CSE 891 - Section 1: Parallel Computing - Fundamentals and Applications

Fall 2014 - Lectures 4&5:

Interplay between Computer Architecture & Application Performance

H. Metin Aktulga

hma@cse.msu.edu

517-355-1646

Lecture 3 - Summary

- Parallel hardware
  - Hardware multithreading
  - SIMD systems (Vector processors, GPUs)
  - MIMD systems (Shared- vs Distributed-memory systems)

- Interconnection networks
  - Latency vs. bandwidth
  - Network diameter, bisection bandwidth
  - Switch vs. packet routing
  - Interconnect topologies: ring, hypercube, tori, trees

Modeling the application performance

- At any given time, a processor can be busy with one of the following:
  - Computation
  - Data access (to memory or cache)
  - Communication
  - Disk I/O (avoided in HPC except for program start/finish)

- A simple model for total execution time (excluding IO)
  - T_{total} = Flop time + Memory time + Comm time
    $= T_F + T_M + T_C$

Modeling the application performance

- $T_{total} = T_F + T_M + T_C$
- Definitions:
  - **f** = total # of floating point arithmetic (flop) operations
  - $\mathbf{t_f}$ = time per flop << t_m
  - **m** = total # of memory elements (words) moved between fast and slow memory
  - $\mathbf{t_m}$ = time per slow memory operation
  - **c** = total # of bytes communicated between compute nodes
  - $\mathbf{t_c}$ = average time for communicating one byte
- $T_{total} = f * t_f + m * t_m + c * t_c$

A note about memory access vs. network communication

- Recall our discussion about network comm:

  data transmission time = $\alpha + D/\beta$

- Same applies for memory access (with different values)
- Typically α is much lower for memory systems (short distances, fast links)
- Also prefetching mechanisms exist for hiding memory latency
- So t_m is generally determined by the bandwidth

Modeling the application performance

- Different operations can be overlapped with each other
- Prefetching:
  - overlapped computation and memory access
  - hardware/software prefetching
- Non-blocking communication primitives:
  - overlapped computation and communication
- Algorithmically overlapping computation/communication:
  - designate computation & communication threads
  - can still use blocking communication primitives

Modeling the application performance

- $T_{total} = f * t_f + m * t_m + c * t_c$
- If $T_{total} \approx c * t_c$ => performance is communication-bound
  - Performance improvements may be possible
  - Ex: graph traversals, sparse matrix computations
- If $T_{total} \approx c * t_c$ => performance is memory-bound
  - Data locality/access patterns may be improved
  - Ex: bioinformatics applications
- If $T_{total} \approx f * t_f$ => performance is CPU-bound
  - CPU performing close to its peak - as good as one can get
  - Ex: dense matrix computations

Modeling the application performance

- We will discuss communication later - let's talk about computation and memory interplay for now…
- Assumptions:
  - Only 2 levels in the hierarchy:

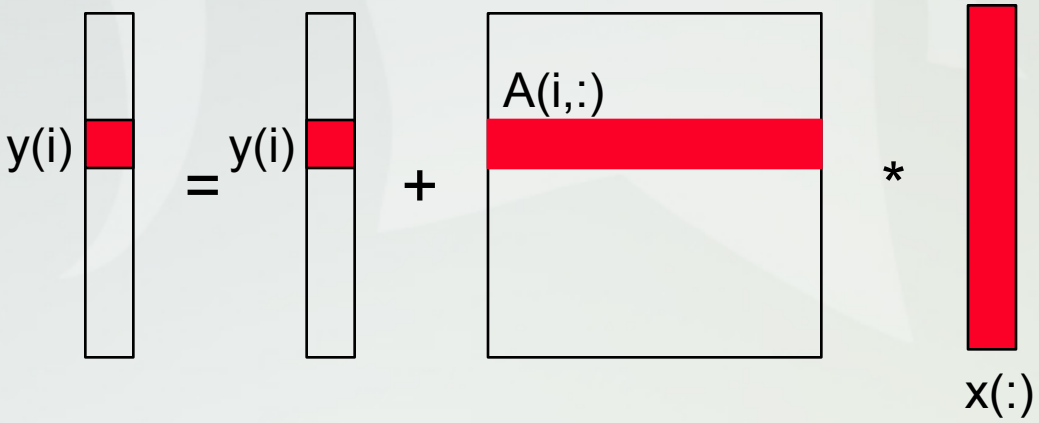
    fast (cache) and slow (memory)
  - All data are initially in the slow memory
- Arithmetic Intensity: **q = f / m**
  - average number of flops per slow memory access
  - key to the efficiency of an algorithm

Modeling the application performance

- $T_{total} = f * t_f + m * t_m = f * t_f * (1 + t_m/t_f * 1/q)$
  - t_m/t_f: machine balance - key to machine efficiency
  - Larger q => closer to minimum possible time
  - $q \geq t_m/t_f$ needed to get at least half of peak speed
- Minimum possible time = $f * t_f$
  - achieved if $(t_m/t_f * 1/q) \approx 0$
  - if all data in fast memory
  - if the arithmetic intensity is very very high
  - CPU-bound performance

Case study: Matrix-Vector multiply [6]

