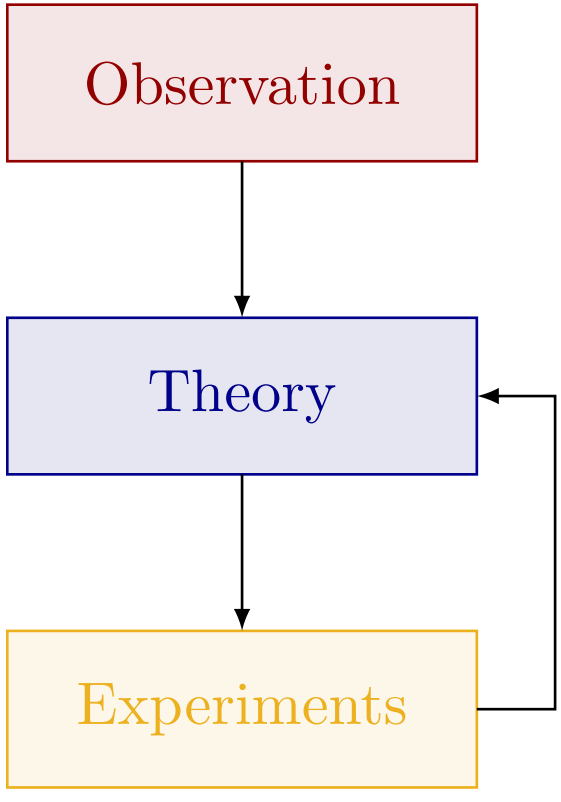
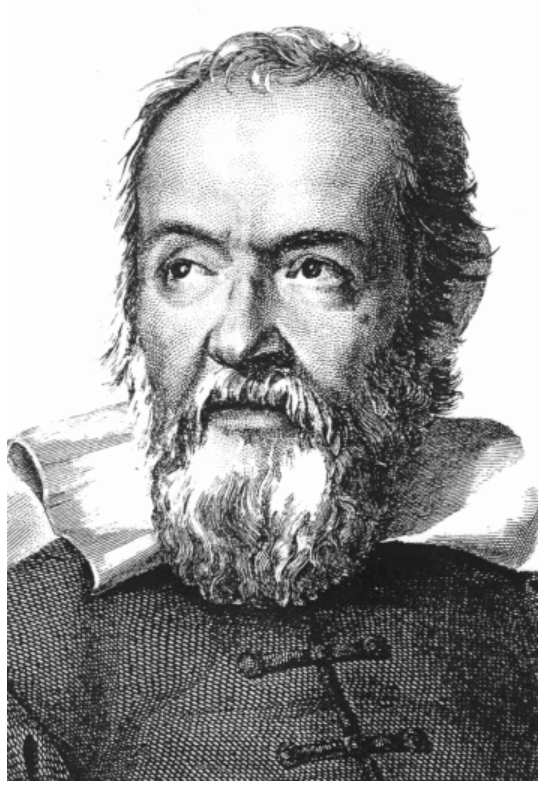
High Performance Computing Algorithms

Outline

Simulation: the third pillar of science





The traditional scientific method relies on two pillars

1. Based on observations, formulate a theory
2. conduct experiments to validate theory and iterate if necessary

Simulation: the third pillar of science

Limitations of the classic scientific method:

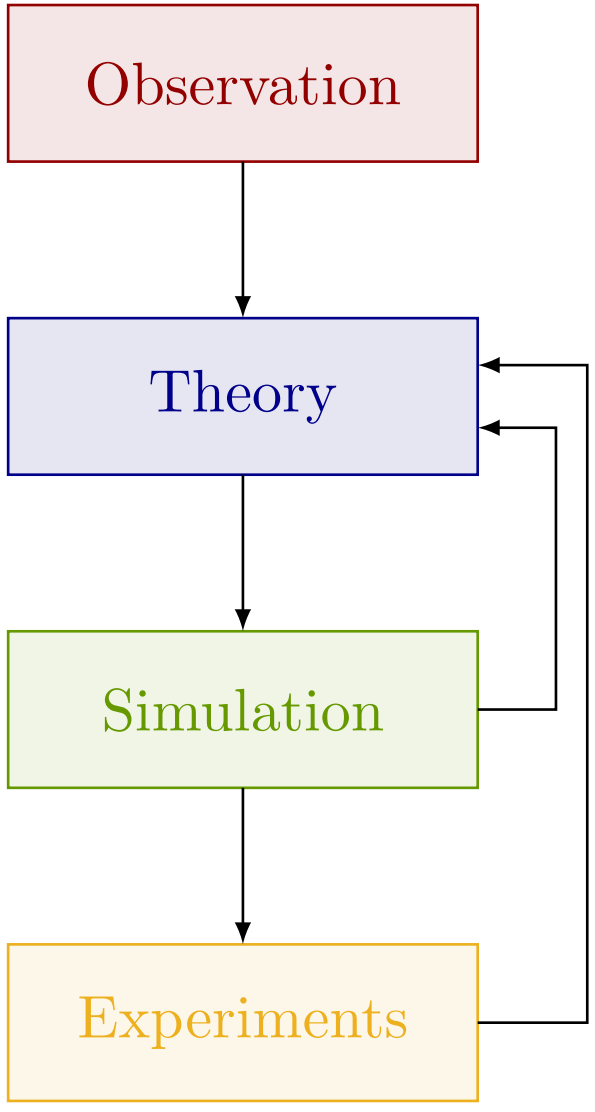
- Expensive
- Slow
- Dangerous
- Intrusive

Simulation: the third pillar of science

Limitations of the classic scientific method:

- Expensive
- Slow
- Dangerous
- Intrusive

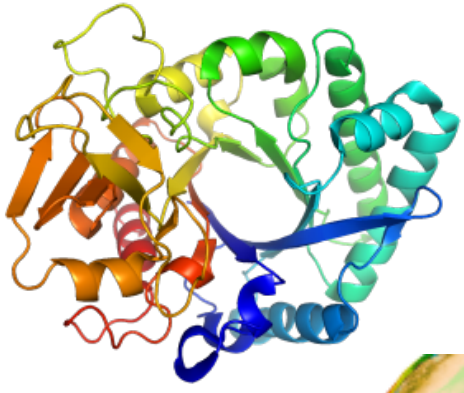
Simulation is universally recognized as the third pillar of science.

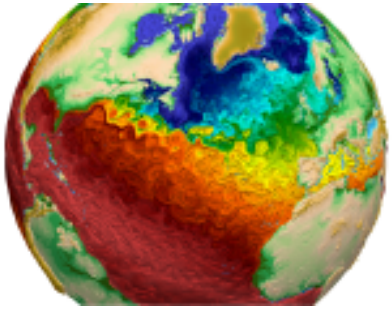


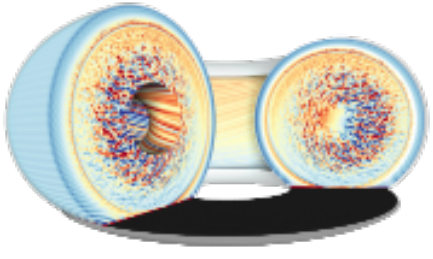
Simulation: the third pillar of science

Example of applications/disciplines that heavily rely on simulation:

- Life science and health
  - genome processing and sequencing
  - drugs
  - protein folding
- Geo-science
  - weather modeling (climate change)
  - geophysics (earthquakes, oil reservoirs)
- Transport and space
  - aircraft design
  - autonomous vehicle
- Astrophysics
- Energy

