Number of slow memory refs on unblocked matrix multiply

$m = n^3$: read each column of B n times

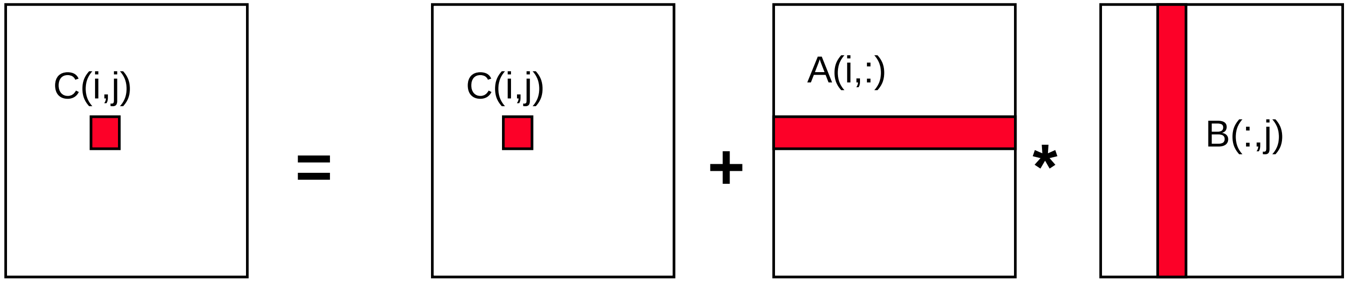
$+ n^2$: read each column of A once for each i

$+ 2n^2$: read and write each element of C once

$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (n^3 + 3n^2)$

$\sim= 2$ for large n



Alternative forms of Matrix Matrix Multiply

I-J-K nest:

```
do i=1,N
do j=1,N
    s=a(i,j)
    do k=1,N
           s=s+ b(i,k) *c(k,j)
     end
     a(i,j)=s
end
end
```

Large N: Estimate number of memory accesses
2*N2 + N* N2 + N2~ **N3**
High probability that b(i,1:N)remains in cache for each j loop + stride=N for c access

K-I-J nest:

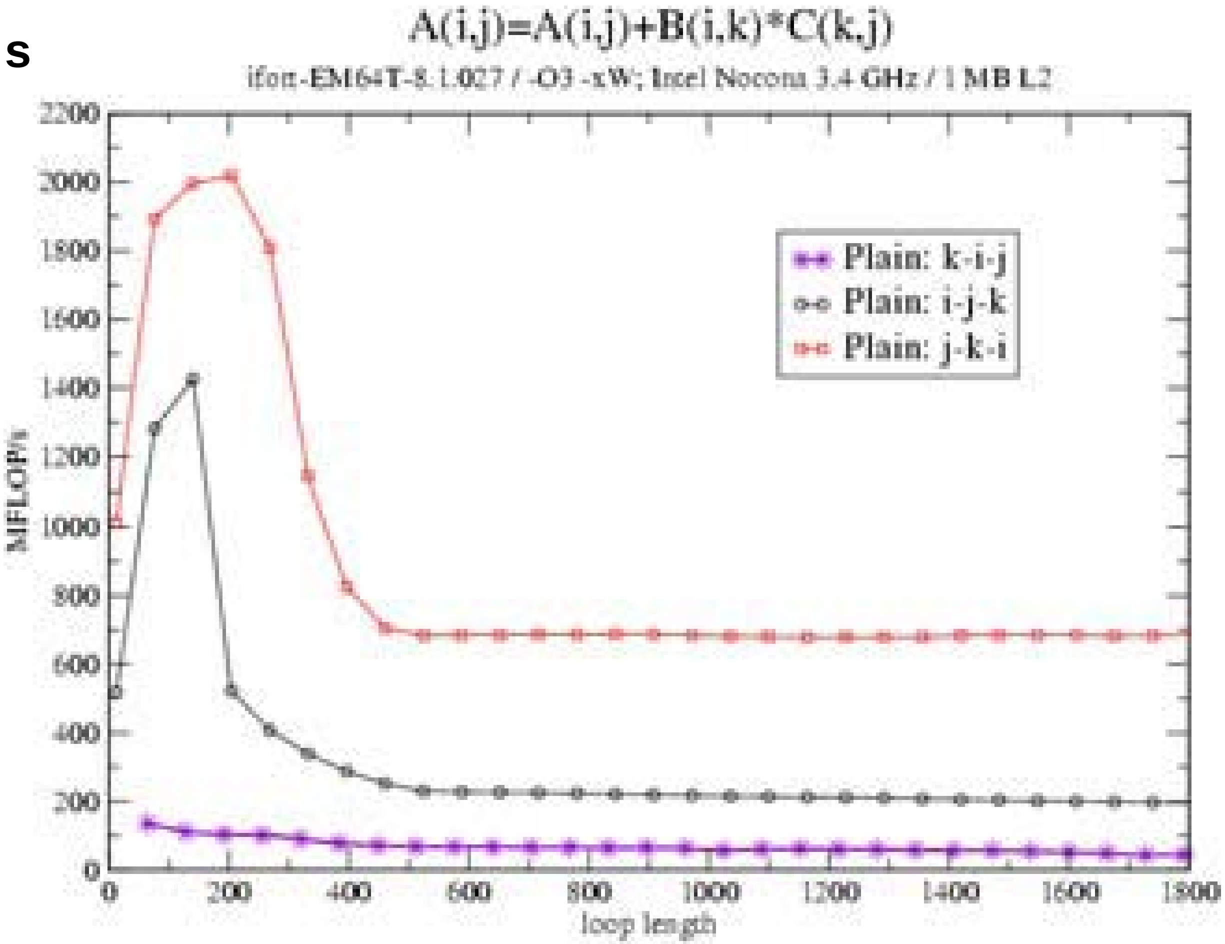
```
do k=1,N
do i=1,N
    s=b(i,k)
   do j=1,N
         a(i,j)= a(i,j) + s *c(k,j)
   end
end
end
```

Large N: Estimate number of memory accesses
2*N*N2 + N2 + N2~ **2*N3**
Matrix A must be loaded and stored N times + stride=N for a & c accesses!

- □ **J-K-I nest:**

```
° do j=1,N
° do k=1,N
°       s=c(j,k)
°       do i=1,N
°           a(i,j) = a(i,j) + b(i,k) * s
°       end
° End
° end
```

Large N: Estimate number of memory accesses
2*N2 + N* N2 + N2~ **N3**
B must be loaded N-times but
stride=1 access in inner loop!