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



Case study: Matrix-Vector multiply [6]

```
{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 bytes accessed on slow memory = $3n + n^2$ doubles
- **f** = number of arithmetic operations = $2n^2$
- **q** = f / m ≈ 2 / 8 (in double precision)

=> Matrix-vector multiplication limited by the slow memory speed, because in general $t_f << t_m$

Case study: Matrix-Vector multiply

$$T_{total} = f * t_f * (1 + t_m/t_f * 1/q)$$

- Assume memory bandwidth is 20 GB/s, $q \approx 2/8 = 0.25$.
- What's the peak performance, if CPU peak is 20 Gflop/s?
- How about if CPU peak is 1 Gflop/s?

Case study: Matrix-Vector multiply

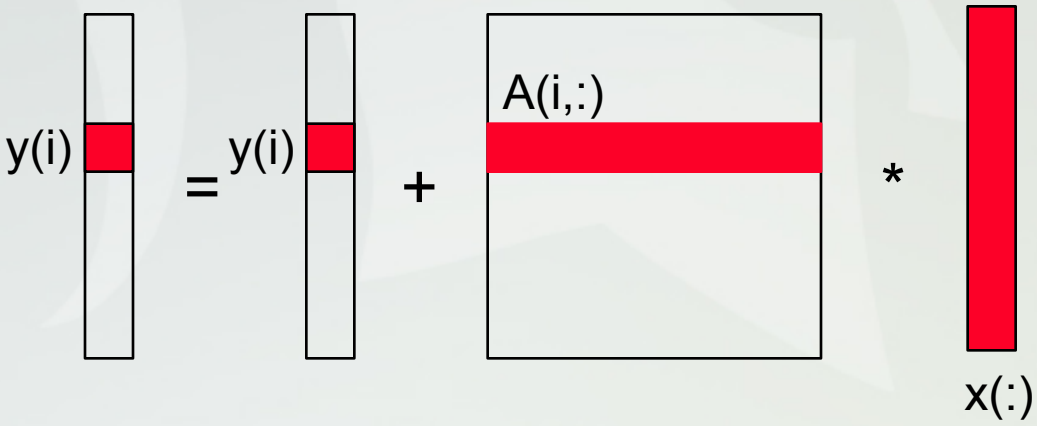
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
   {read row i of A into fast memory}
   for j = 1:n
      y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

- Note y & A(i,:) are accessed in bulk to reduce latency overheads
- Important assumptions:
  - Fast memory is large enough to hold 3n doubles
  - Time to access data in fast memory is 0
- Using (Intel) intrinsics, it's possible to control what is/isn't cached
  - So one can do fine with a cache (slightly) larger than n

Case study: Matrix-Vector multiply

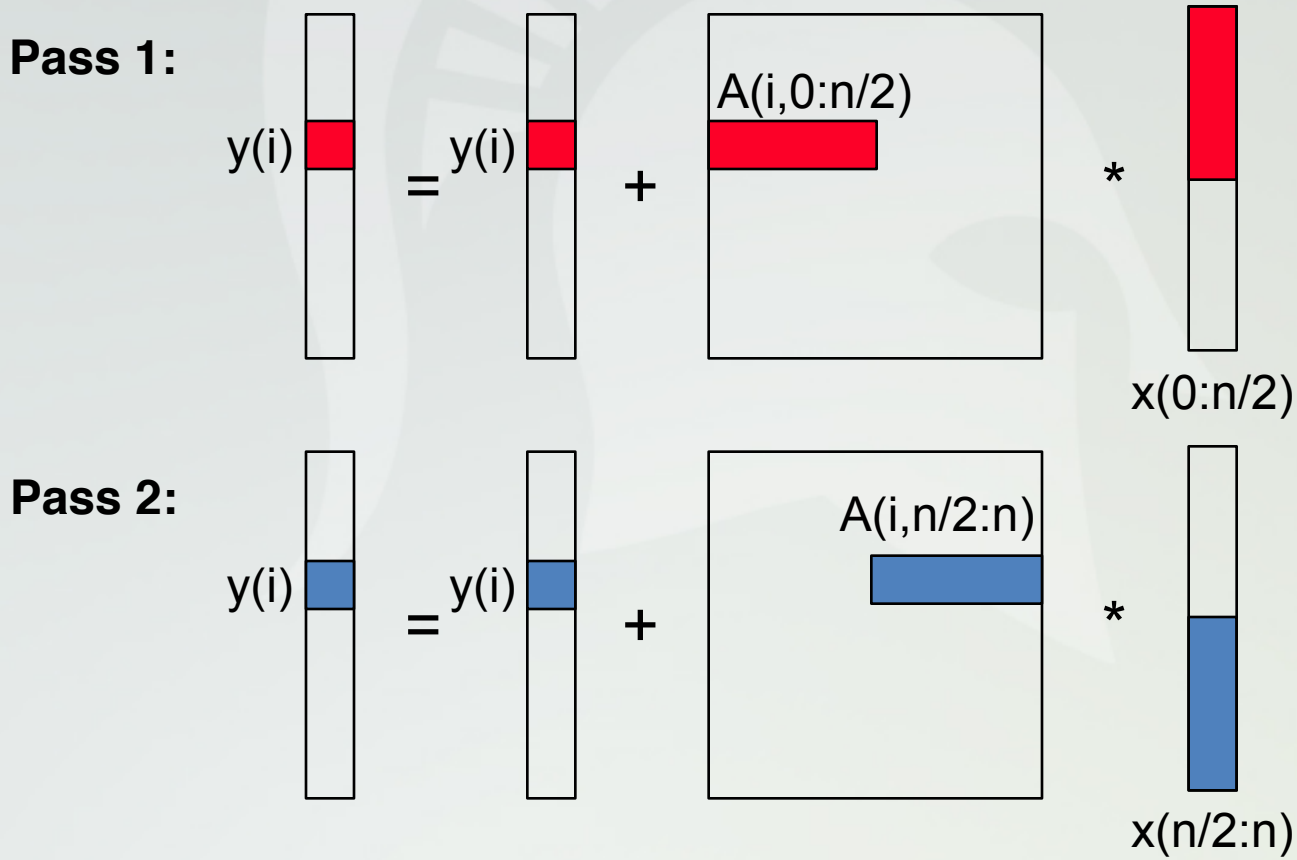
- How many cache misses would such an implementation incur on x, if the cache size was < n and LRU policy is used? **n^2/cache-line width**

  Is it possible to do better?



Case study: Matrix-Vector multiply

- A blocked algorithm should do better



Case study: Matrix-Vector multiply

With cache-blocking:

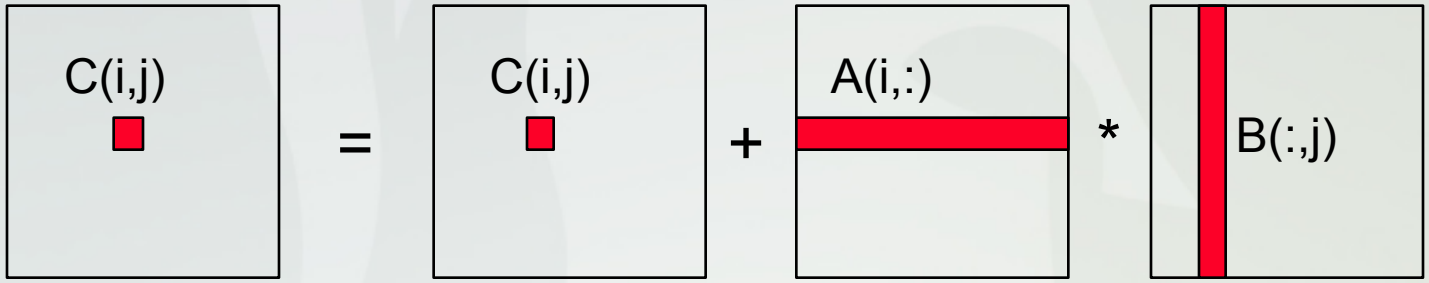
- read each element of **A** once from slow memory
- read each element of **x** once from slow memory
- read & write each element of y twice from/to memory
- total cache misses on x: n/cache-line width
- total cache misses on y: 2n/cache-line width

In practice though, you probably will not see any significant performance difference between the two implementations because:

- there are multiple levels of cache to ease programming
- there are advanced prefetching hardware to hide latencies between cache levels

A better example: Matrix multiply [6]

```
{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)
```

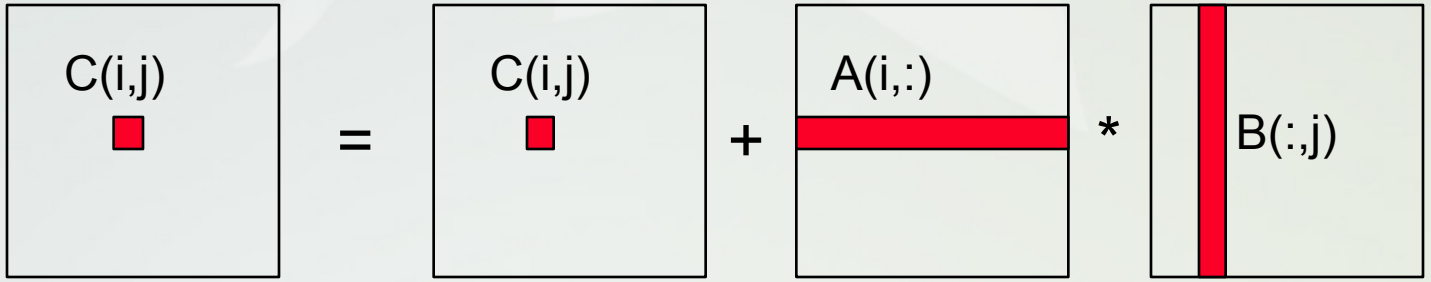


Algorithm has $2*n^3 = O(n^3)$ flops and operates on $3*n^2$ words (8 bytes for double) of memory

q can potentially be as large as $2*n^3/(3*n^2*8) = O(n)$

A better example: Matrix multiply [6]