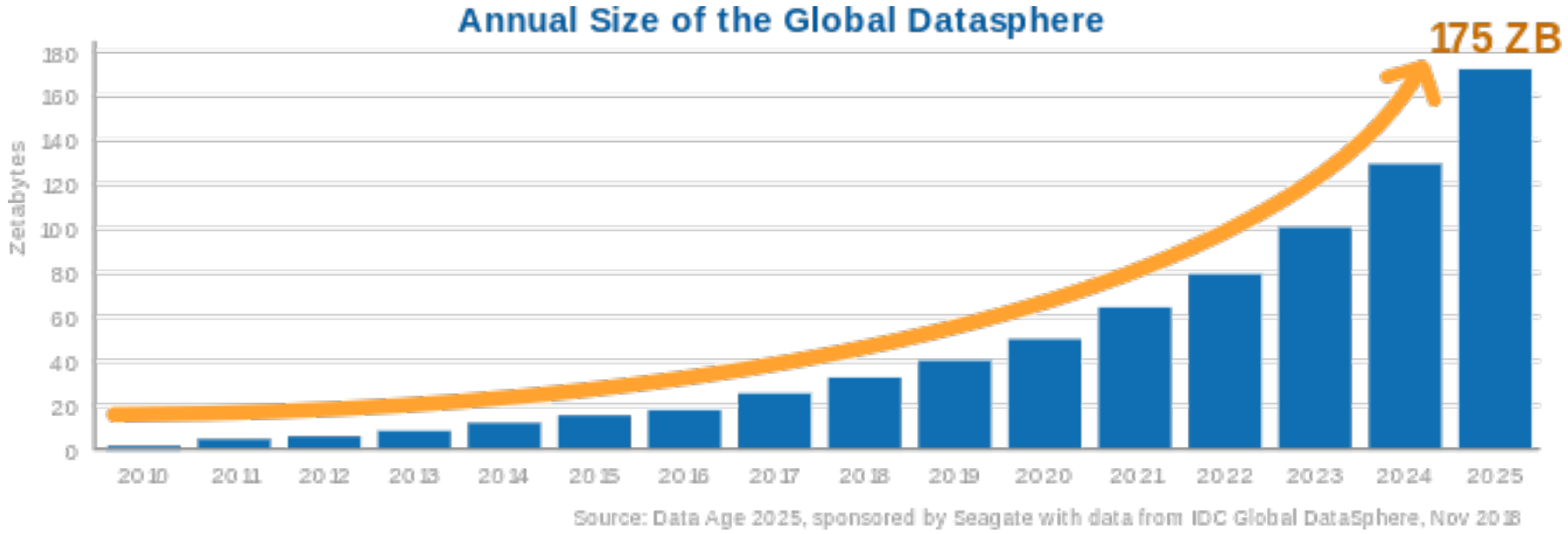






Images from:Wikipedia, Columbia Climate School, Dassault systèmes, Maison de la Simulation

High Performance Data Analytics



The volume of produced data is growing exponentially. Where does data come from:

- Sensors (telescopes, IoT, etc.)
- Experiments
- Simulations
- Social networks
- Transactions

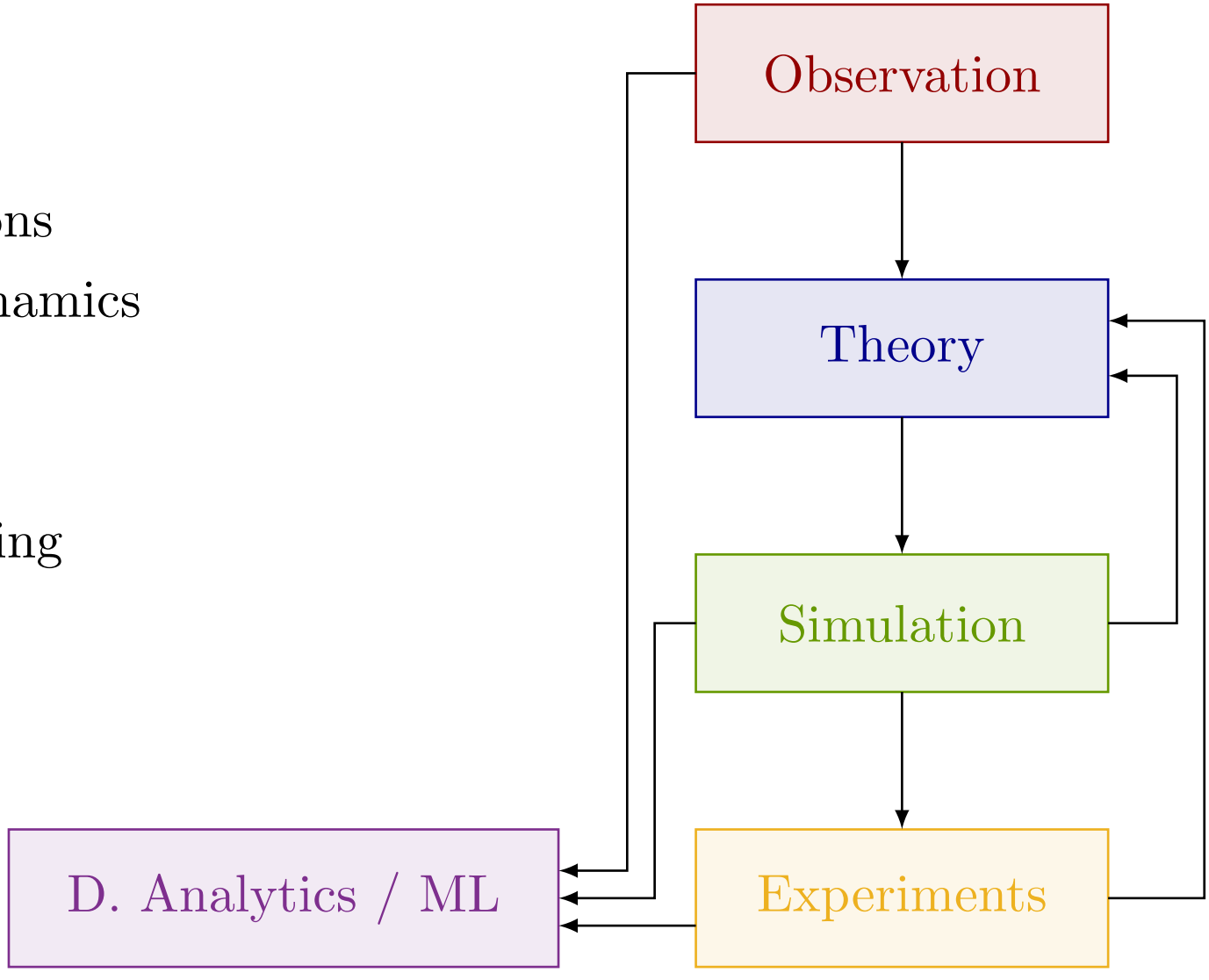
Most of this data is unstructured

Extracting information out of this data is the objective of High Performance Data Analytics