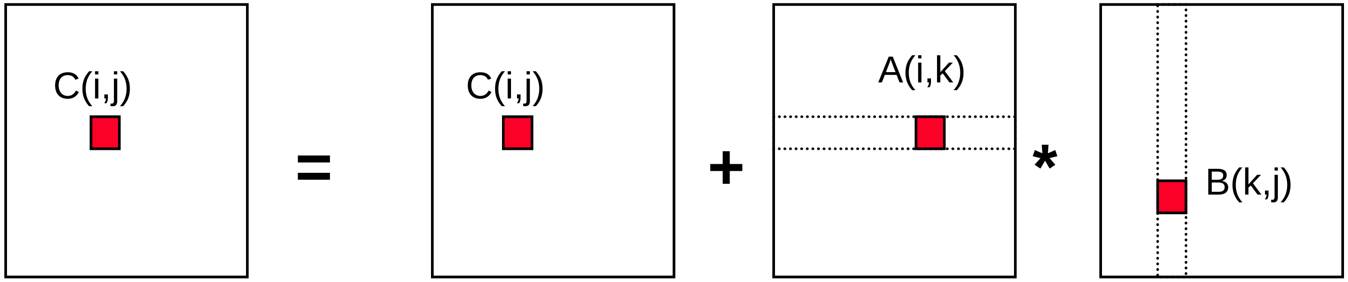


Block Structured Matrix Multiply

Let A,B,C be n by n matrices split into

N by N matrices of b by b subblocks where block size is b=n / N

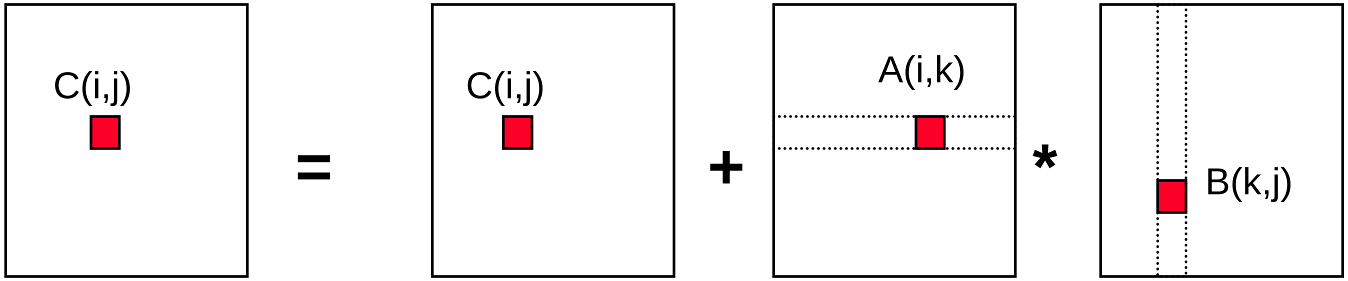
```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
             C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



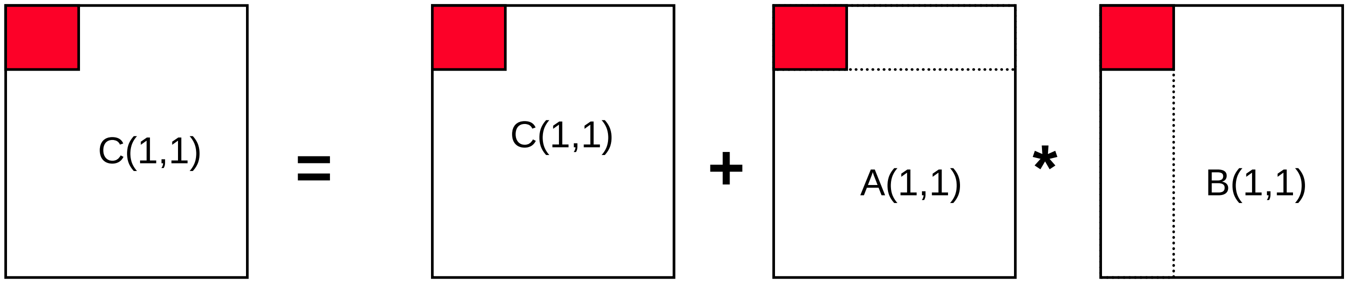
Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N by N matrices of b by b subblocks where b=n / N is called the block size

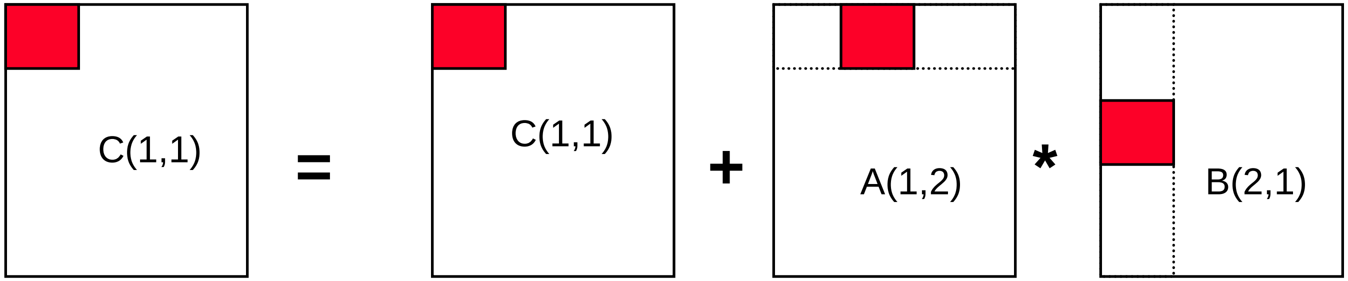
```
for i = 1 to N
  for j = 1 to N
     {read block C(i,j) into fast memory}
     for k = 1 to N
          {read block A(i,k) into fast memory}
          {read block B(k,j) into fast memory}
           C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
      {write block C(i,j) back to slow memory}
```



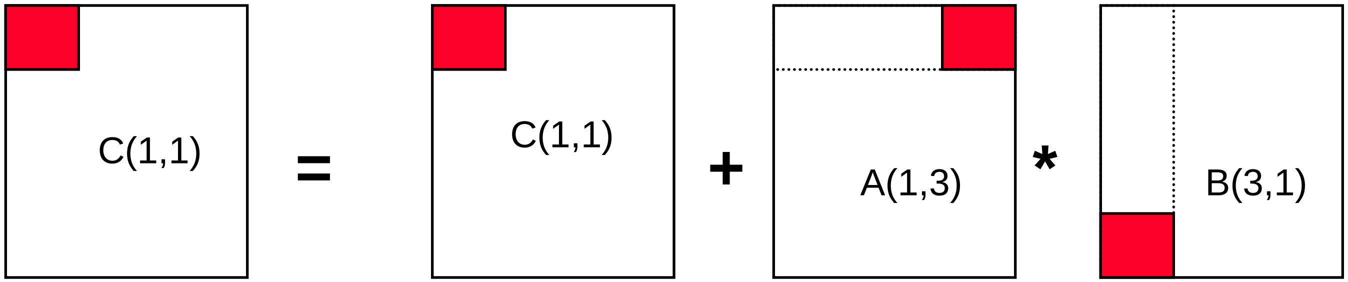
Blocked (Tiled) Matrix Multiply



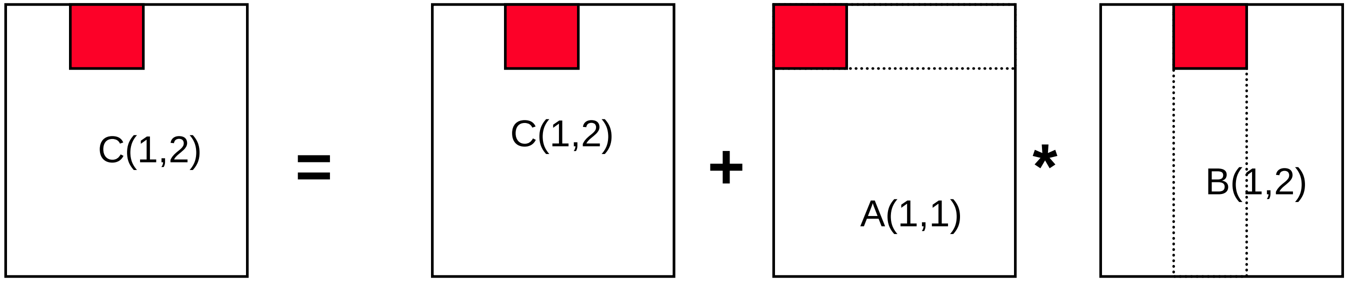
Blocked (Tiled) Matrix Multiply



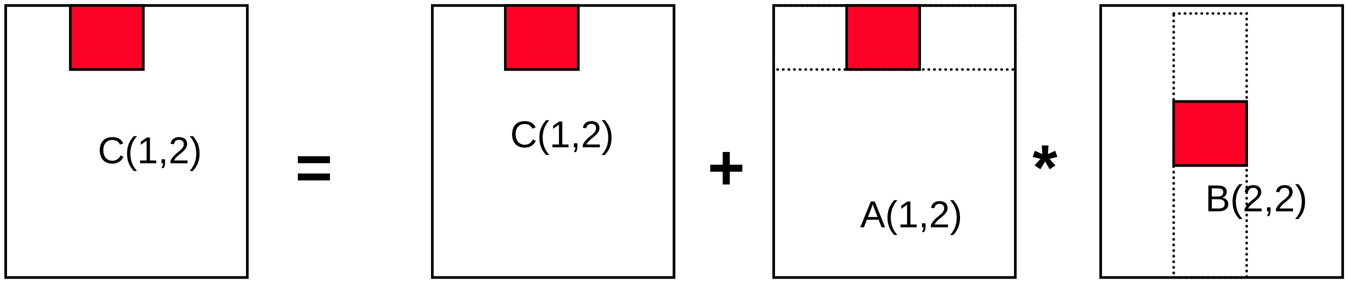
Blocked (Tiled) Matrix Multiply



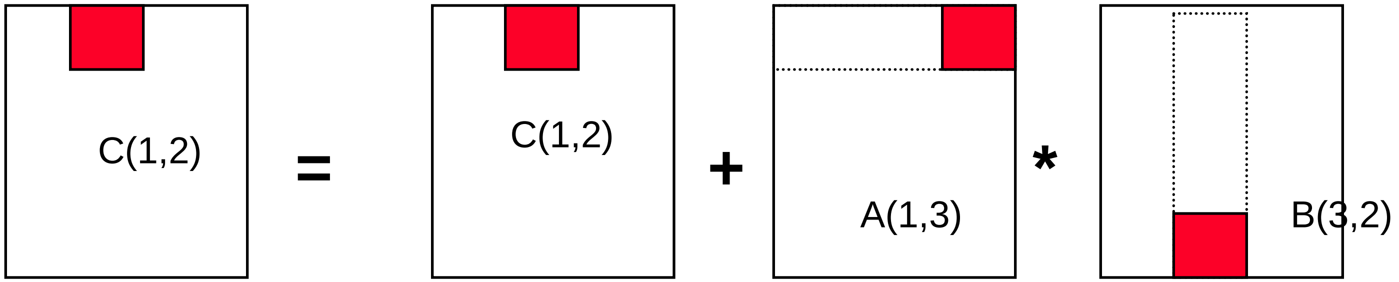
Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply

Recall:

m is amount memory traffic between slow and fast memory

matrix has nxn elements, and NxN blocks each of size bxb

f is number of floating point operations, $2n^3$ for this problem

q = f / m is our measure of algorithm efficiency in the memory system

So:

$m = N*n^2$ read each block of B N^3 times ($N^3 * n/N * n/N$)

$+ N*n^2$ read each block of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$

$\sim= n / N = b$ for large n

So we can improve performance by increasing the blocksize b

Can be much faster than matrix-vector multiply (q=2)

Using Analysis to Understand Machines

The blocked algorithm has computational intensity q ~= b

- The larger the block size, the more efficient our algorithm will be

- Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large

- Assume your fast memory has size M_{fast}

$$3b^2 <= M_{fast}, \text{ so } q \sim= b <= \text{sqrt}(M_{fast}/3)$$

To build a machine to run matrix multiply at the peak arithmetic speed of the machine, we need a fast memory of size

$$M_{fast} >= 3b^2 \sim= 3q^2 = 3(T_m/T_f)^2$$

This sizes are reasonable for L1 cache, but not for register sets

Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each C[i,j] by applying associativity
- The previous analysis showed that the blocked algorithm has computational intensity:

$$q \sim= b <= \mathrm{sqrt}(M_{fast}/3)$$

- There is a lower bound result that says we cannot do any better than this (using only algebraic associativity)

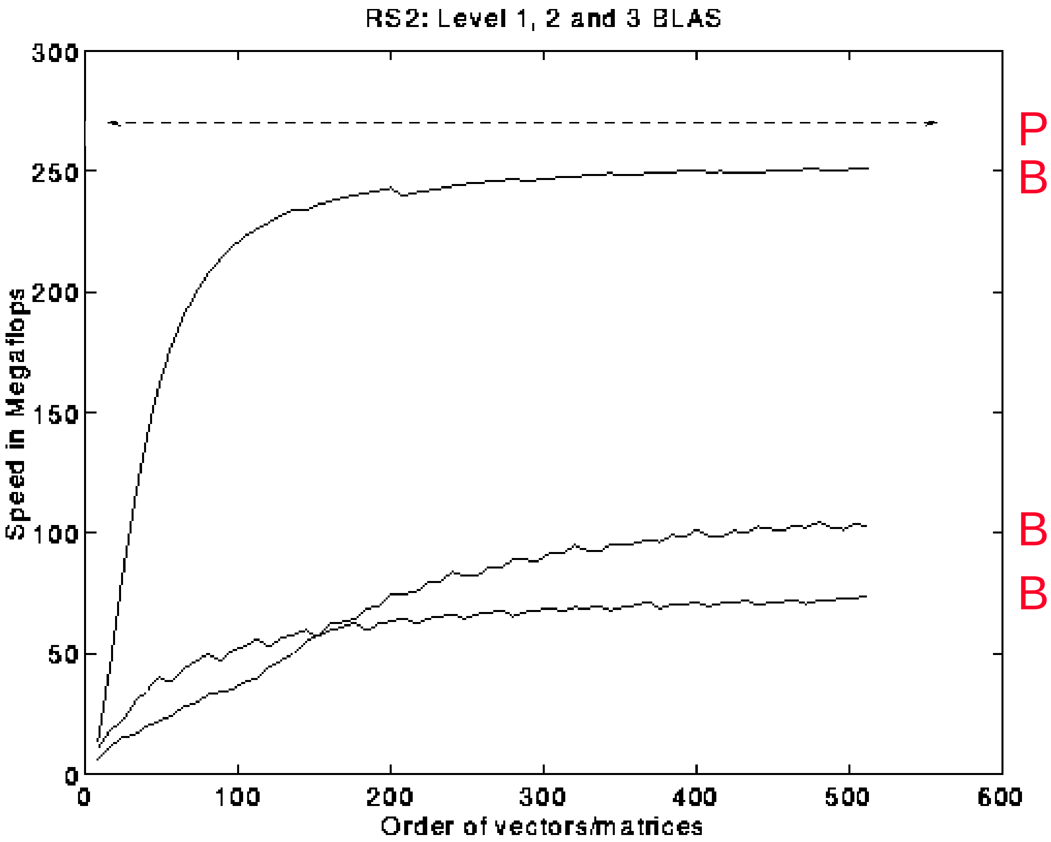
- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only algebraic associativity) is limited to q = $O(\mathrm{sqrt}(M_{fast}))$

Basic Linear Algebra Subroutines

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y=\alpha*x+y$), etc
    - m=2*n, f=2*n, q ~1 or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - m=n^2, f=2*n^2, q~2, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - m >= 4n^2, f=O(n^3), so q can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible (LAPACK)
  - See `www.netlib.org/blas, www.netlib.org/lapack`

BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops



Peak
BLAS 3

BLAS 2

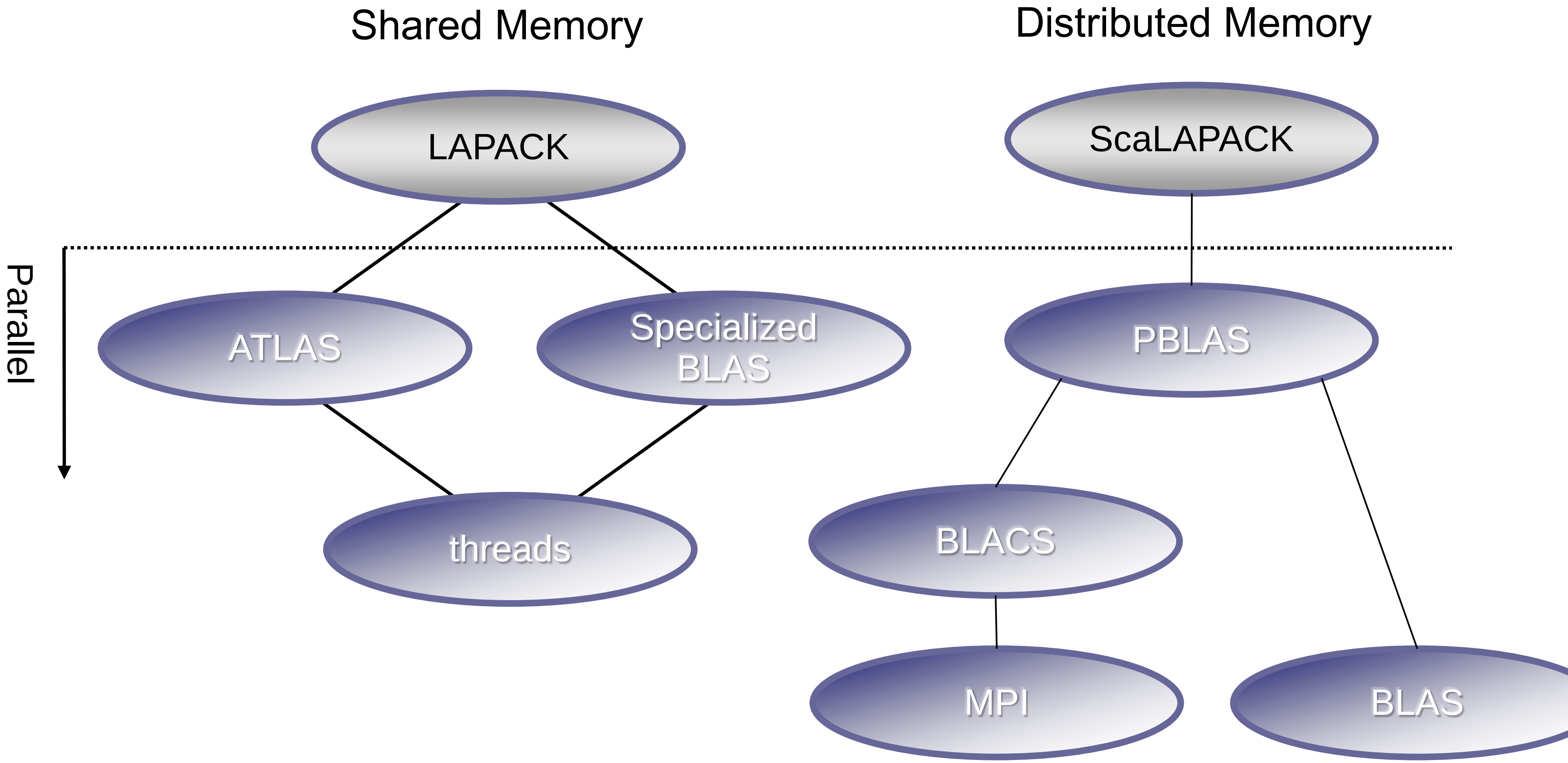
BLAS 1

BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

Search Over Block Sizes

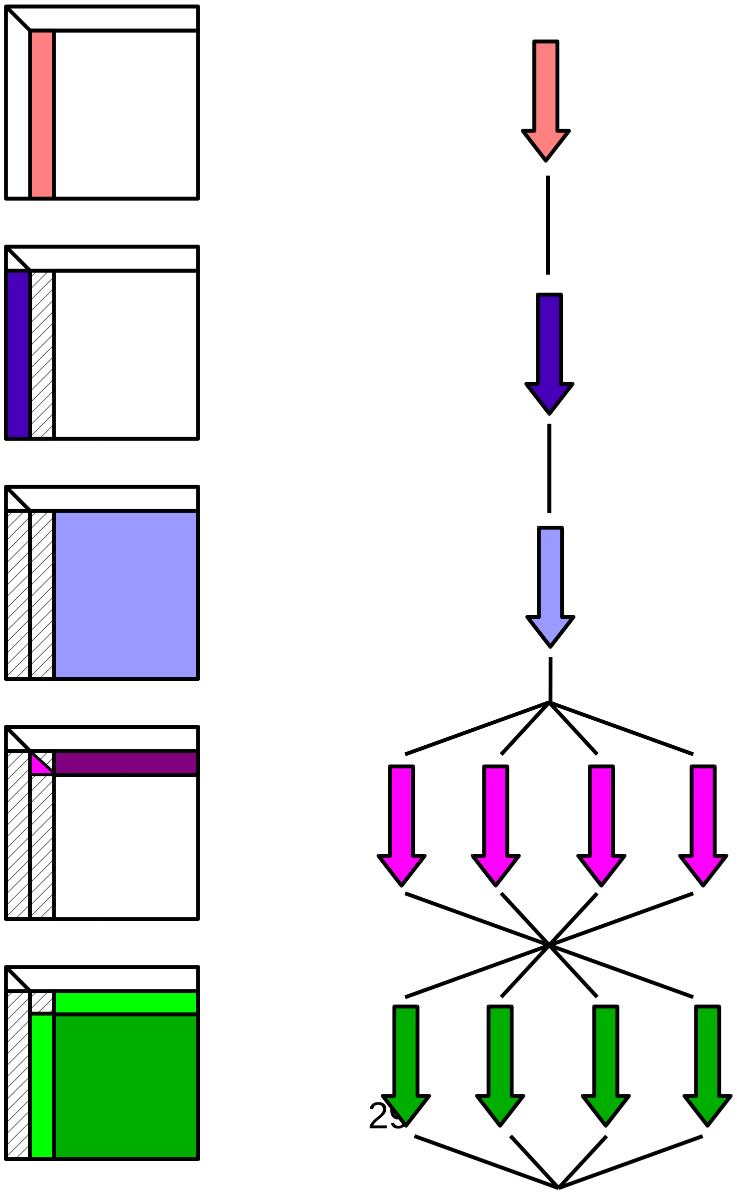