```
{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}
```



A more interesting example: Matrix multiply [6]

Number of slow memory references on unblocked matrix multiply

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

$+ n^2$ to read each row of A once

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

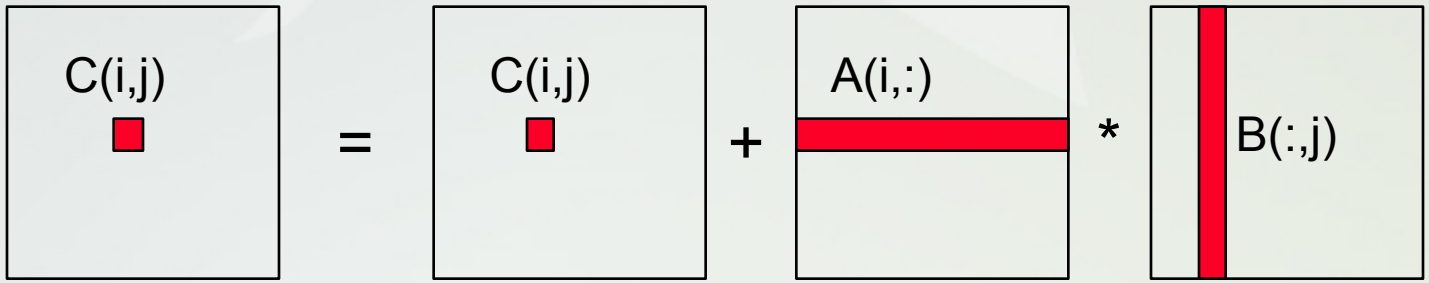
$= n^3 + 3n^2$

So $q = f / m = 2n^3 / (8*(n^3 + 3n^2))$

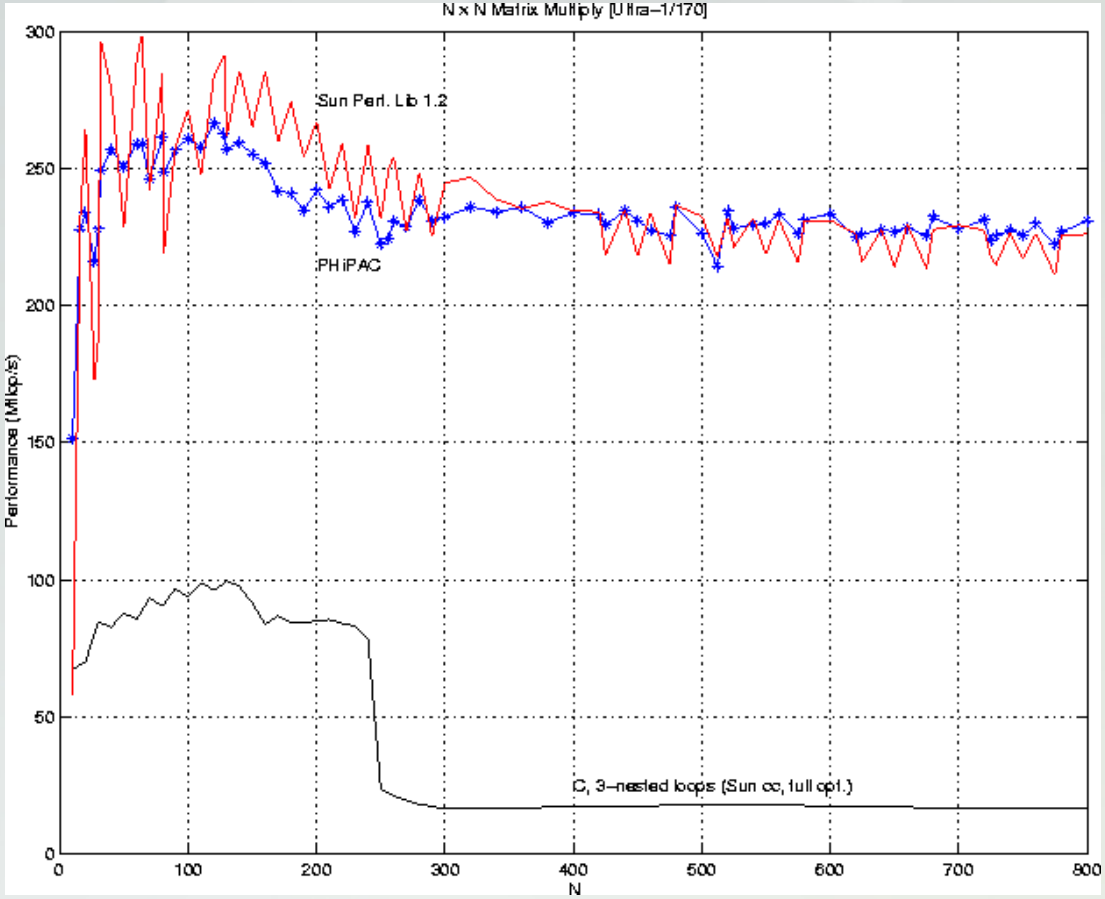
$\approx 2 / 8$ for large n - NO improvement over matrix-vector multiply!

Inner two loops are just matrix-vector multiply, of row i of A times B

Similar analysis holds for any other ordering of the 3 loops



Optimized Matrix-multiply vs. 3 loops [6]

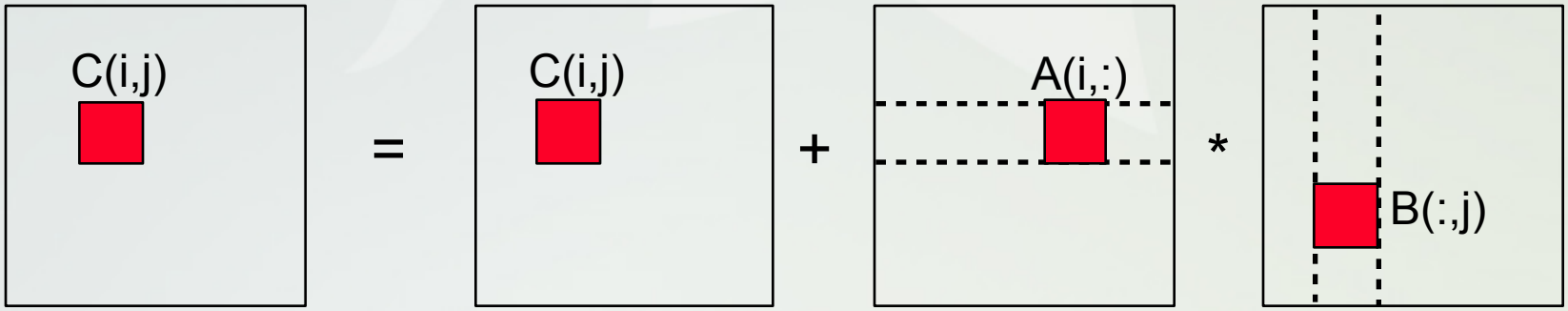


Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

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 [6]

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

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

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

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

$f = 2n^3$ for matrix multiplication

$\Rightarrow q = f / m = 2n^3 / ((2N + 2) * n^2) \approx n / N = b$ for large n

- So matrix multiply ($q \approx b$) can be much faster than matrix-vector multiply ($q = 2$)

Using Analysis to Design/Understand Machines

- Performance of blocked matrix multiplication increases with b
- **Limit:** All three blocks ($A_{(i,k)}$,$B_{(k,j)}$ and $C_{(i,j)}$) must fit in cache
- **M_{fast}:** cache (fast memory) size

  => $3b^2 \le M_{fast}$, so $b \le (M_{fast}/3)^{1/2}$
- **Recall:** $T_{total} = f * t_f * (1 + t_m/t_f * 1/q)$
  - $q \ge t_m/t_f$ to get 1/2 of peak speed
  - Arithmetic intensity $q \approx b$

  => $M_{fast} \ge 3b^2 \approx 3q^2 = 3(t_m/t_f)^2$
- Reasonable size for L1 cache, but not for registers