High Performance Data Analytics

HPDA applications

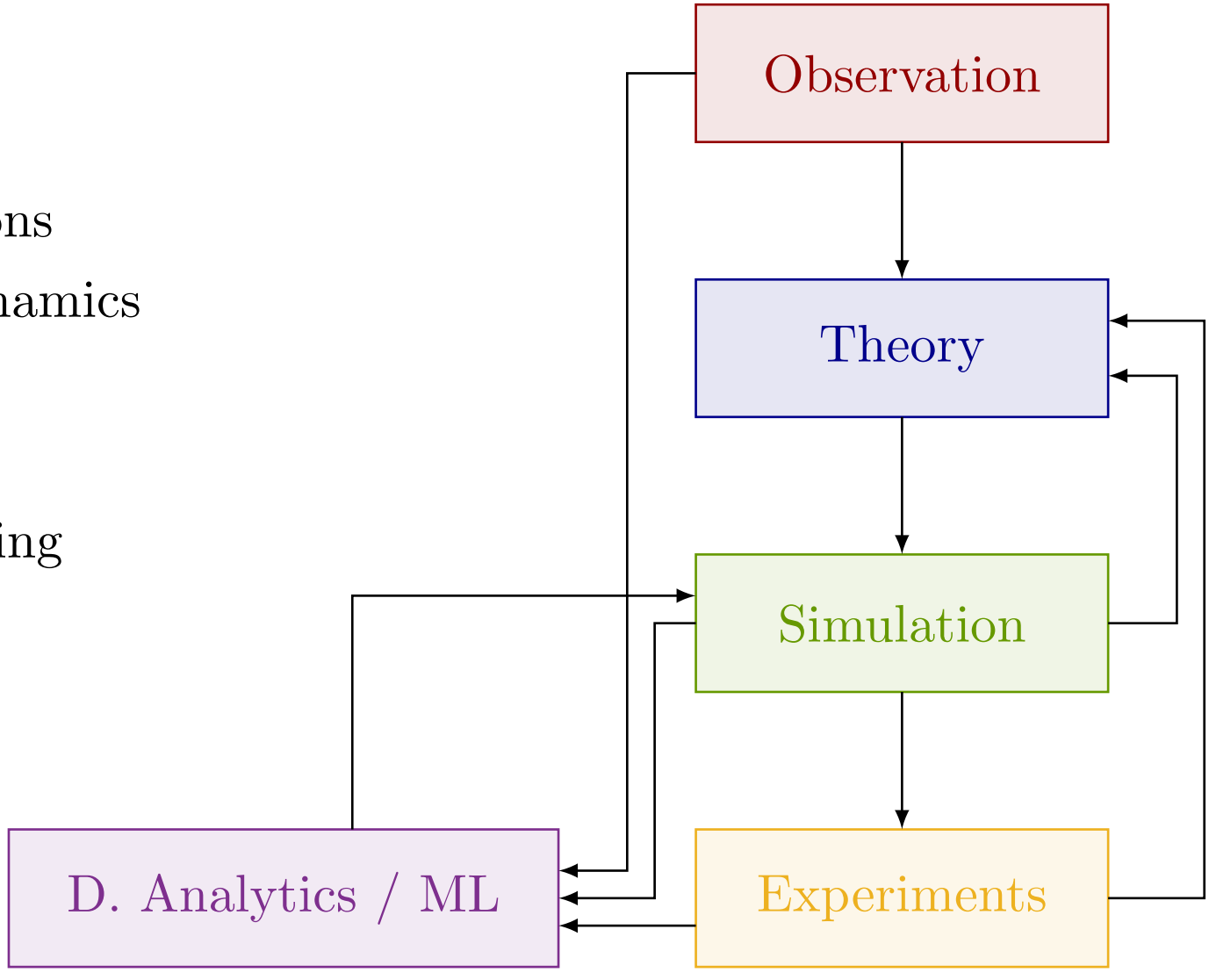
- Molecular Dynamics
- Astrophysics
- Banking
- Image processing



High Performance Data Analytics

HPDA applications

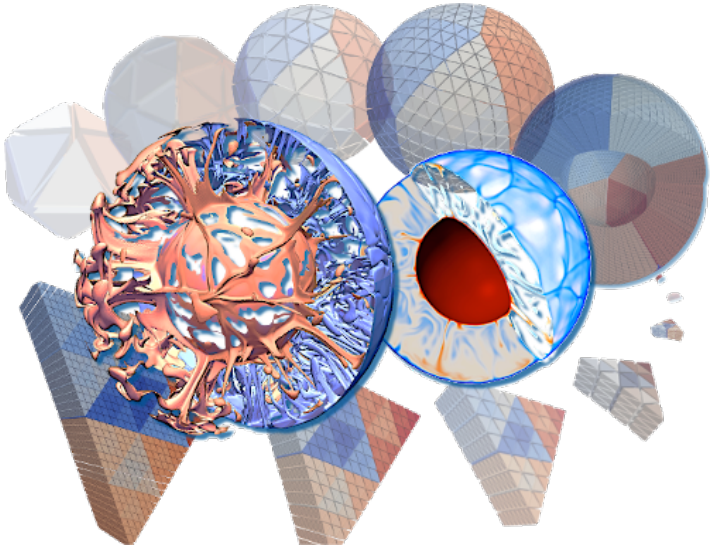
- Molecular Dynamics
- Astrophysics
- Banking
- Image processing
- Simulation



Simulation: example

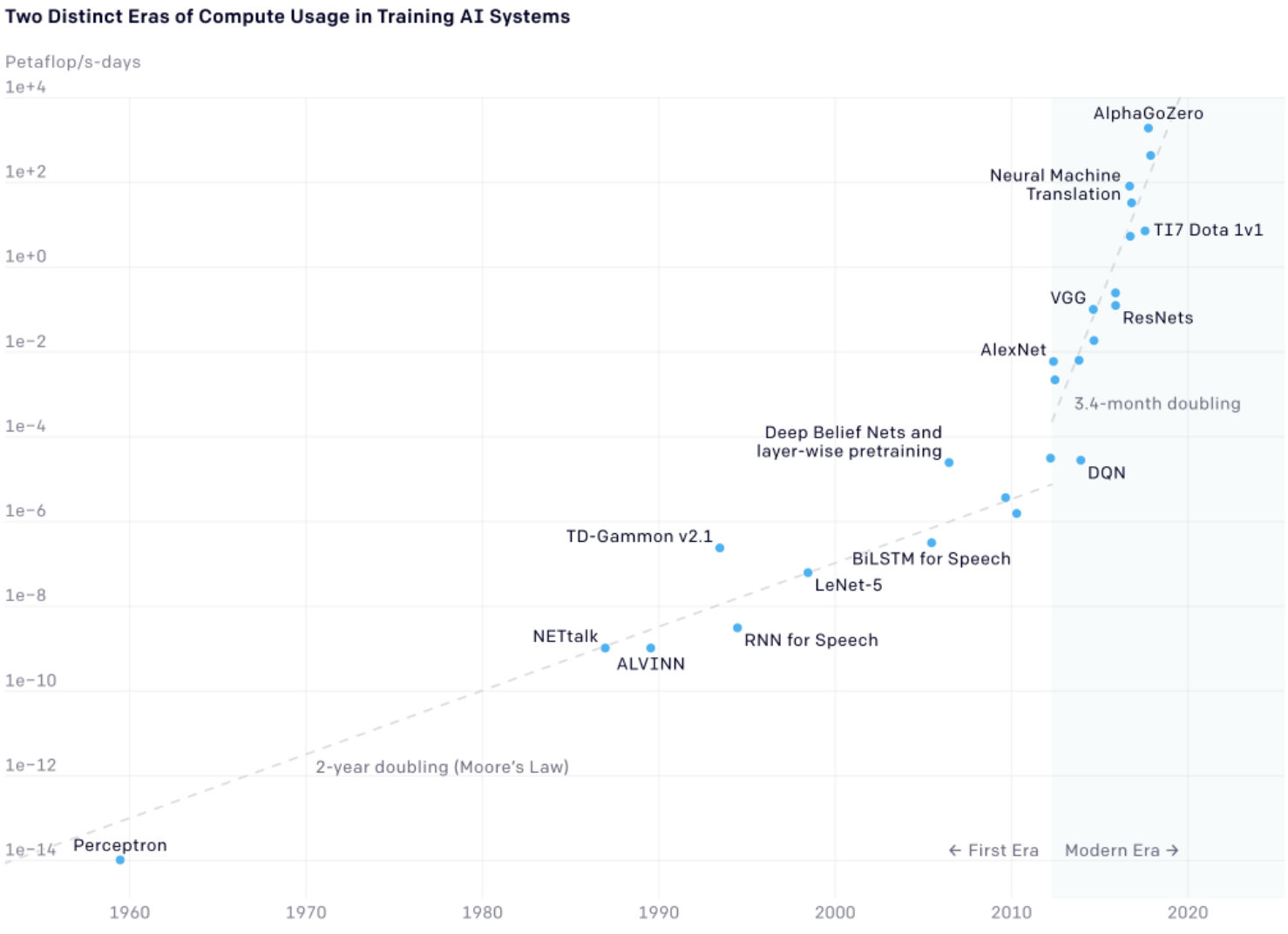
Example, mantle convection problem (courtesy of Ulrich Rüde [3]):

$$
\begin{aligned}
-\mathrm{div}\left(\frac{\nu}{2}(\nabla\mathbf{u} + (\nabla\mathbf{u})^\top)\right) + \nabla\mathbf{p} &= \mathbf{f} \quad \text{in } \Omega, \\
\mathrm{div}(\mathbf{u}) &= 0 \quad \text{in } \Omega, \\
\mathbf{u} &= \mathbf{g} \quad \text{on } \partial\Omega,
\end{aligned}
$$



This application requires solving a linear system with up to 10^{12} unknowns. For running simulations large supercomputers with up to 50K cores were used.