- **Performance models are useful for high level algorithms**
  - **Helps in developing a blocked algorithm**
  - **Models have not proven very useful for block size selection**
    - **too complicated to be useful**
    - **too simple to be accurate**
      - Multiple multidimensional arrays, virtual memory, etc.
- **Some systems use search**
  - **Atlas**
  - **BeBOP**
- **Graph Based Approach is now used – Plasma**

Parallelism in LAPACK / ScaLAPACK



Two well known open source software efforts for dense matrix problems.

Steps in the LAPACK LU



DGETF2 (Factor a panel) LAPACK

DLSWP (Backward swap) LAPACK

DLSWP (Forward swap) LAPACK

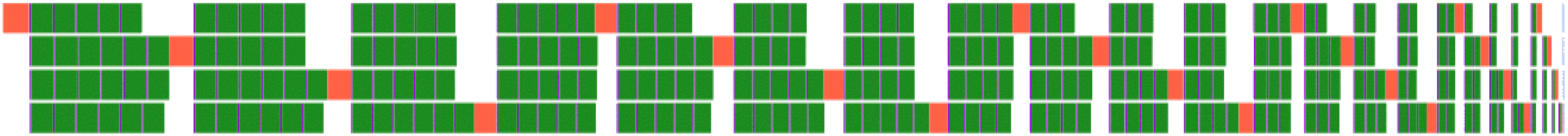
DTRSM (Triangular solve) **BLAS**

DGEMM (Matrix by Matrix multiply) **BLAS**

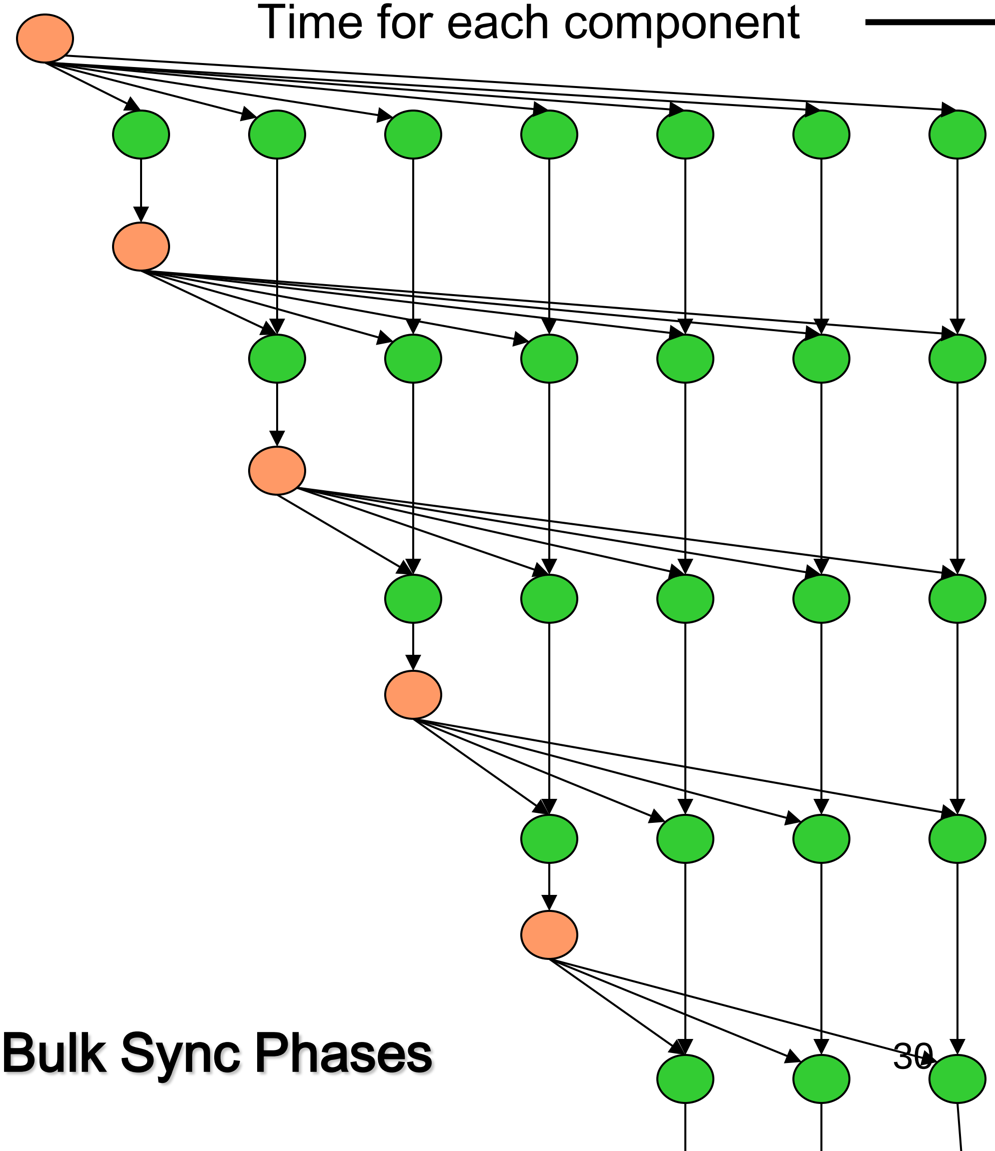
Most of the work done here

LU Timing Profile (4 Core System)

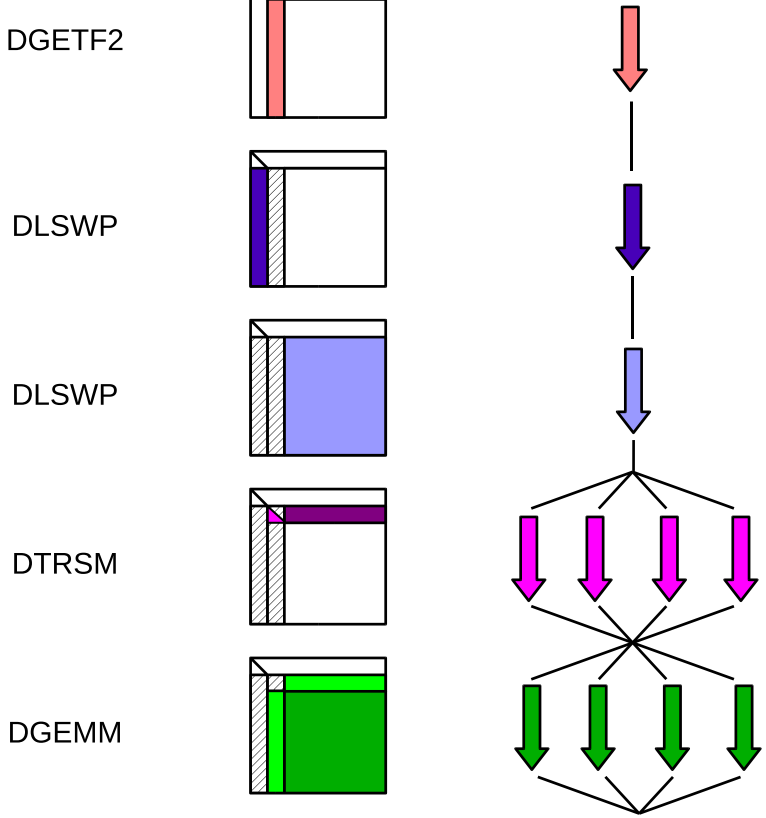
Threads – no lookahead



Time for each component → 1D decomposition

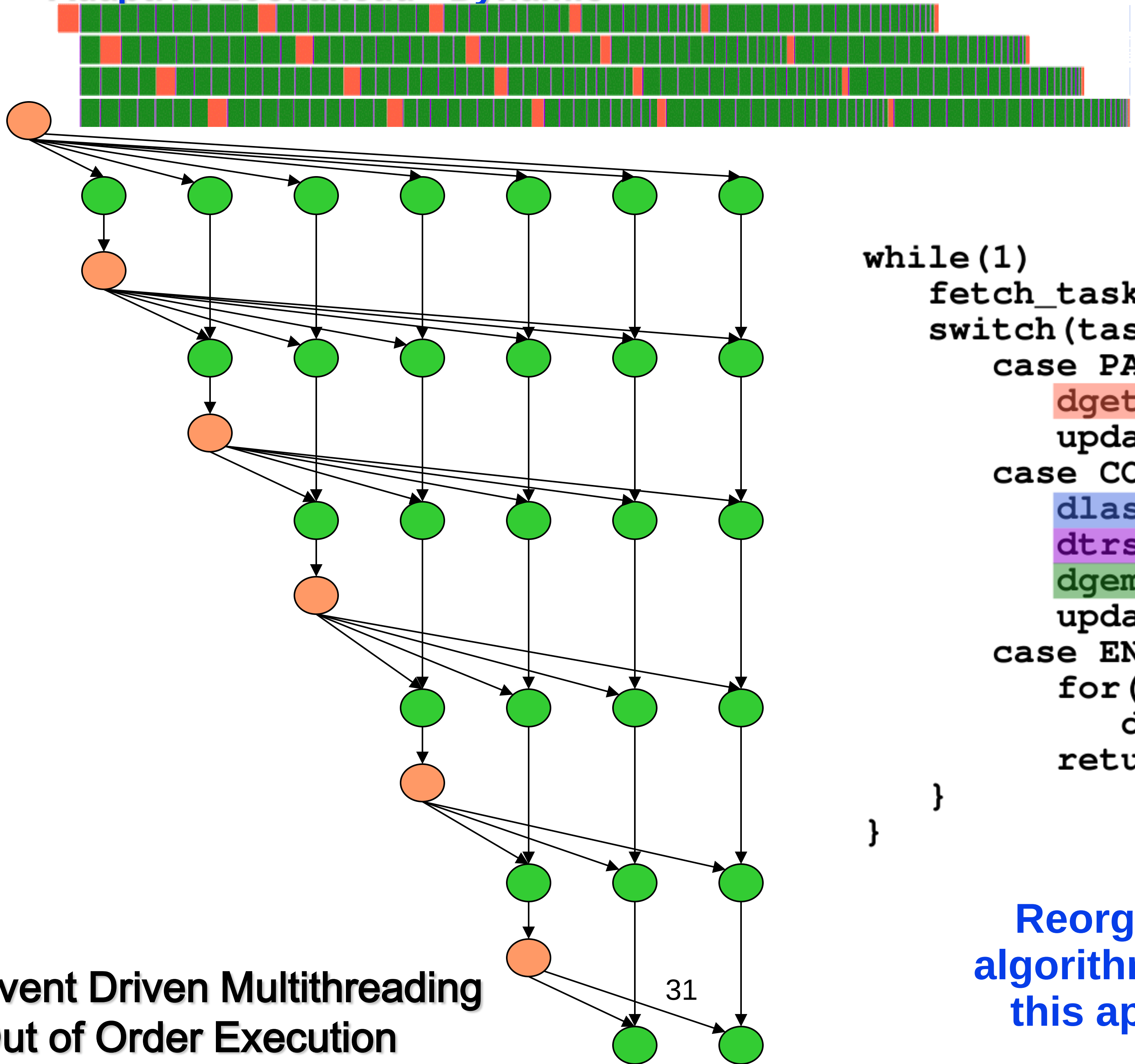


- 🟧 DGETF2
- 🟦 DLASWP(L)
- 🟦 DLASWP(R)
- 🟪 DTRSM
- 🟩 DGEMM



Bulk Sync Phases

Adaptive Lookahead - Dynamic

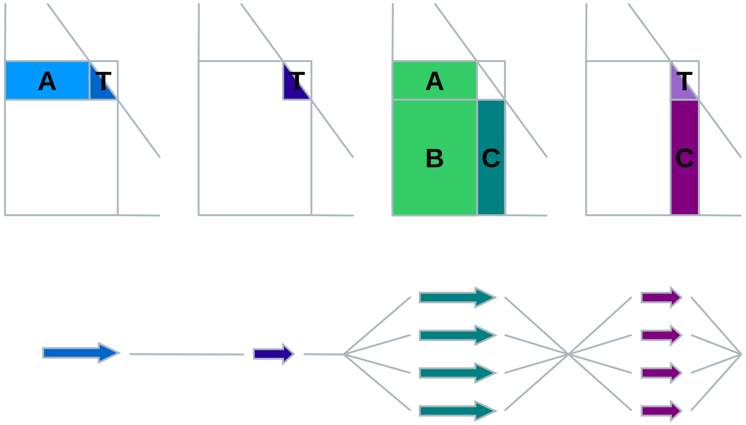