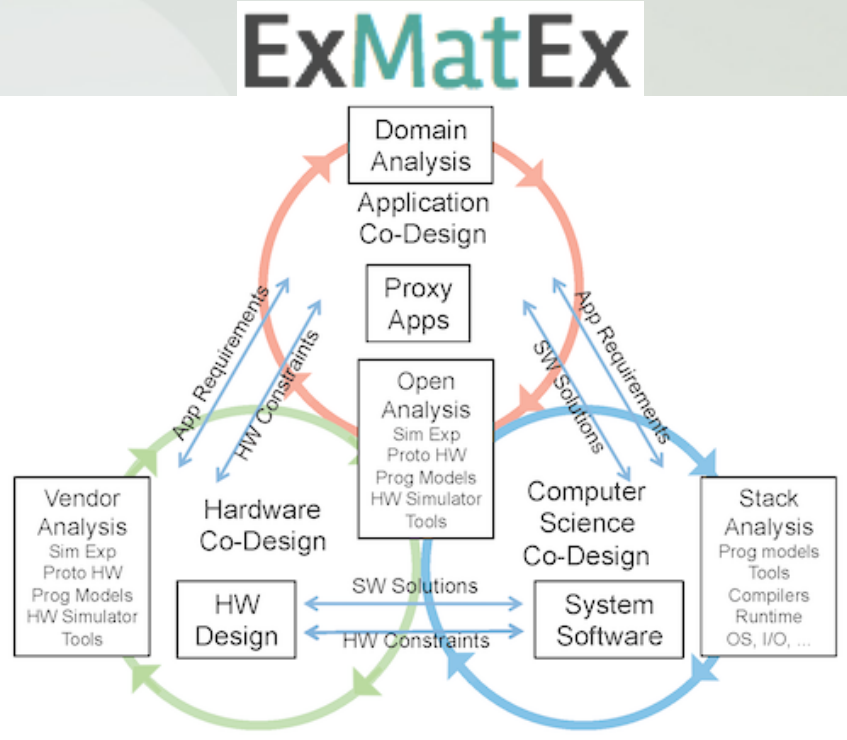
		required
	t_m/t_f	KB
Ultra 2i	24.8	14.8
Ultra 3	14	4.7
Pentium 3	6.25	0.9
Pentium3M	10	2.4
Power3	8.75	1.8
Power4	15	5.4
Itanium1	36	31.1
Itanium2	5.5	0.7

The Idea of Co-design

- The analysis above is a perfect example for co-design
- **Co-design:** "a computer system design process where scientific problem requirements influence architecture design and technology and constraints inform formulation and design of algorithms and software.
- **DOE Co-design program's goal:** "To ensure that future architectures are well-suited for DOE target applications and that major DOE scientific problems can take advantage of the emerging computer architectures"
- However, in reality scientific & engineering applications are much more complex than matrix multiply - so multi-institution, multi-year efforts are needed.

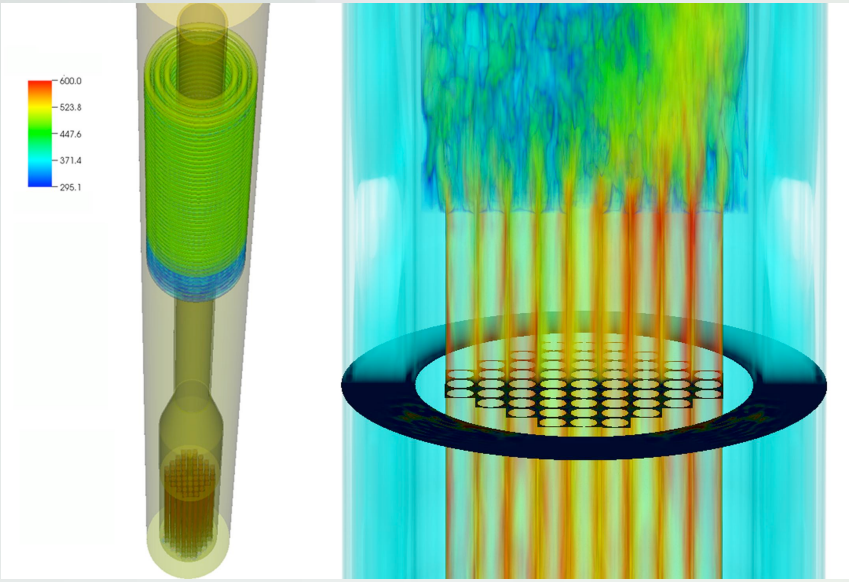
DOE Co-design projects - 1

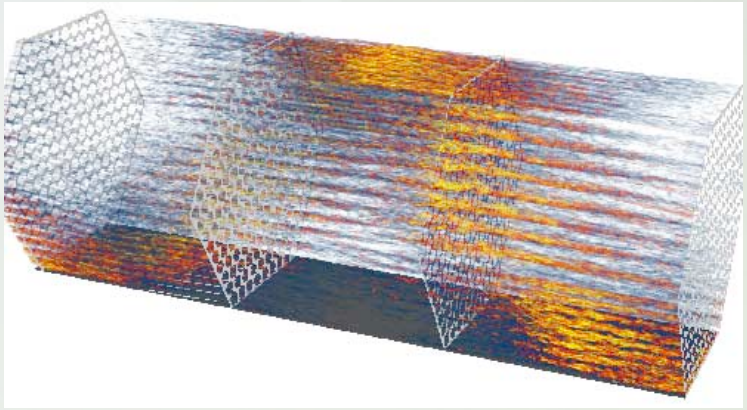
- ExMatEx: Exascale Co-design Center for Materials in Extreme Environments
- **Target applications:** Molecular Dynamics and coarse-grain (meso-scale) models
- **Goal:** "to establish the interrelationships between hardware, middleware (software stack), programming models and algorithms to enable a productive exascale environment for multiphysics simulations of materials in extreme mechanical and radiation environments."



DOE Co-design projects - 2

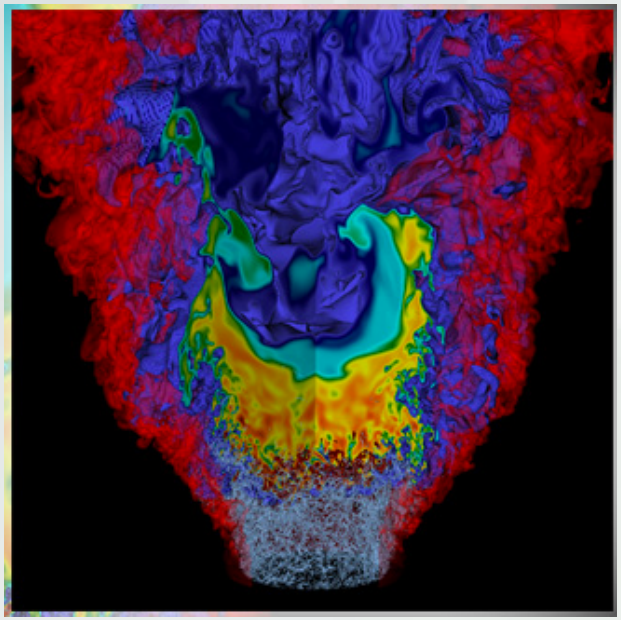
- CESAR: Center for Exascale Simulation of Advanced Reactors
- **Target Applications:** computational fluid dynamics and neutron transport

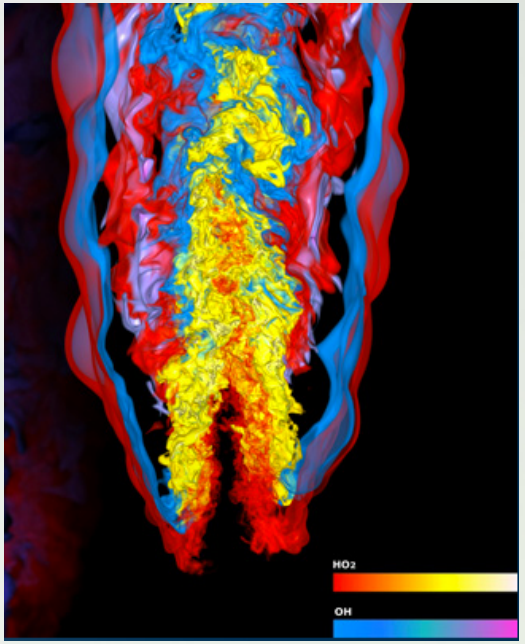




DOE Co-design projects - 3

- ExaCT: Center for Exascale Simulation of Combustion in Turbulence
- **Target applications:** Direct numerical simulation (DNS), Adaptive Mesh Refinement (AMR)





Back to matrix multiplication [6]