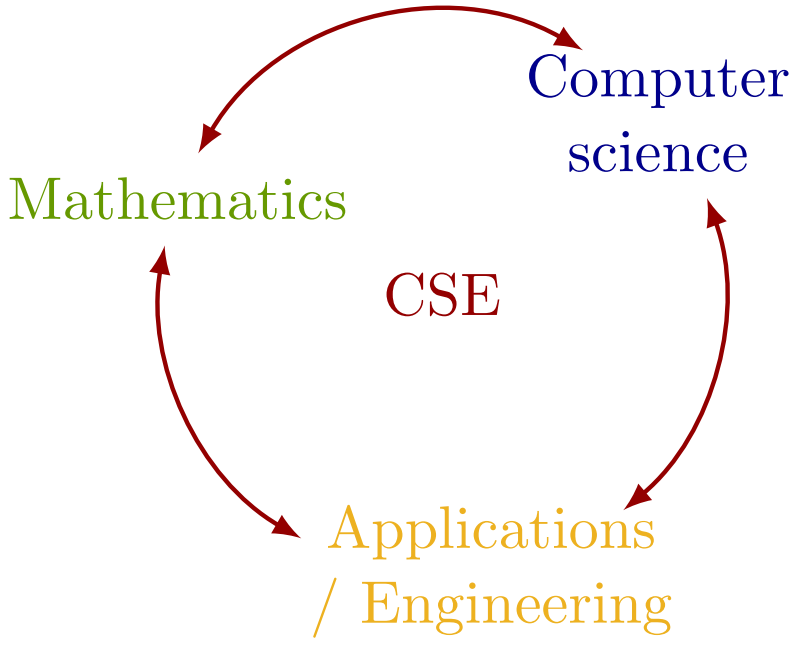
Data analytics/ML: example



From `https://openai.com/blog/ai-and-compute/`

Computational science

Computational Science and Engineering is a discipline that exploits the power of computation for solving complex problems for numerous applications of science and engineering



- develop accurate and robust algorithms that have low operational complexity and storage and good scalability
- implement these algorithms efficiently on high performance, parallel supercomputers

How to measure performance

How is performance of a numerical algorithm computed?

Number-crunching applications (scientific computing, IA,...) are mostly based on floating point operations

	Signif. bits	Exp. bits	Range	Unit roundoff u
fp128	113	15	$10^{\pm 4932}$	1×10^{-34}
fp64	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	24	8	$10^{\pm 38}$	6×10^{-8}
fp16	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16	8	8	$10^{\pm 38}$	4×10^{-3}

Speed: Floating point operations per second

How to measure performance

How well does a code/algorithm use the available computing power?

$$E = \frac{\text{Speed}}{\text{peak performance}}$$

The peak performance of a processor is the maximum speed at which it can execute floating-point operations.
How to compute the peak performance of a processor?

$$\#\text{ cores} \times \text{freq.} \times (\text{ops per clock})$$

Example: Intel Xeon Gold 6140 Processor in fp64

$$18 \times 2.3\text{ GHz} \times (2 \times 2 \times 8) \simeq 1.3\text{ TFlop/s}$$

The first 2 is because the are two ALUs (Arithmetic Logical Unit). The second 2 is because of Fused Multiply-Add (FMA). The 8 is because of AVX-512 vector units (it becomes 16 for fp32).

How to measure performance

Is an efficiency of 1 attainable? No

- Codes do more than just floating-point operations (integer, logic, etc.)
- Not all algorithms can take advantage of FMA or vectorization
- Memory access: data has to be transferred to the processor to be computed upon
- Communications: in a parallel systems data must be moved around
- In a parallel setting, some parts of the algorithm may not be (efficiently) parallelized (see the Amdahl's law later)

How to measure performance

HPL (High Performance Linpack) is a refence benchmark for measuring the performance of supercomputers
Top500 06/2021 top entries (`http://top500.org`):

	Name	Constructor	Country	# cores	Rmax.	Rpeak	KW
1	Fugaku	Fujitsu	Japan	7,630,848	442	537	29,899.23
2	Summit	IBM	United States	2,414,592	148	200	10,096.00
3	Sierra	IBM	United States	1,572,480	94	125	7,438.28
4	Sunway	NRCPC	China	10,649,600	93	125	15,371.00
5	Perlmutter	HPE	United States	706,304	64	89	2,528.00

Is HPL always relevant? No.
Other lists/benchmarks:

- HPCG (High Performance Conjugate Gradient)
- Graph500
- Green500

Evolution of supercomputers

The evolution of supercomputers was ruled by two laws:

- **Dennard's scaling (1974)**: we can scale the transistor's feature size and voltage by a factor $1/\lambda$, increase the frequency by a factor λ and keep the power density constant:

$$P_n = QCV^2 f, \qquad P_{n+1} = (Q\lambda^2)\left(\frac{C}{\lambda}\right)\left(\frac{V^2}{\lambda^2}\right)(f\lambda) = P_n$$

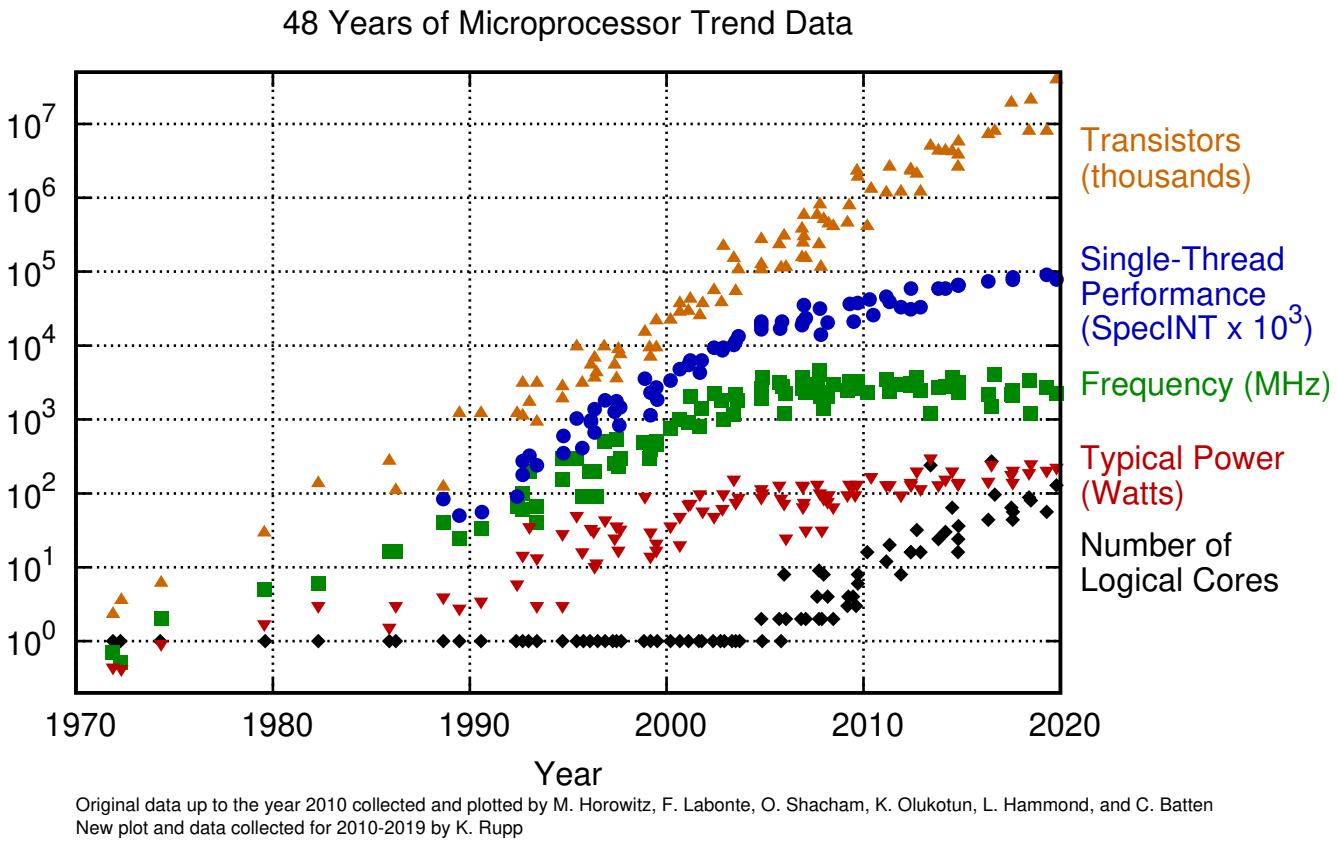
  At each new generation we have processors that have more transistors, are faster and consume the same power!!!