```
while(1)
    fetch_task();
    switch(task.type) {
        case PANEL:
            dgetf2();
            update_progress();
        case COLUMN:
            dlaswp();
            dtrsm();
            dgemm();
            update_progress();
        case END:
            for()
                dlaswp();
            return;
    }
}
```

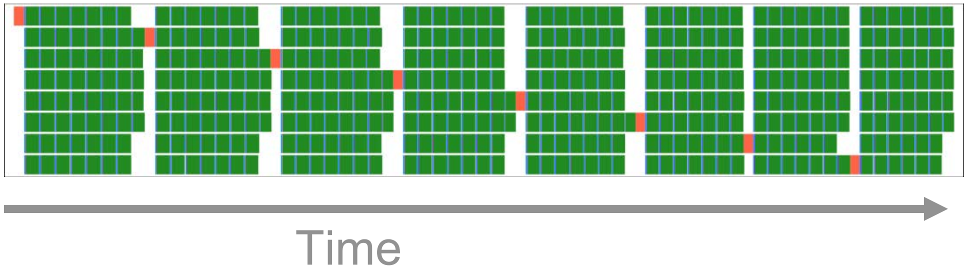
Event Driven Multithreading
Out of Order Execution

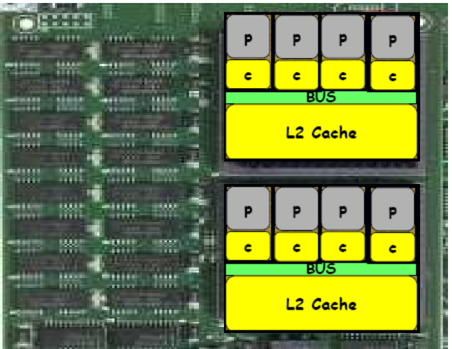
Reorganizing algorithms to use this approach

Fork-Join vs.
Dynamic Execution



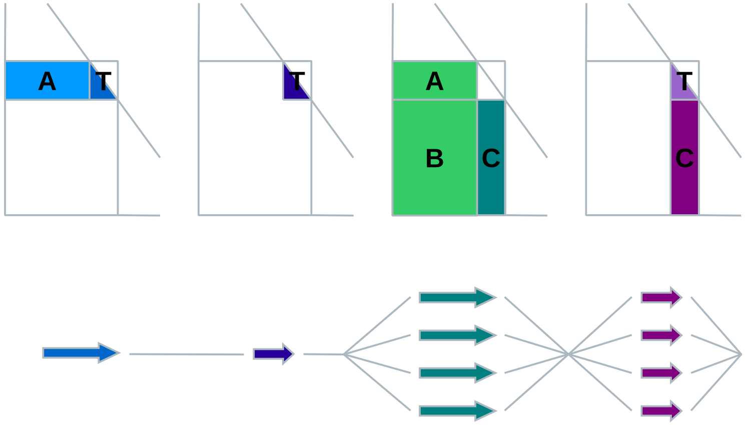
Fork-Join – parallel BLAS

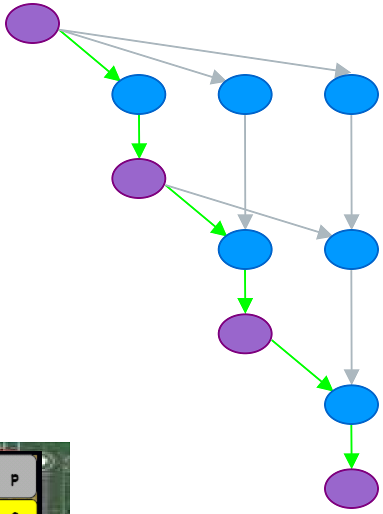




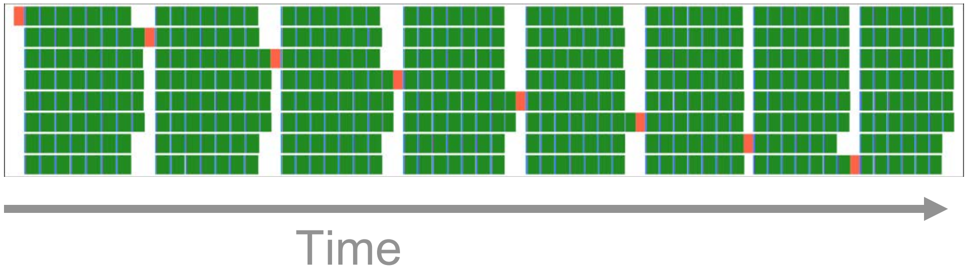
Experiments on
Intel’s Quad Core Clovertown
with 2 Sockets w/ 8 Treads

Fork-Join vs. Dynamic Execution

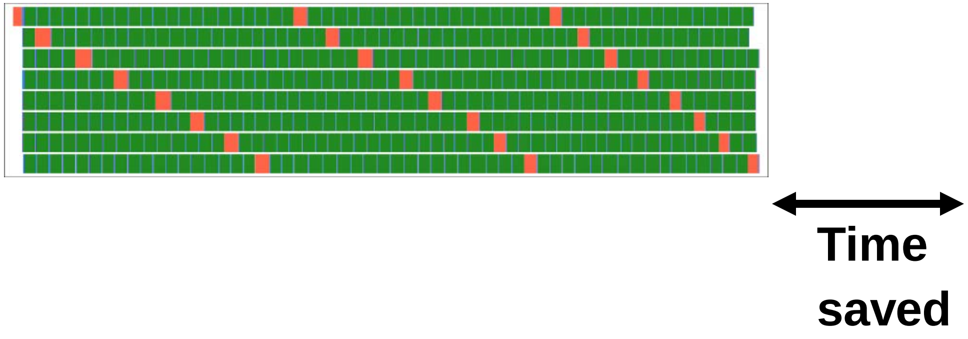


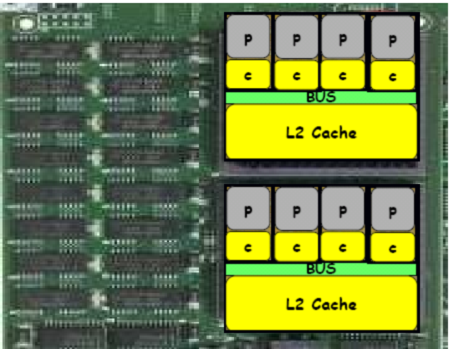


Fork-Join – parallel BLAS



DAG-based – dynamic scheduling





Experiments on
Intel's Quad Core Clovertown
with 2 Sockets w/ 8 Treads

Cholesky factorization

Consider a system of linear equations

$$A\,x = b,$$

where A is symmetric positive definite (SPD). This means

$$z^TA\,z >= 0 \text{ for all nonzero } x$$

We solve this by computing the Cholesky factorization

$$A = L\,L^T$$

and then solve by successive forward and backward substitution

$$Ly = b \qquad\qquad L^T x = y.$$

Cholesky factorization algorithm

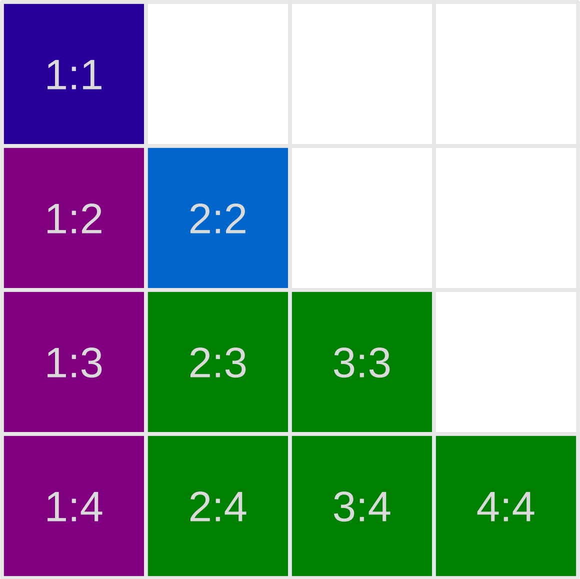
```
for j = 1, n
    for k = 1, j - 1
        for i = j, n
            a(i,j) = a(i,j) – a(i,k)* a(j,k);
          end
     end
     a(j,j) = sqrt (a(j,j))
     for k = j+1, n
          a(k,j) = a(k,j)/a(j,j);
     end
end
```

This is only one way to arrange the loops.

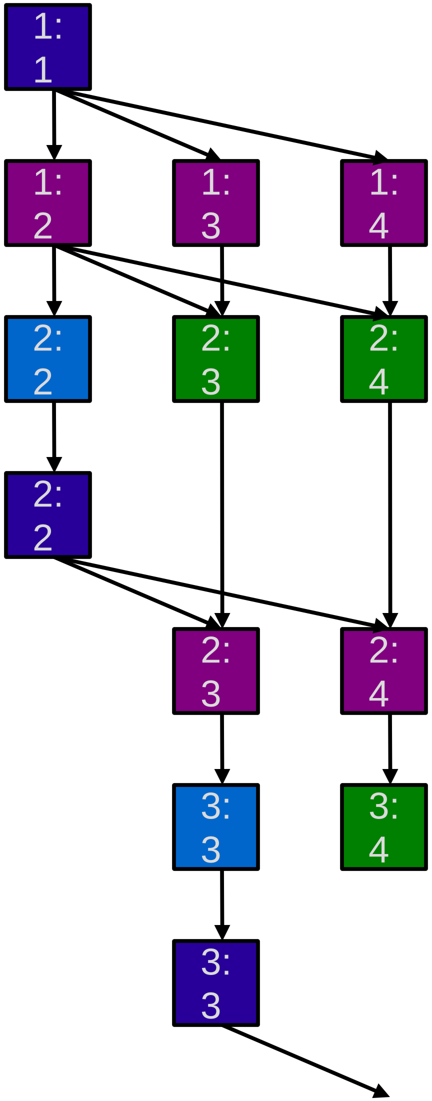
Cholesky factorization algorithm

- Since A is Symmetric Positive Definite the square roots are taken from positive numbers
- No pivoting is needed
- Only the lower triangle L is ever accessed and overwrites A
- Each column j is modified by a multiple of each prior column
- Elements of A which were non-zero become zero - fill-in

Cholesky Factorization
DAG-based Dependency Tracking



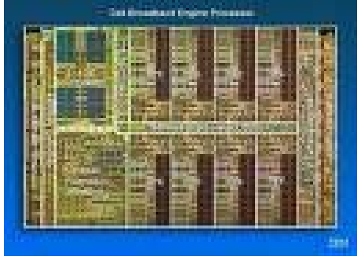
1:1			
1:2	2:2		
1:3	2:3	3:3	
1:4	2:4	3:4	4:4

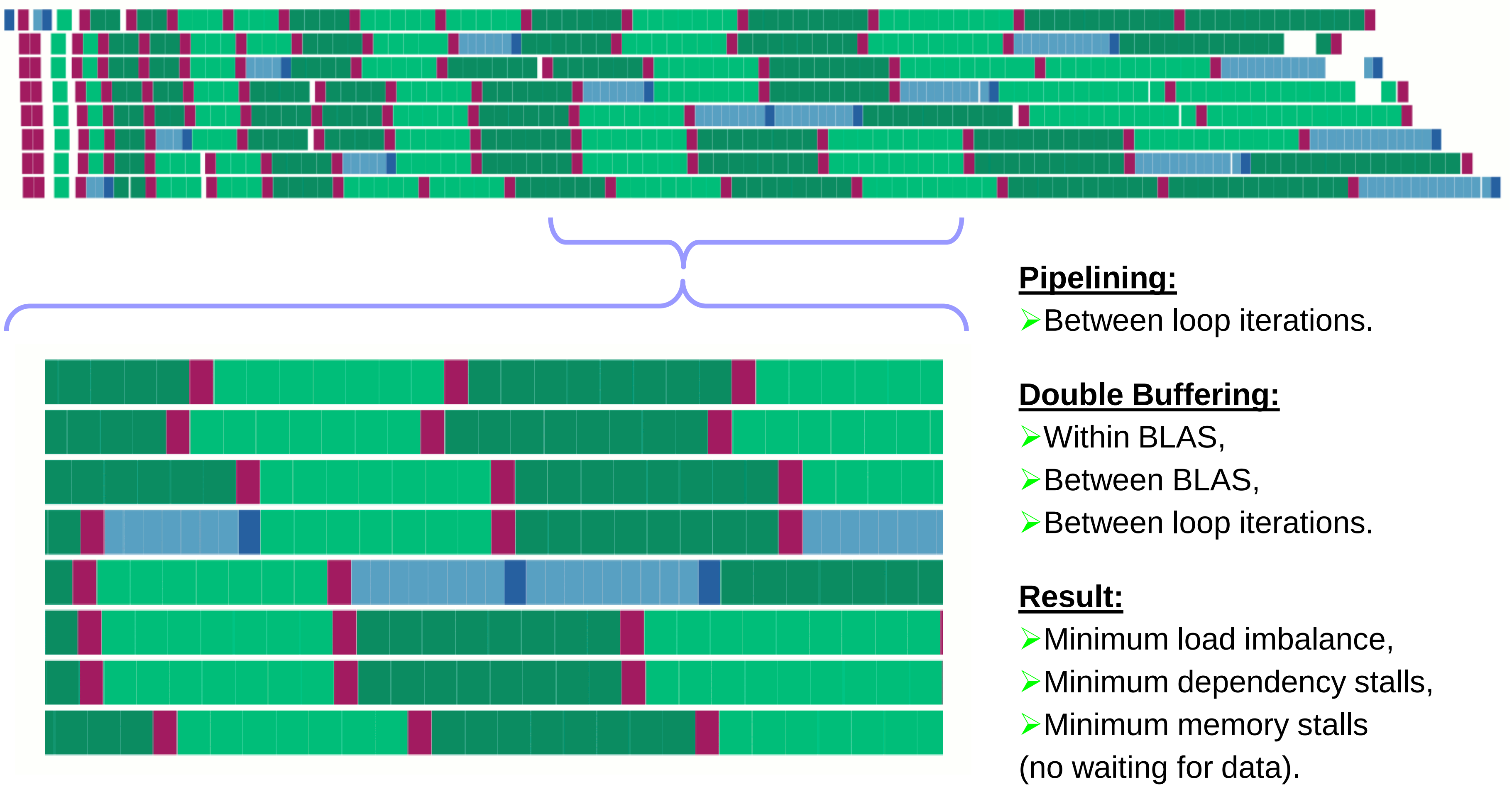


<u>Dependencies expressed by the DAG are enforced on a tile basis:</u>

- fine-grained parallelization
- flexible scheduling

Cholesky on the IBM Cell





Pipelining:

- Between loop iterations.

Double Buffering:

- Within BLAS,
- Between BLAS,
- Between loop iterations.

Result:

- Minimum load imbalance,
- Minimum dependency stalls,
- Minimum memory stalls

(no waiting for data).

Achieves 174 Gflop/s; 85% of peak in SP.

How to Deal with Architectural and Algorithmic Complexity?

- Adaptivity is the key for applications to effectively use available resources whose complexity is exponentially increasing
- Goal:
  - Automatically bridge the gap between the application and computers that are rapidly changing and getting more and more complex
- Achieving this Goal
  - Writing programs as collections of tasks with dependencies is one way to achieve this as it allows the specification of parallelism to be decoupled from the implementation