- The blocked algorithm changes the order in which values are accumulated into each C[i,j] by applying commutativity and associativity
  - Slightly different answers from naïve code due to roundoff - OK
- For the blocked algorithm, $q \approx b \leq (M_{fast}/3)^{1/2}$, and there is a lower bound result that says we cannot do any better than this (using only associativity, so still doing n^3 multiplications - no Strassen-like algorithms)
- **Theorem (Hong & Kung, 1981):** Any reorganization of the matrix multiplication algorithm is limited to $q = O((M_{fast})^{1/2})$

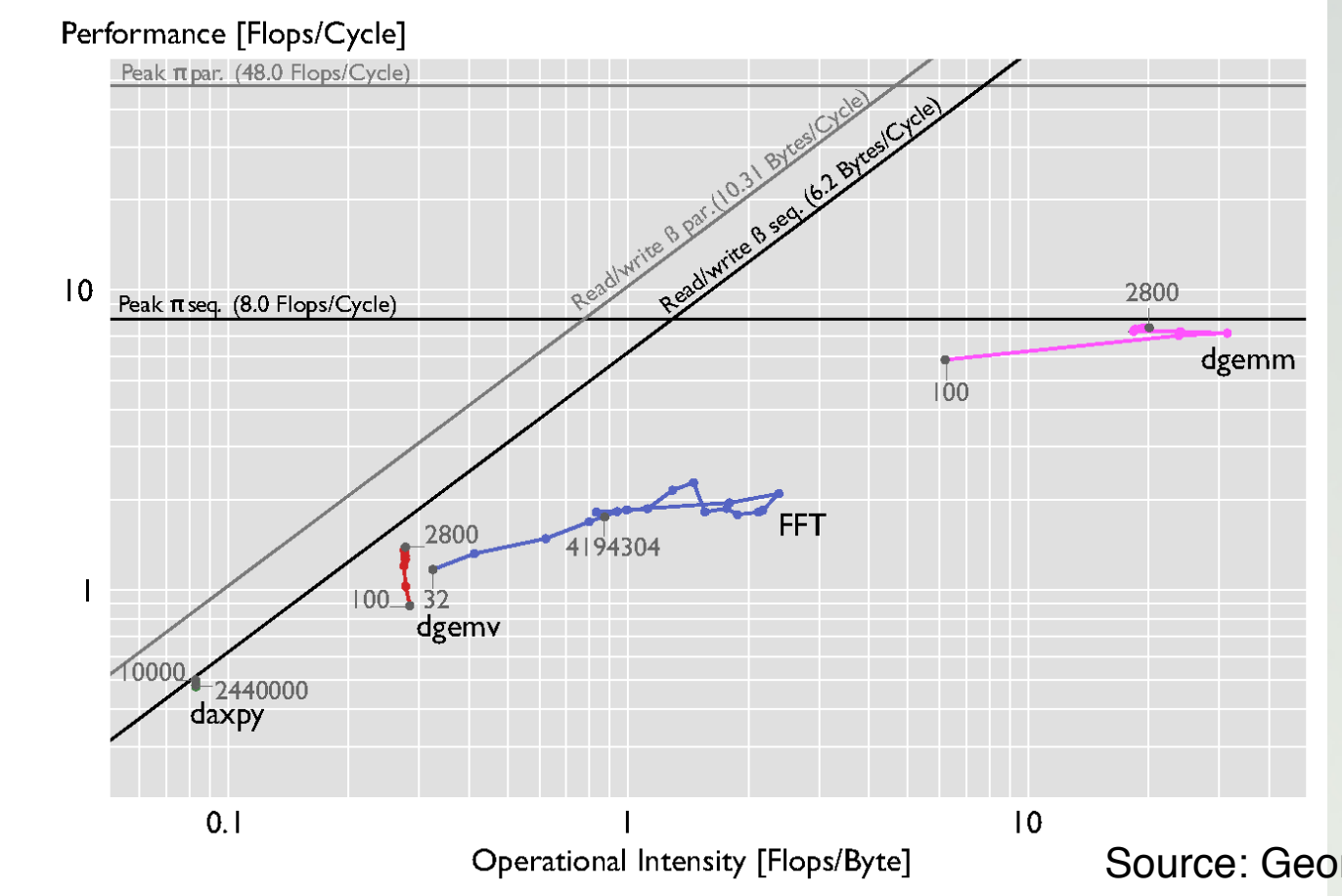
How to implement blocked matrix-multiply?

- We assumed a single-level of cache
- In practice, there are 2-3 levels of cache
- Cache levels and sizes will change from one architecture to another
- How to implement portable code and make sure that you get good performance on different architectures?
- One option is to use libraries
- Another is to use cache-oblivious algorithms

Libraries: Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving) - www.netlib.org/blas
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y=\alpha*x+y$), etc.
    - $m = 2n$, $f = 2n$, $q = f/m$ = computational intensity ~ 1 or less
  - BLAS2 (mid 1980s):
    - matrix-vector operations: matrix vector multiply, etc.
    - $m=n^2$, $f=2n^2$, $q\sim2$, less overhead - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc.
    - $m <= 3n^2$, $f=O(n^3)$, so $q = f/m$ is typically very large, BLAS3 is potentially much faster than BLAS2
- Good algorithms use BLAS3 when possible

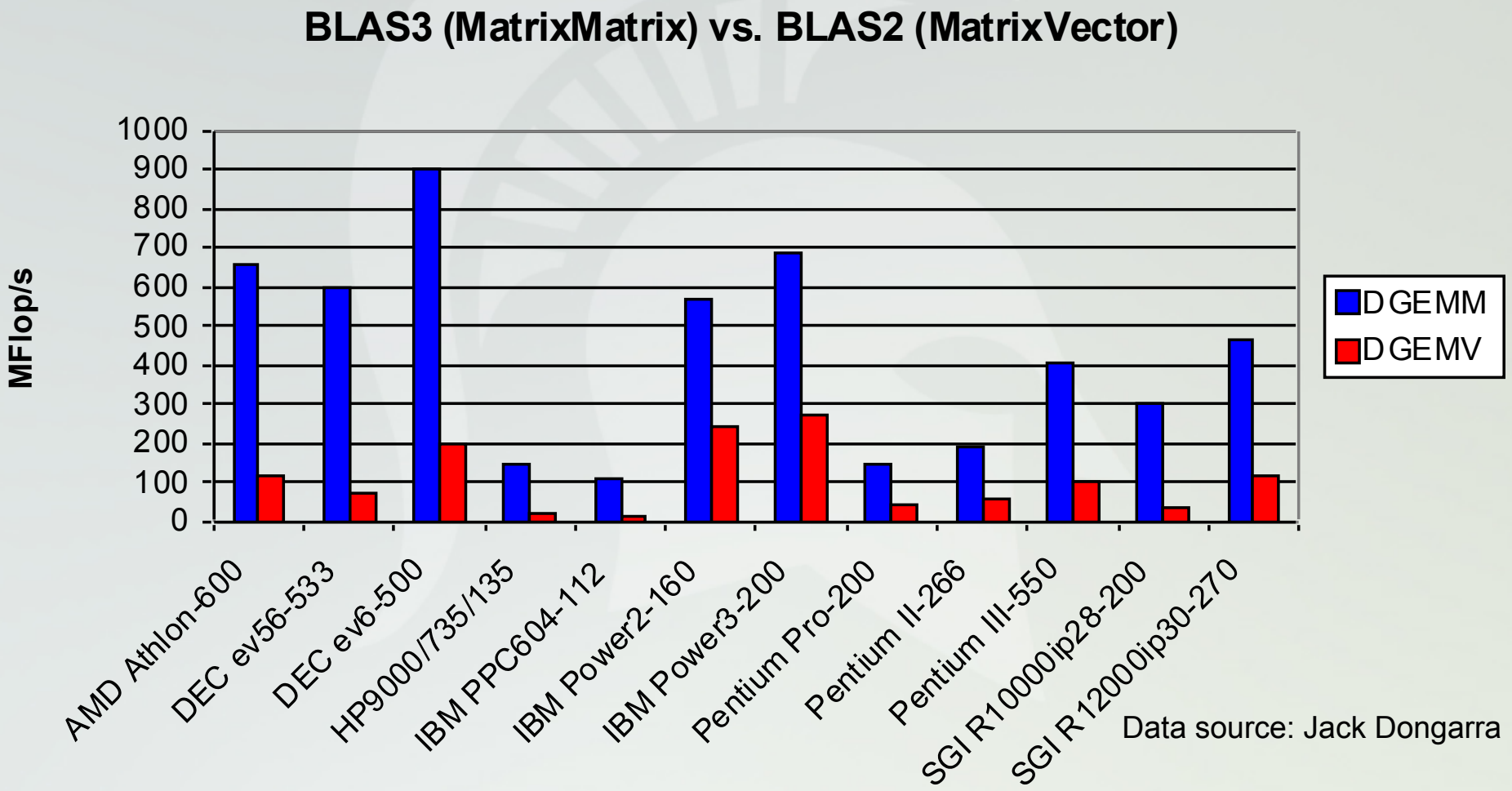
BLAS speeds comparison



Source: Georg Ofenbeck

Different levels of BLAS from
Intel MKL on a Core i7-3930K, single threaded

BLAS2 vs. BLAS3 [6]

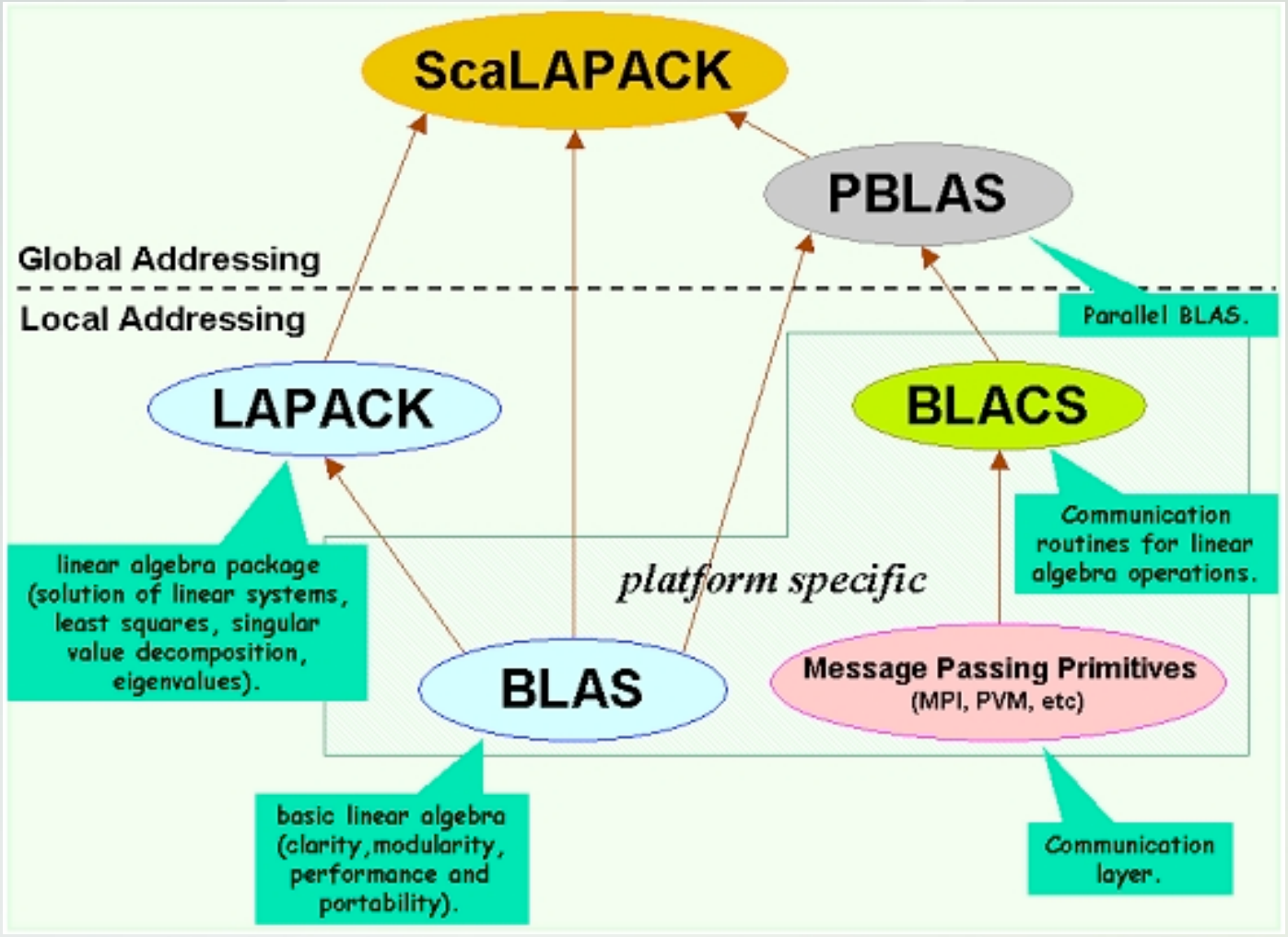


Data source: Jack Dongarra