- **Moore's law (1965)**: empiric observation that the number of transistors doubles every 2 years (i.e., $\lambda = \sqrt{2}$)

Evolution of supercomputers

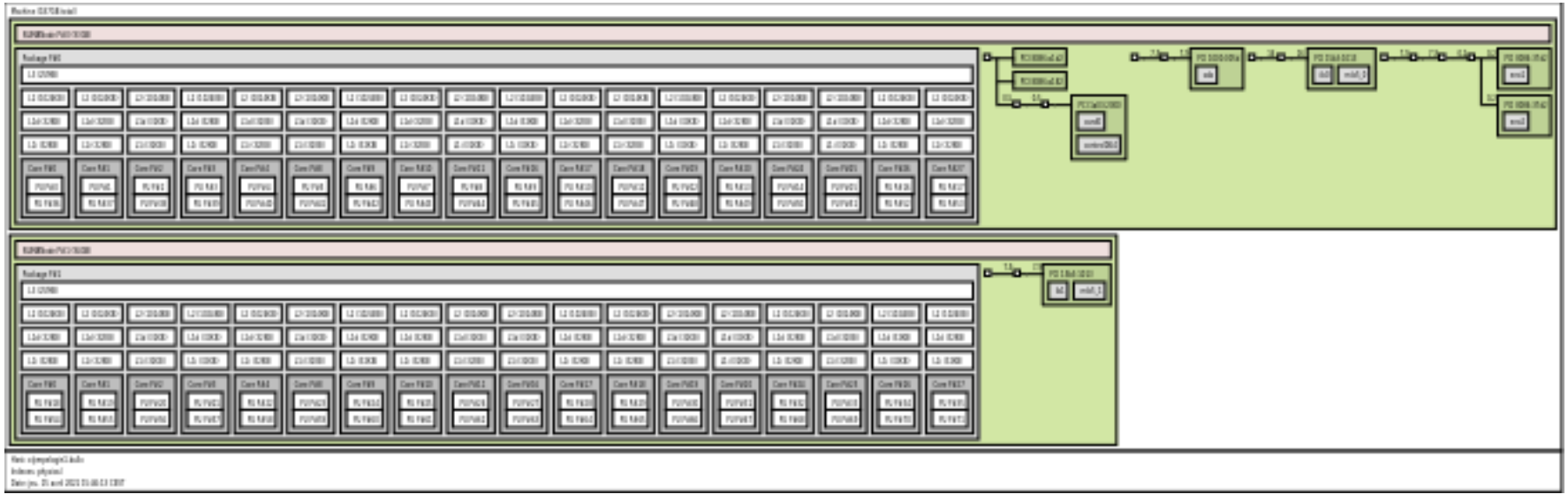
Around 2005: breakdown of Dennard's scaling. Below 65nm transistor size leakeage current (which was negligible at Dennard's time) become prohibitive → can't lower the voltage → can't increase the frequency. This is called the power wall.

Moore's law still holds though: what to do with the extra transistors? Multicores!



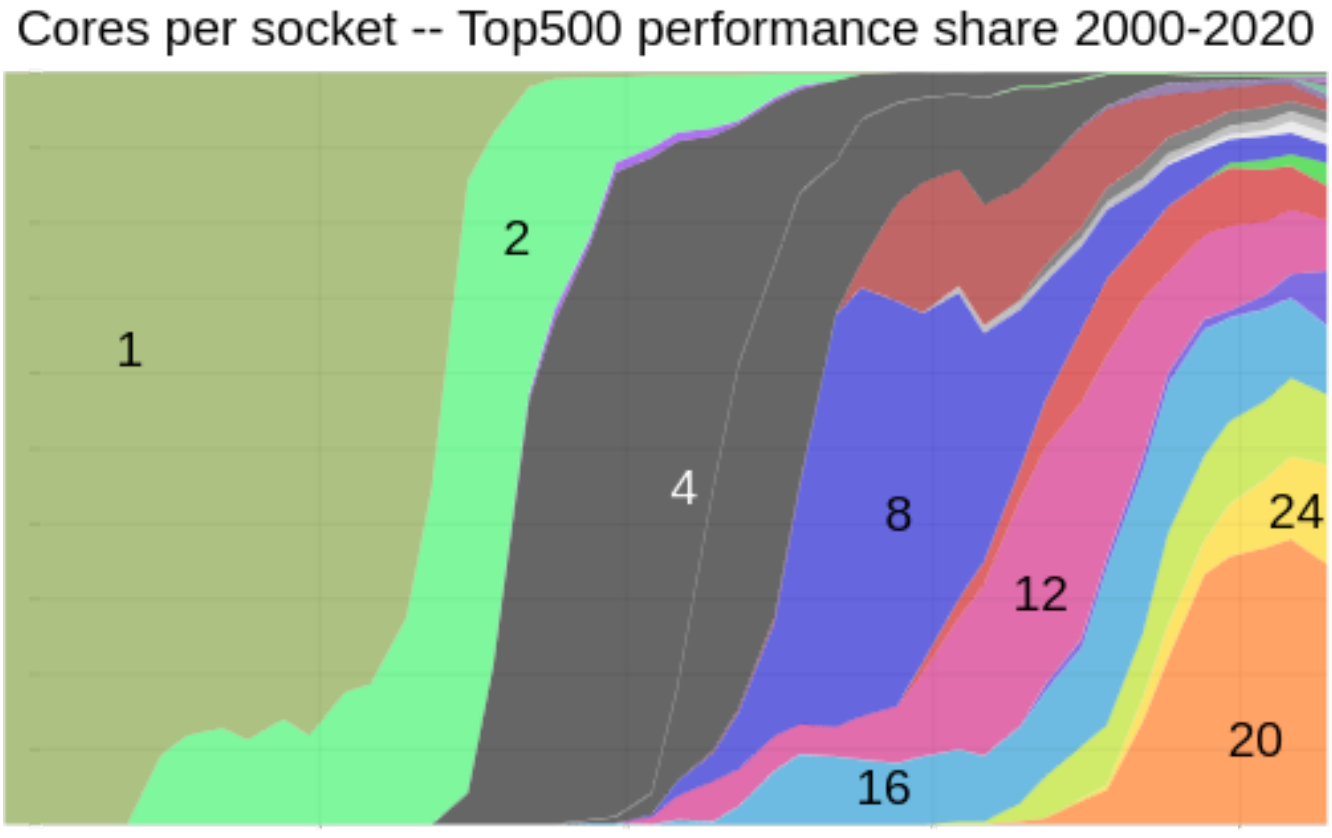
Evolution of supercomputers

Example of a multicore architecture: Intel Xeon Gold 6140 (two of them)