  - This approach also allows tasks to be executed when they can be and not to be subject to some arbitrary ordering
  - An important side effect of this is that communication is to some extent overlapped with computation
  - A major challenge with this approach is that the run-time system has to be very efficient.
- Examples – Plasma , Charm++, Uintah and CnC concurrent collections from Intel

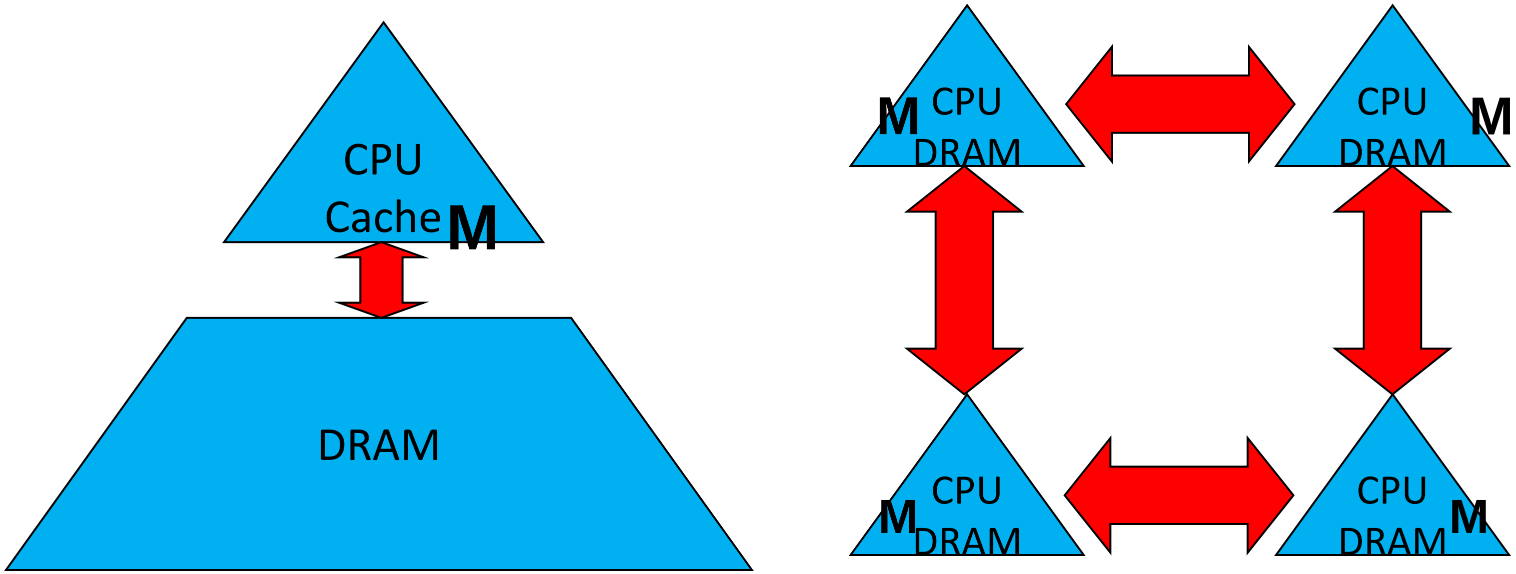
Summary of CA Linear Algebra

- "Direct" Linear Algebra
  - Lower bounds on communication for linear algebra problems like Ax=b, least squares, Ax = λx, SVD, etc
  - Mostly not attained by algorithms in standard libraries
  - New algorithms that attain these lower bounds
    - Being added to libraries: Sca/LAPACK, PLASMA, MAGMA
    - Large speed-ups possible
  - Autotuning to find optimal implementation
- Ditto for "Iterative" Linear Algebra

Avoiding communication helps performance

Algorithms have two costs (measured in time or energy):

1. Arithmetic (FLOPS)
2. Communication: moving data between
   - levels of a memory hierarchy (sequential case)
   - processors over a network (parallel case).



Fast memory of size M

Lower bound for all “n^3-like” linear algebra

- Let M = “fast” memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

#messages_sent ≥ #words_moved / largest_message_size

#messages_sent (per processor) = Ω(#flops (per processor) / $M^{3/2}$)

- Parallel case: assume either load or memory balanced
- Holds for
  - Matmul, BLAS, LU, QR, eig, SVD, and others
  - ense and sparse matrices (where #flops << n^3)
  - Sequential and parallel algorithms

Lower bound F(x) = Ω(g(x)) if 0 < c g(x) < f(x) for some c and x > x_0

Strassen's Algorithm for Matrix Multiplication

$$\begin{array}{|c|c|}\hline c_{11} & c_{12} \\ \hline c_{21} & c_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline a_{11} & a_{12} \\ \hline a_{21} & a_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline b_{11} & b_{12} \\ \hline b_{21} & b_{22} \\ \hline \end{array}$$

$$d_1 = (a_{11}+a_{22}) * (b_{11}+b_{22})$$

$$d_2 = (a_{12}-a_{22}) * (b_{21}+b_{22})$$

$$d_3 = (a_{11}-a_{21}) * (b_{11}+b_{12})$$

$$d_4 = (a_{11}+a_{12}) * (b_{22})$$

$$d_5 = (a_{21}+a_{22}) * (b_{11})$$

$$d_6 = (a_{11}) * (b_{12}-b_{22})$$

$$d_7 = (a_{22}) * (-b_{11}+b_{21})$$

$$C_{11} = d_1 + d_2 - d_4 + d_7$$

$$C_{12} = d_4 + d_6$$

$$C_{21} = d_5 + d_7$$

$$C_{22} = d_1 - d_3 - d_5 + d_6$$

$d_1 = (a_{11}+a_{22}) * (b_{11}+b_{22})$

$d_2 = (a_{12}-a_{22}) * (b_{21}+b_{22})$

$d_3 = (a_{11}-a_{21}) * (b_{11}+b_{12})$

$d_4 = (a_{11}+a_{12}) * (b_{22})$

$d_5 = (a_{21}+a_{22}) * (b_{11})$

7 multiplications and 18 Additions or Subtractions

$d_6 = (a_{11}) * (b_{12}-b_{22})$

$d_7 = (a_{22}) * (-b_{11}+b_{21})$

$C_{11} = d_1 + d_2 - d_4 + d_7$

$C_{21} = d_5 + d_7$

$C_{12} = d_4 + d_6$

$C_{22} = d_1 - d_3 - d_5 + d_6$

Strassen's Algorithm for Matrix Multiplication

$$\begin{array}{|c|c|}\hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array} = \begin{array}{|c|c|}\hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} * \begin{array}{|c|c|}\hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array}$$

$T(n)$ = Time to multiply two n by n matrices.

$T(n)= 7\ T(n/2) + 18(n/2)^2$

Solution: $T(n)= O(\ n^{k})$ where $k= \log_2(7)$.

Recursive Use of Strassen`s algorithm