BLAS Implementations, LAPACK, ScaLAPACK libraries

- **BLAS** is an interface, vendors & others supply optimized implementations
  - Netlib BLAS (reference implementation)
  - ATLAS (Automatically Tuned Linear Algebra Subroutines)
    - Tuned during installation
  - Apple Accelerate, AMD ACKL, Cray libsci, Intel MKL, Goto BLAS, Nvidia cuBLAS, etc.
- **LAPACK** (Linear Algebra PACKage): built on top of BLAS for common linear algebra kernels (to solve systems of linear equations, least squares, eigenvalue and SVD problems)
- **ScaLAPACK:** distributed memory implementation of LAPACK

Dense linear algebra software stack



Source: ACTS Collection, acts.nersc.gov/scalapack

Option 2: Cache-oblivious algorithms [6]

- Need to minimize communication between all levels
  - between L1-L2, L2-L3, cache-DRAM
- The blocked algorithm requires finding a good block size
  - Machine dependent
  - Need to "block" manually
    - 1 level of memory (no cache) ⇒ 3 nested loops (naïve algorithm)
    - 2 levels of memory ⇒ 6 nested loops
    - 3 levels of memory ⇒ 9 nested loops …
- Cache Oblivious Algorithms offer an alternative
  - Treat matrix multiply as a set of smaller (recursive) problems
  - Eventually, blocks will fit in cache
  - Will minimize # words moved between every level of memory hierarchy
  - "Oblivious" to number and sizes of levels

Recursive Matrix Multiplication [6]

- $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A * B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

$$= \begin{pmatrix} A_{11}\cdot B_{11} + A_{12}\cdot B_{21} & A_{11}\cdot B_{12} + A_{12}\cdot B_{22} \\ A_{21}\cdot B_{11} + A_{22}\cdot B_{21} & A_{21}\cdot B_{12} + A_{22}\cdot B_{22} \end{pmatrix}$$

- For simplicity: square matrices with $n = 2^m$
- Extends to general rectangular cases too
- Or can pad with zeros to have $n = 2^m$

Recursive Matrix Multiplication [6]