Evolution of supercomputers

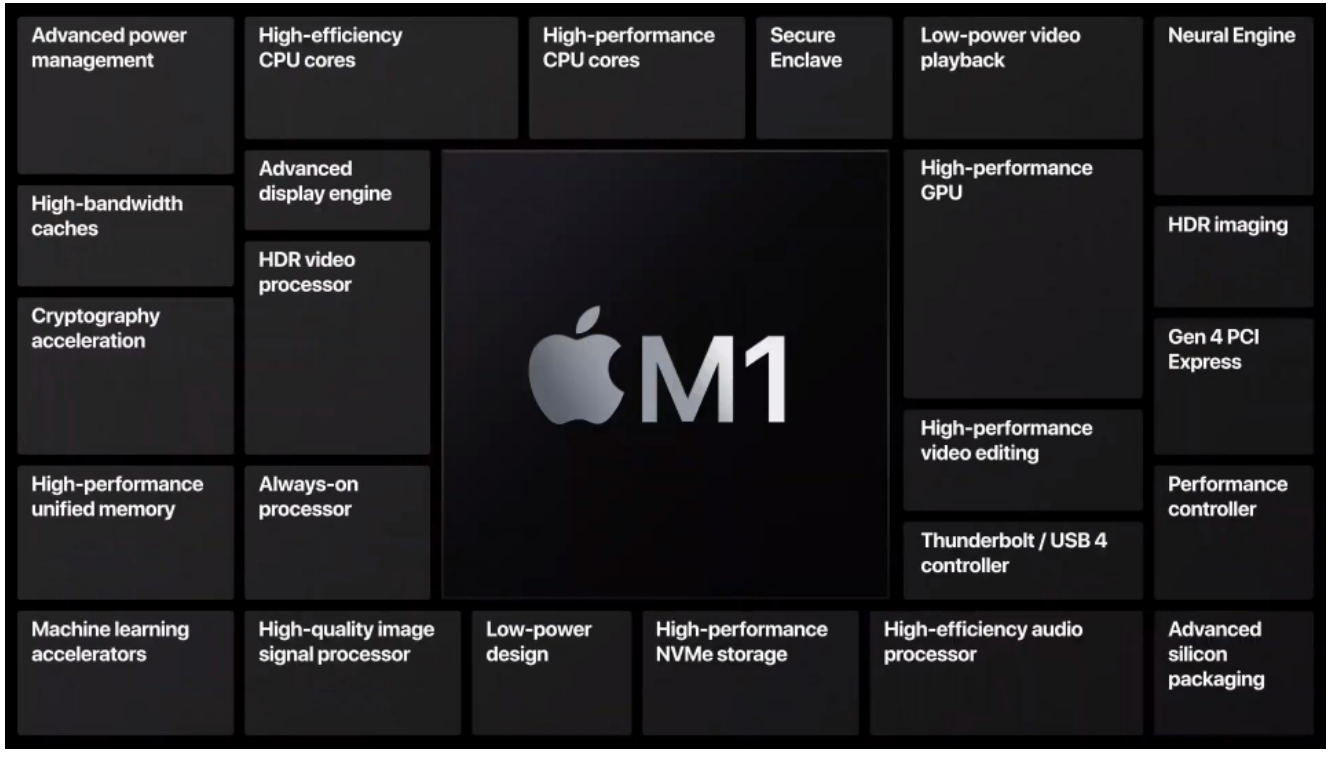
All today's supercomputers have multicore processors.



Evolution of supercomputers

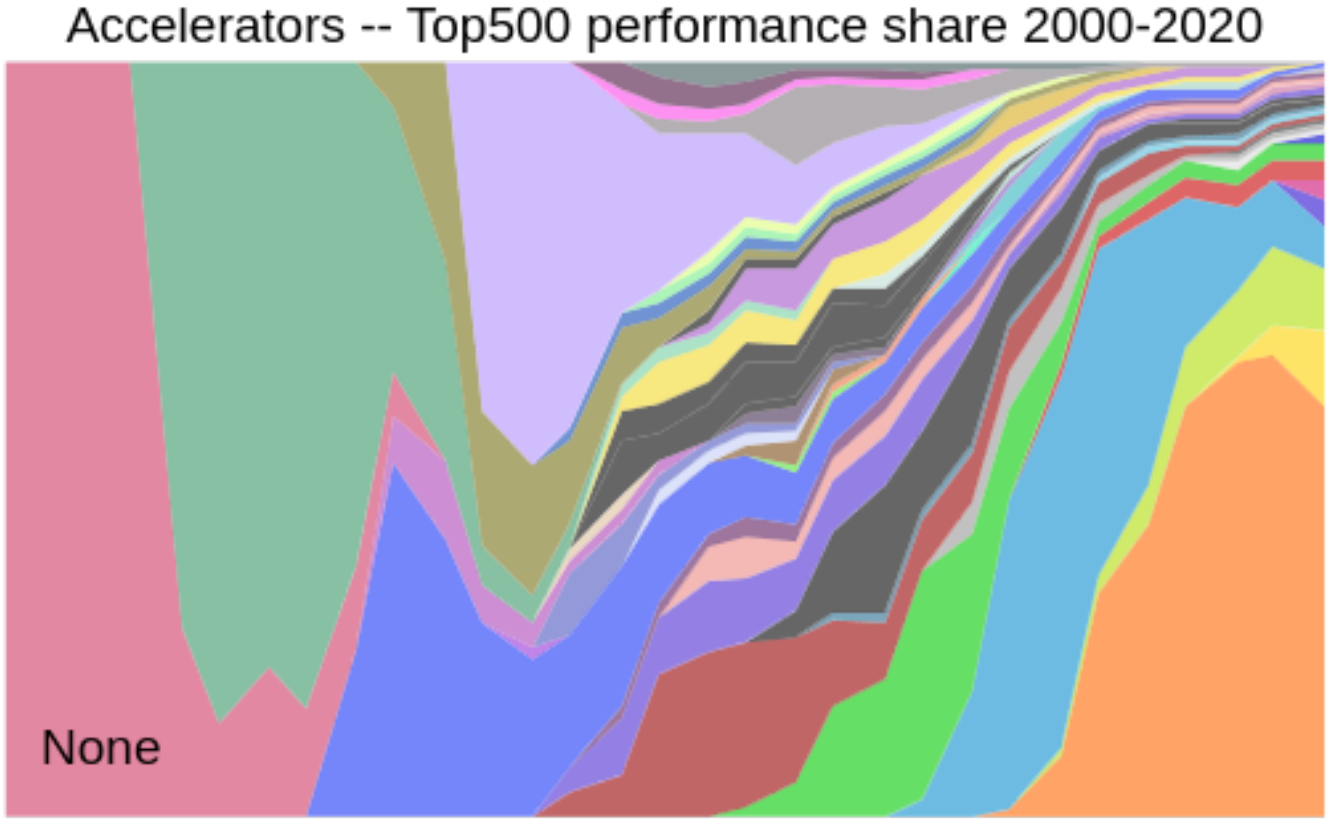
Moore's law is breaking down too.
We can't densify processors → use available transistors more efficiently through specialization (accelerators, GPUs, low-precision units, etc.)

Example: Apple M1 processor



Evolution of supercomputers

Most of today's supercomputers are equipped with accelerators or have specialized units



Evolution of supercomputers

What does a modern supercomputer look like?
IBM Summit (Oak Ridge National Lab., USA)