```
func C = StrMM (A, B, n)
    if n=1 (or small enough),  C = A * B, else
      {  $P_1$ = StrMM ($A_{12}$ - $A_{22}$ , $B_{21}$ + $B_{22}$ , n/2)
         $P_2$ = StrMM ($A_{11}$ + $A_{22}$ , $B_{11}$ + $B_{22}$ , n/2)
         $P_3$ = StrMM ($A_{11}$ - $A_{21}$ , $B_{11}$ + $B_{12}$ , n/2)
         $P_4$ = StrMM ($A_{11}$ + $A_{12}$ , $B_{22}$ , n/2)
         $P_5$ = StrMM ($A_{11}$ , $B_{12}$ - $B_{22}$ , n/2)
         $P_6$ = StrMM ($A_{22}$ , $B_{21}$ – $B_{11}$ , n/2)
         $P_7$ = StrMM ($A_{21}$ + $A_{22}$ , $B_{11}$ , n/2)
         $C_{11}$ = $P_1$+ $P_2$ – $P_4$ + $P_{6,}$     $C_{12}$ = $P_4$+ $P_5$
         $C_{22}$ = $P_2$ -  $P_3$ + $P_5$ – $P_{7,}$     $C_{21}$ = $P_6$+ $P_7$ }
    return
```

$$
\begin{aligned}
T(n) &= \text{Cost of multiplying nxn matrices} \\
&= 7*T(n/2) + 18*(n/2)^2 \\
&= O(n^{\log_2 7}) \\
&= O(n^{2.81})
\end{aligned}
$$

Asymptotically faster
Several times faster for large n in practice
Cross-over depends on machine

Needs more memory than standard algorithm
Can be a little less accurate because of roundoff error

Communication Lower Bounds for Strassen-like matmul algorithms

Classical
$O(n^3)$ matmul:

#words_moved =
$\Omega\ (M(n/M^{1/2})^3/P)$

Strassen's
$O(n^{\lg 7})$ matmul:

#words_moved =
$\Omega\ (M(n/M^{1/2})^{\lg 7}/P)$

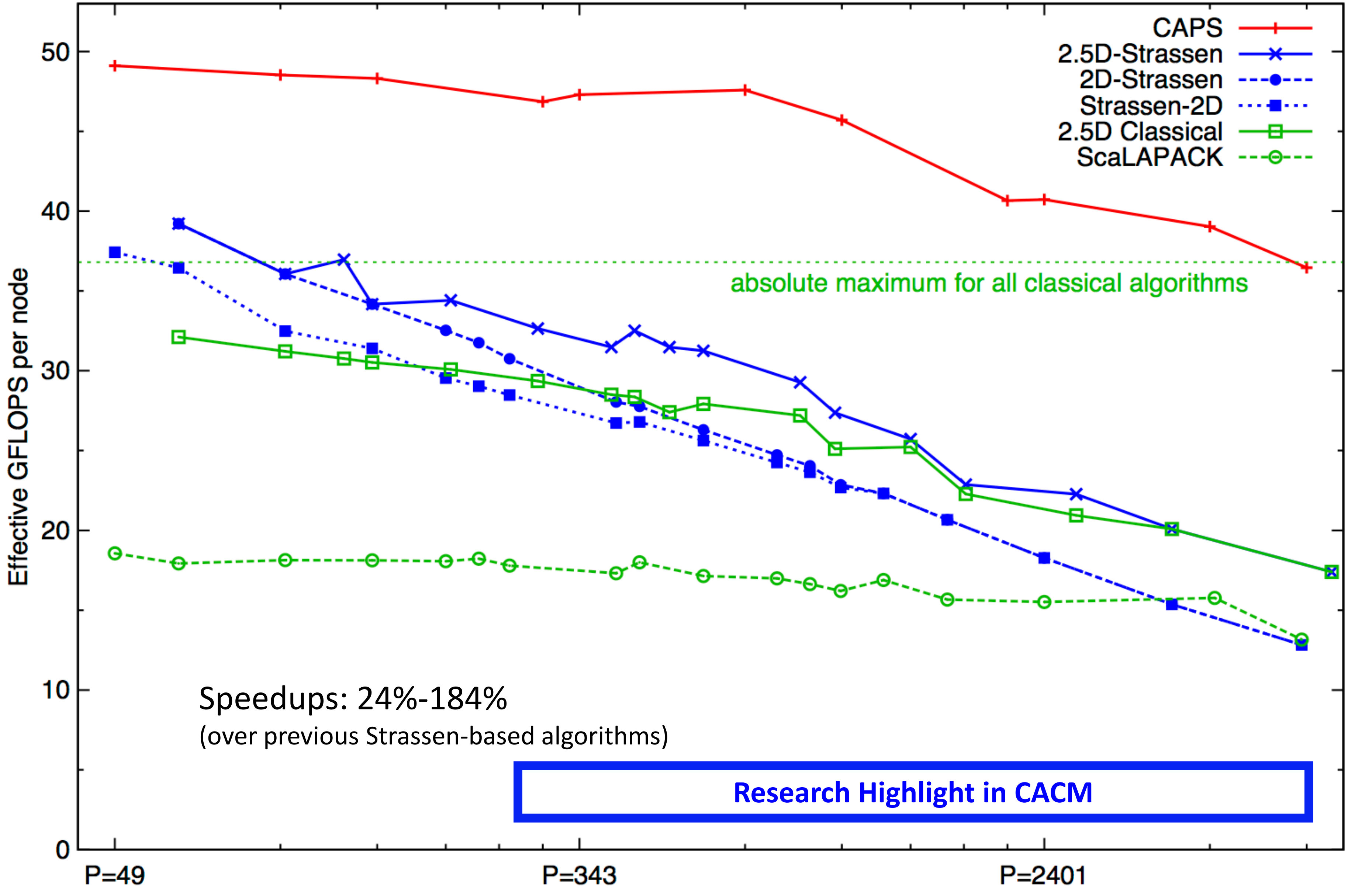
Strassen-like
$O(n^{\omega})$ matmul:

#words_moved =
$\Omega\ (M(n/M^{1/2})^{\omega}/P)$

- Proof: graph expansion (different from classical matmul)
  - Strassen-like: DAG must be "regular" and connected
- Extends up to $M = n^2 / p^{2/\omega}$
- Best Paper Prize (SPAA'11), Ballard, D., Holtz, Schwartz,
- Is the lower bound attainable?

Performance Benchmarking, Strong Scaling Plot

Franklin (Cray XT4) n = 94080



Speedups: 24%-184%

(over previous Strassen-based algorithms)

Research Highlight in CACM