```
func C = RMM (A, B, n)
    if (n=1) C = A * B
    else {
        C11 = RMM (A11 , B11 , n/2) + RMM (A12 , B21 , n/2)
        C12 = RMM (A11 , B12 , n/2) + RMM (A12 , B22 , n/2)
        C21 = RMM (A21 , B11 , n/2) + RMM (A22 , B21 , n/2)
        C22 = RMM (A21 , B12 , n/2) + RMM (A22 , B22 , n/2) }
    return
```

A(n) = # arithmetic operations in RMM(. , . , n)

= $8 \cdot A(n/2) + 4(n/2)^2$ if (n>1), else 1

= $2n^3$ . . . same operations as usual, in different order

W(n) = # words moved between fast & slow memory by RMM(. , . , n)

= $8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$

= $O(n^3 / (M_{fast})^{1/2} + n^2)$ … same as blocked matrix multiply

Don't need to know M_{fast} for this to work!

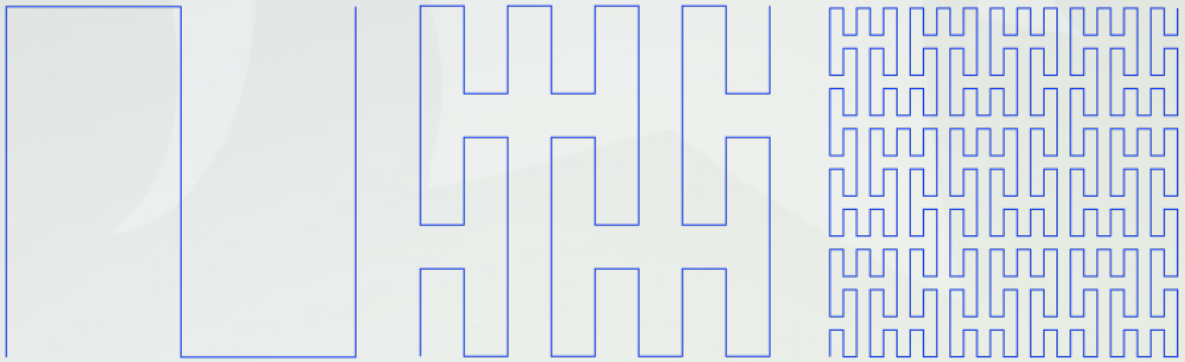
Remarks about Cache Oblivious Algorithms

- Recursion is an elegant way of delegating the task of finding optimal problem size (block size in MatMul) - easy and clean code

- However, achieving good performance is still a challenge
  - Recursion creates deep call stacks in the heap
  - Micro-kernel: Need to cut-off recursion well before reaching the leaf nodes (micro-kernel in MatMul - 1x1 blocks)
  - Careful attention to micro-kernel is still required
  - Recursive partitioning may not conform well with the default data layout - bad data locality

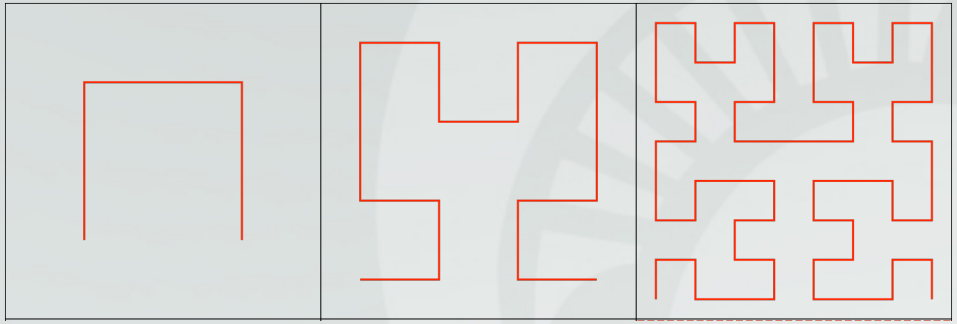
Recursive Data Layouts

- Use a data structure designed for the recursive algorithm
- Different data layout options that would increase data locality in a recursive algorithm are possible
- Space-filling curve: a curve whose range contains the entire 2-dimensional unit square (or more generally an n-dimensional hypercube)

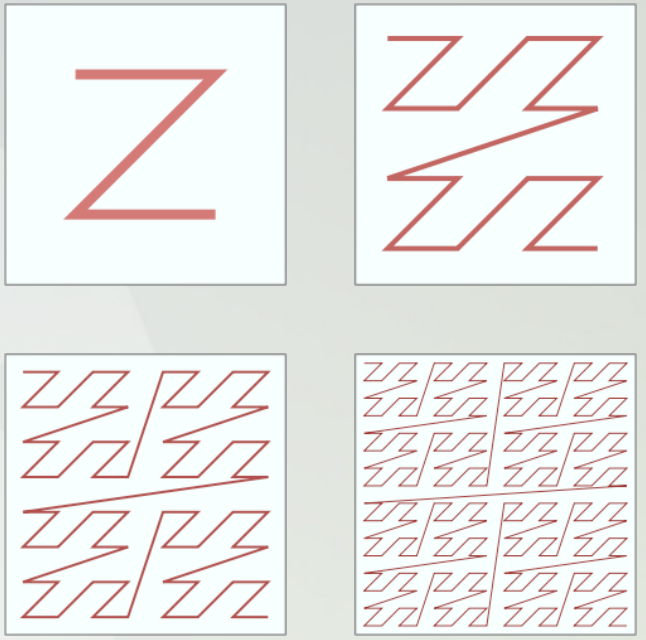


Three iterations of the Peano curve construction. Source: Wikipedia

Recursive Data Layouts for Matrix Multiplication



Hilbert curve construction.
Source: Wikipedia



Morton Z-order
Source: Wikipedia

- Advantages: improved data locality
- Disadvantages: index calculations are expensive - should switch to row/column-major in micro-kernels

Performance, Speedup, Scalability

Speedup

- Number of cores= p
- Serial run-time = T_s or T_1 (run-time on 1 core)
- Parallel run-time = T_p (run-time on p cores)
- **Ideal case**: linear speedup - $T_p = T_1 / p$
- But this is rarely the case (more later)

Latency & Throughput

- Latency: the time it takes to complete an instruction, task or job
  - **Ex**: cycles per instruction, the time needed to do a single BLAST search, execution time of a program
- Throughput: number of instructions, tasks or jobs that can be completed in per unit time.
  - **Ex**: instructions per cycle, the number of BLAST searches that are performed in a minute, Gflop/s