Summit Overview

OpenPOWER

Compute System

10.2 PB Total Memory
256 compute racks
4,608 compute nodes
Mellanox EDR IB fabric
200 PFLOPS
~13 MW

Compute Rack

18 Compute Servers
Warm water (70°F direct-cooled components)
RDHX for air-cooled components

39.7 TB Memory/rack
55 KW max power/rack

Compute Node

2 x POWER9
6 x NVIDIA GV100
NVMe-compatible PCIe 1600 GB SSD

25 GB/s EDR IB- (2 ports)
512 GB DRAM- (DDR4)
96 GB HBM- (3D Stacked)
Coherent Shared Memory

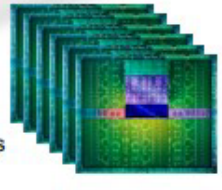
Components

IBM POWER9
- 22 Cores
- 4 Threads/core
- NVLink

GPFS File System

250 PB storage
2.5 TB/s read, 2.5 TB/s write

NVIDIA GV100
- 7 TF
- 16 GB @ 0.9 TB/s
- NVLink

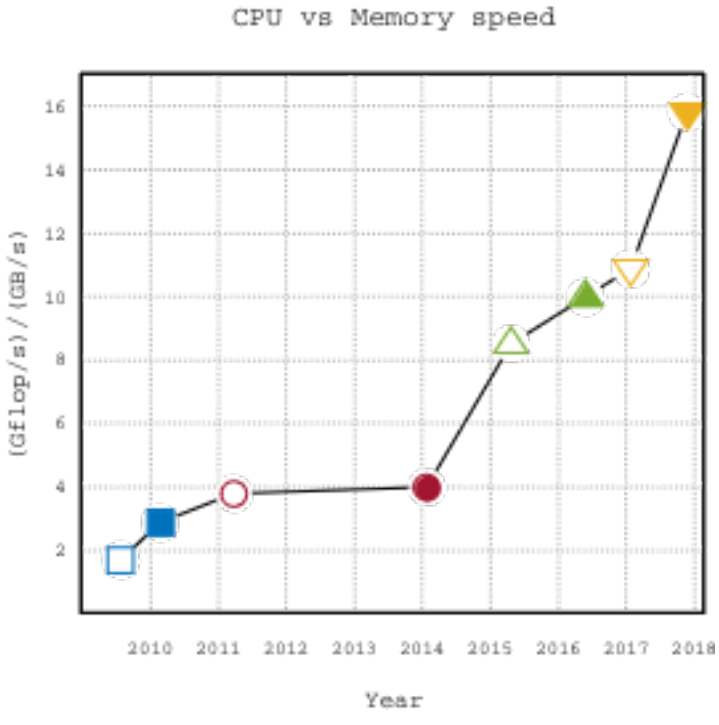


OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

Outline

Memory challenge for sequential performance

One of the most challenging problems to address to achieve high performance in sequential algorithm/codes is related to data access and to the performance gap between processing units and main memory



This gap has lead to the introduction of cache memories

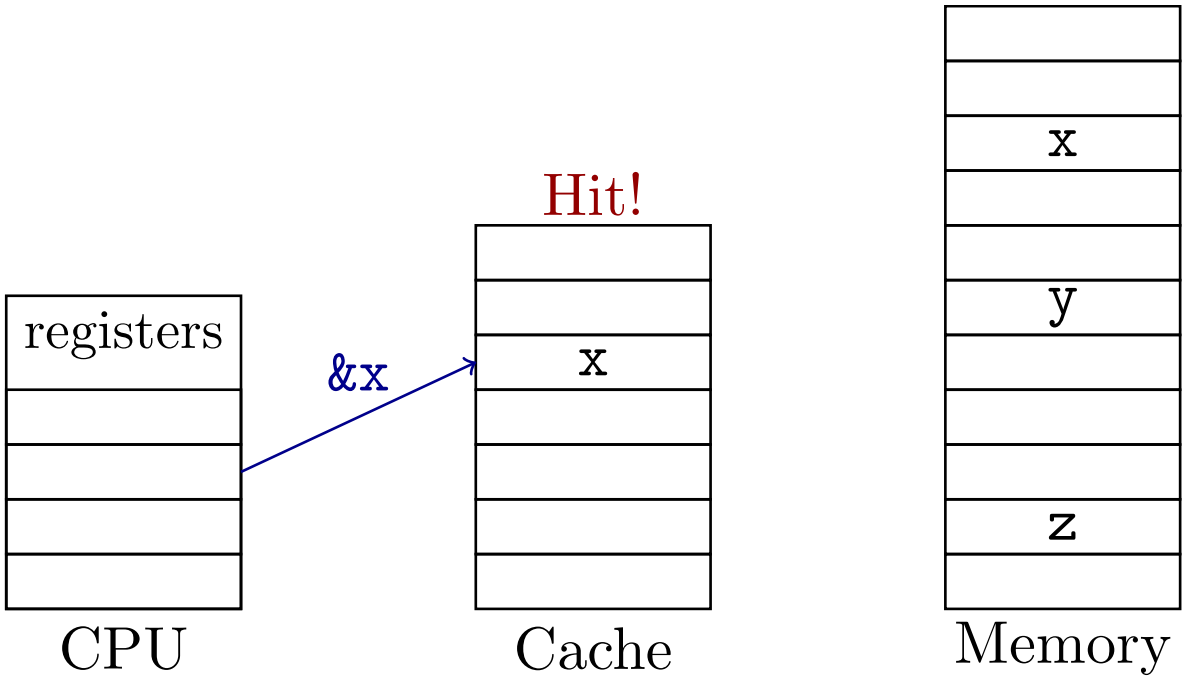
Cache memories

Cache memories are motivated by the observation that most software spend most of their run time in a relatively small portion of the code. The same happens for data:

- temporal locality: repeated access to the same data in a short period
- spatial locality: if some data is accesses at some time, the probablity that nearby data is accessed in a short period is very high

Caches are smaller, faster memories that are closer to the processor and are meant to store frequently accessed data.

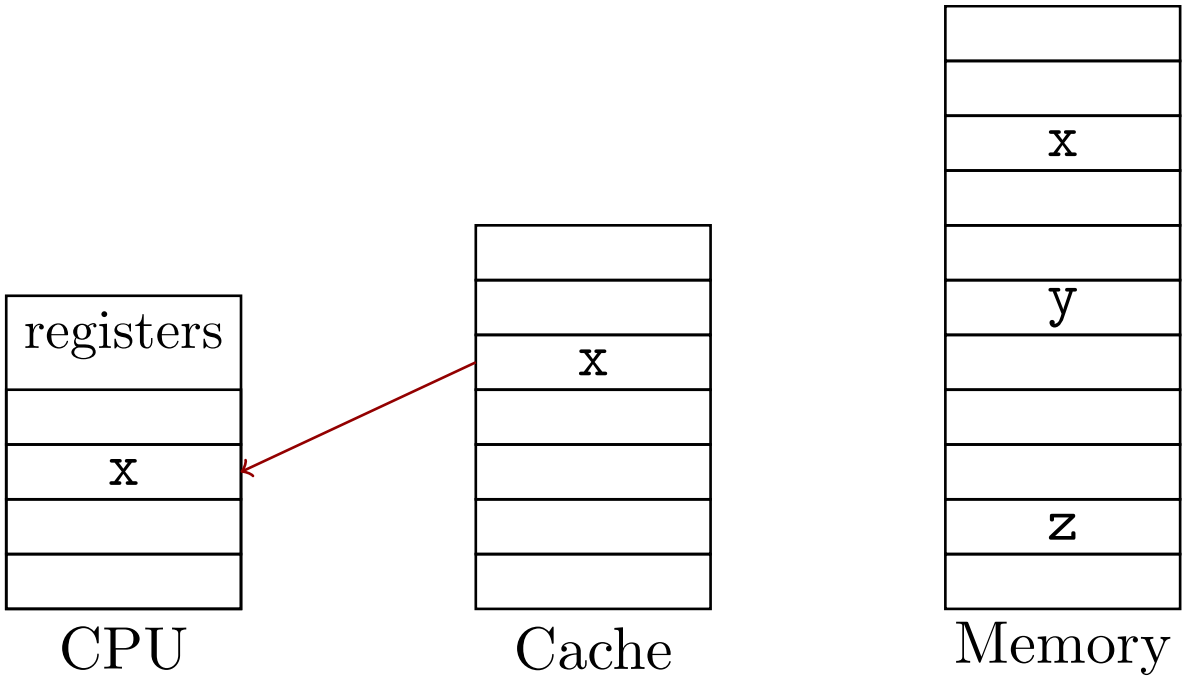
Cache memories



Data must be brought into a register to be used in an instruction. The CPU first looks for the data in the cache