Speedup & Efficiency

- **Speedup** on p cores: $S_p = T_1 / T_p$
- **Efficiency** on p cores: $E_p = S_p / p = T_1 / pT_p$ (*100% optionally)
- Remark 1:
  - S_p is also called **relative speedup**
  - Sometimes there is a more efficient serial algorithm (T_{opt}) that does not parallelize well, any examples?
  - **absolute speedup** = T_{opt} / T_p
  - absolute speedup < relative speedup, but more fair
- Remark 2:
  - For advertisements, use speedup, not efficiency :)

Overheads in Parallel Execution

- For p cores:
  - S_{ideal} = p, E_{ideal} = 1 or 100%
- Ideal speedup is rare.
- Sources of parallel execution overheads:
  - Non-parallelizable serial work (redundant computations)
  - Communication & synchronization
  - Load imbalances
- Also rare, but possible: superlinear speedup!
  - Typically due to having more cache space and better cache utilization

Amdahl's Law

- Captures one aspect of the potential overheads: non-parallelizable serial computations

- $T_1 = W_{ser} + W_{par}$
  - W_{ser}: Time spent on non-parallelizable serial work
  - W_{par}: Time spent on parallelizable work

- $T_p \geq W_{ser} + W_{par} / p$

Amdahl's Law

- **Amdahl's Law:** $S_P \leq \dfrac{W_{ser} + W_{par}}{W_{ser} + W_{par}/P}$

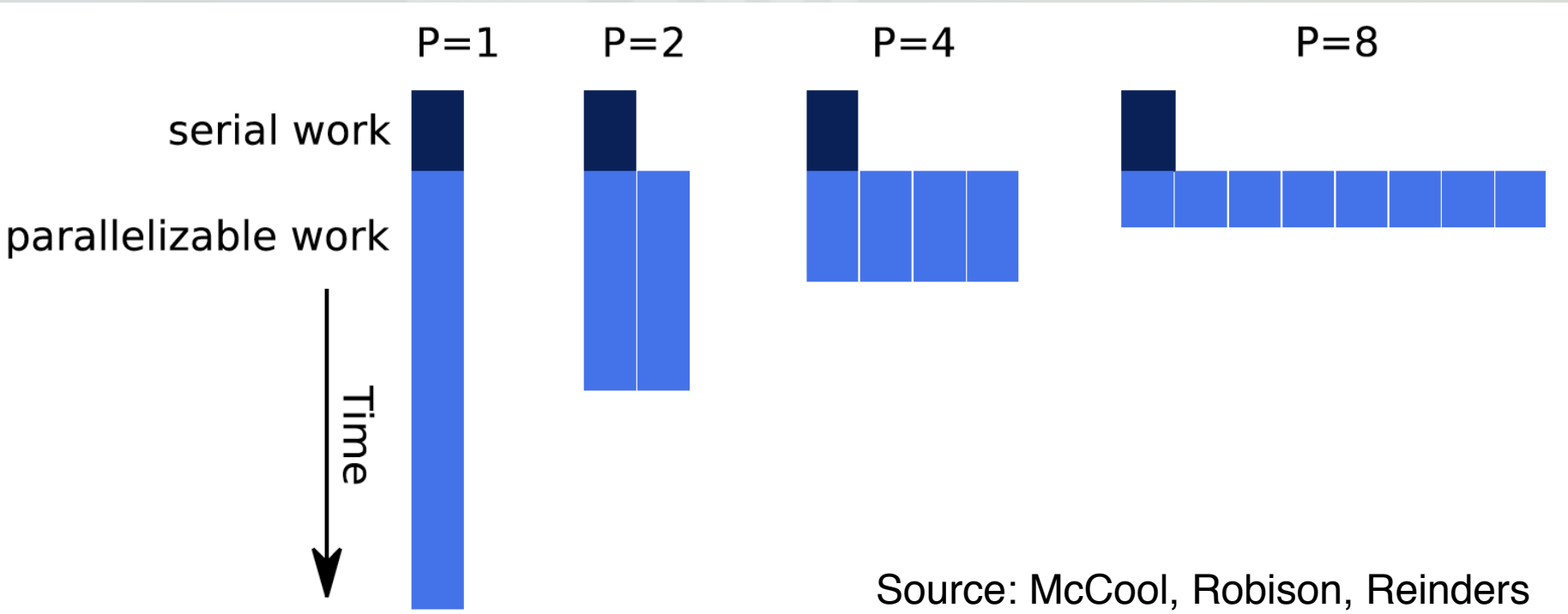
- or let *f* be the serial fraction of the total work,

$$\begin{aligned} W_{ser} &= fT_1 \\ W_{par} &= (1-f)T_1 \\ S_p &\leq \frac{1}{f + (1-f)/P} \end{aligned}$$

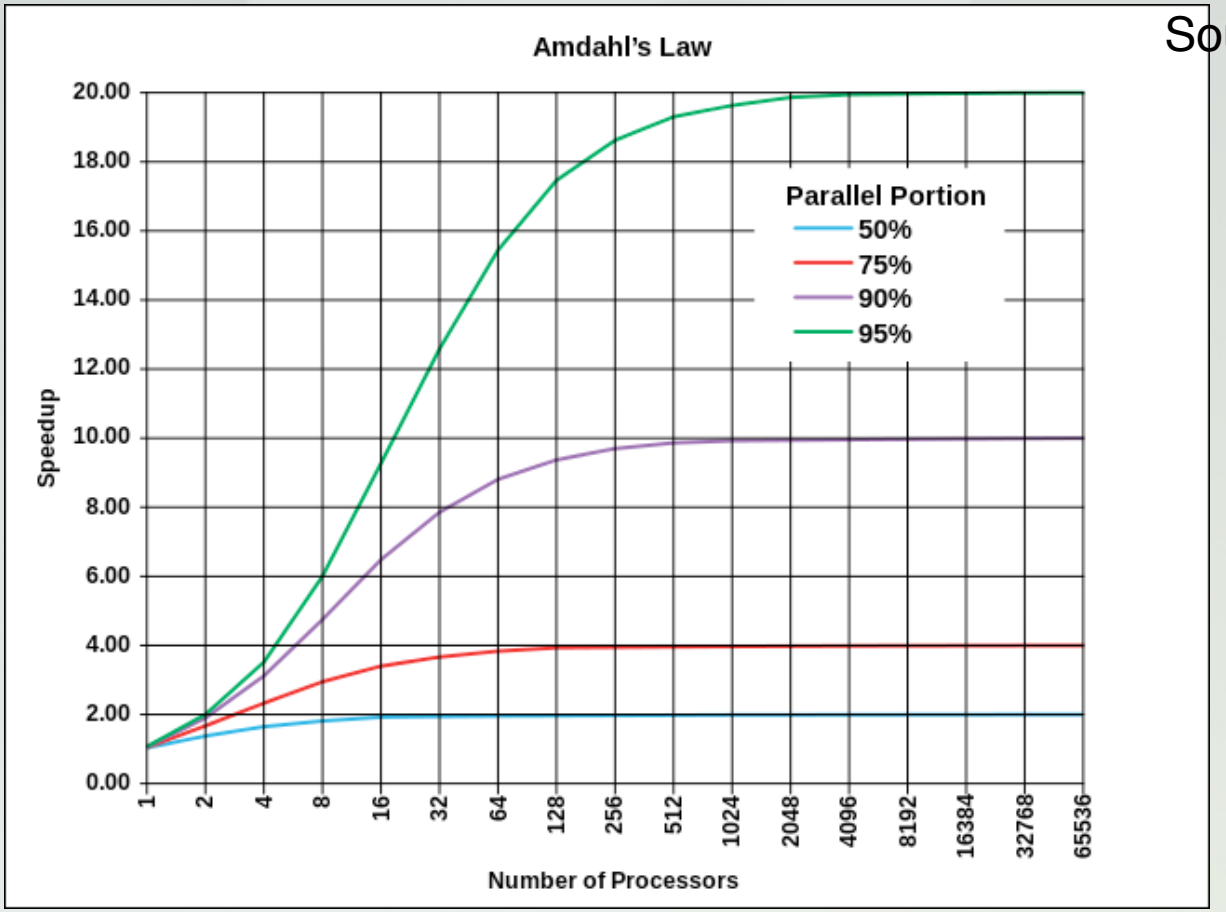
- An important corollary: Even when there are infinite resources, the speedup can not be greater than 1 / *f* !

$$S_\infty \leq \frac{1}{f}$$

Amdahl's Law pictorially



Amdahl's Law asymptotically



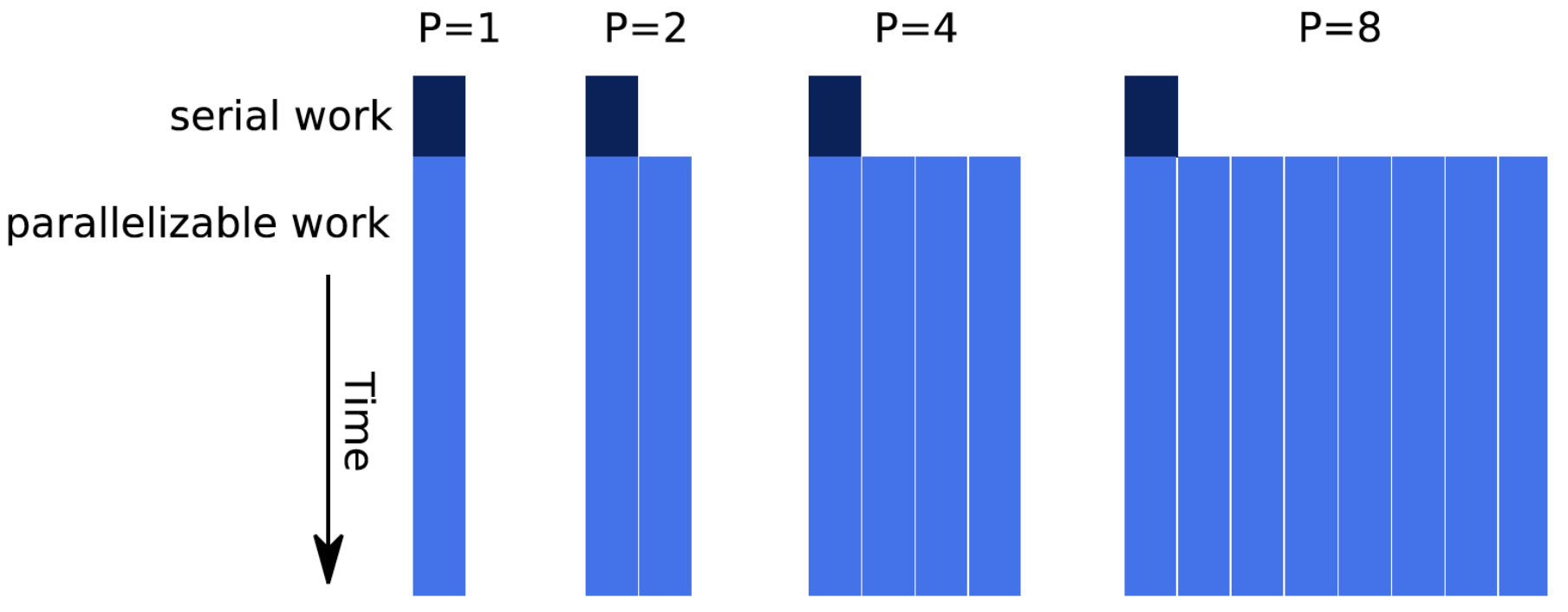
Source: Wikipedia

How about efficiency?

Amdahl's Law & Strong scalability

- Amdahl's law is closely related to a commonly used notion of scalability: strong scaling
- **Strong scaling:** how does the solution time vary as we increase the number of processors for a fixed total problem size?
- But, the problem sizes that we want to solve often grow as the machines that we work with grow, too!
- **Weak scaling:** how does the solution time vary as we increase the number of processors and the total problem size at the same rate? (attributed to Gustafson & Barsis)

Weak scaling (Gustafson-Barsis)



Source: McCool, Robison, Reinders

Work-span model

- Both Amdahl's Law and Gustafson's Law are somewhat optimistic
- They assume the parallelizable parts can be run in parallel without any limits
- In practice, there are limits
- Work-span model tries to capture the speedup and efficiency using a more realistic approach based on a DAG representation