- if the data is found we have a cache hit: the data is copied in the register

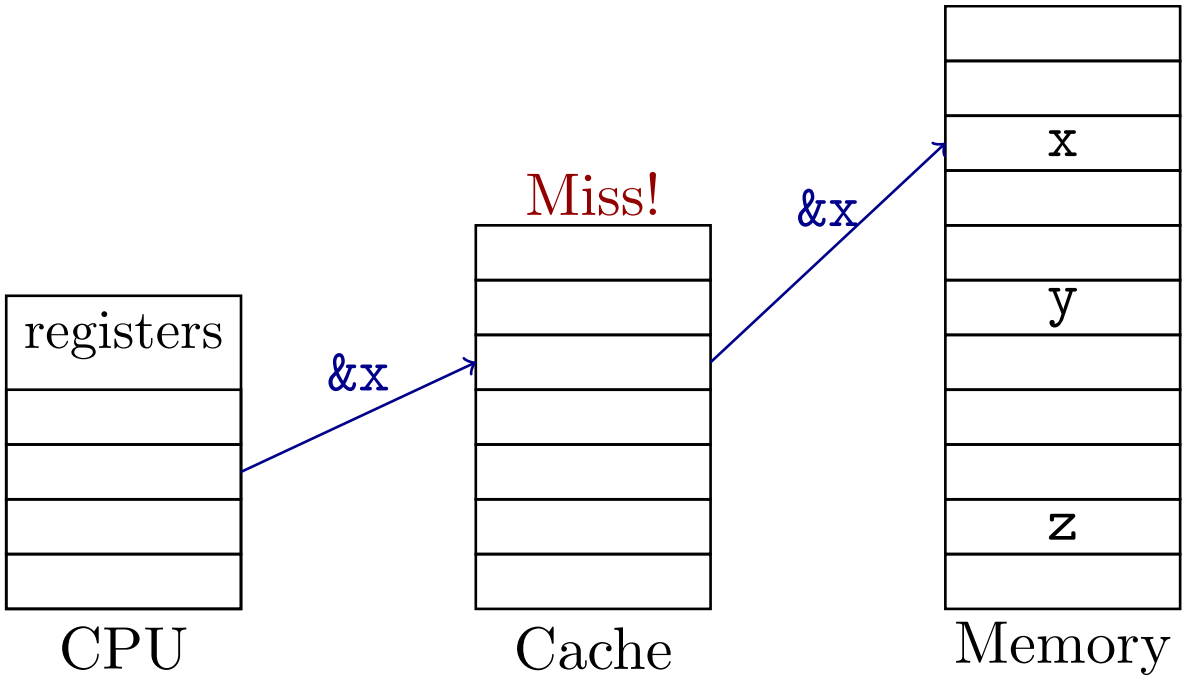
Cache memories



Data must be brought into a register to be used in an instruction. The CPU first looks for the data in the cache

- if the data is found we have a cache hit: the data is copied in the register

Cache memories



Data must be brought into a register to be used in an instruction. The CPU first looks for the data in the cache

- if the data is found we have a cache hit: the data is copied in the register
- if the data is not found we have a cache miss: the data is read from main memory, copied in the cache and then in the register

Cache memories

Where is a data written in the cache? There exist two extreme strategies

- Direct mapped: a data can go in only one location defined by, e.g., its least significant bits (index). Because the cache is smaller than the main memory, multiple data are mapped to the same cache location; the most significant bits (tag) are used to discriminate. Pros: search is fast. Cons: data may be evicted very quickly.
- Fully associative: a data can go in any location. The tag corresponds to the entire address. When the cache is full and a new data comes in, we must choose which data to evict. one common strategy is Least Recently Used (LRU). Pros: data stays in cache as long as its used. Cons: search is very slow.

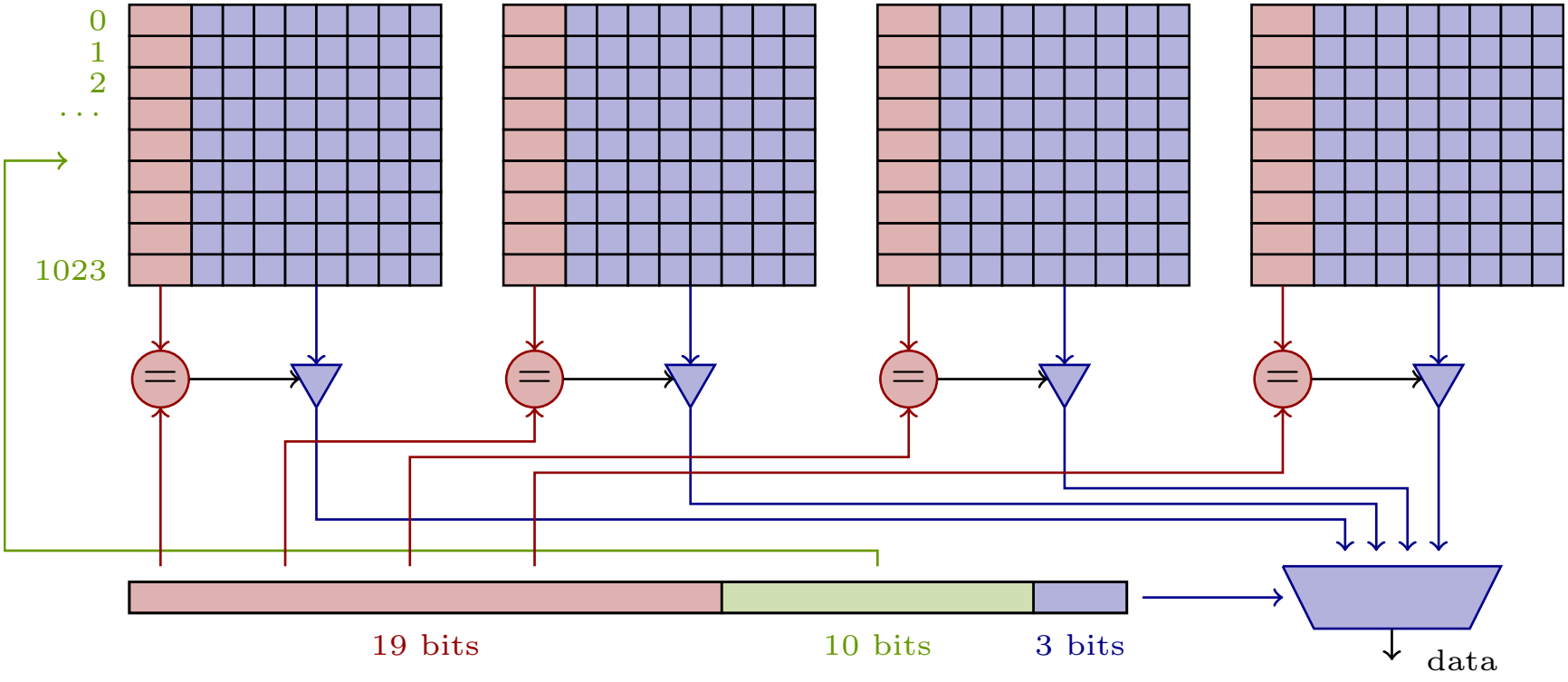
Cache memories

In practice, caches are organized into sets. A data can go in any location of a single set defined by the index bits. Tag bits are used to identify data within a set. When a set is full, LRU can be applied for the eviction.

What about spatial locality? when a cache miss happens, not only the requested data is brought into cache but an entire line, i.e., the nearby data (e.g., 64 Bytes). Data within a line is identified thanks to the offset bits.

Cache memories

Complete example:
32KB, 4-way set associative cache, 8B line



Cache memories

Other cache key points:

- What happens if a data in cache is modified? two common policies.
  - write through: the corresponding data in RAM is immediately updated (easier handling of coherency)
  - write back: data is only modified in cache and the RAM is updated upon eviction (faster but coherency more difficult). In this case a clean/dirty bit is used to define the state of data w.r.t. its copy in the memory
- Valid bit: each block has a bit saying whether its content is valid or not
- LRU bits: how do we know which is the least recently used line in a set? $log_2(k)$ bits for each k-way set
- Coherency: in parallel systems a copy of the same data can exist in multiple caches. Coherency must be ensured when one of these copies is modified (e.g., by invalidating all the others). This is the objective of snooping protocols.

Cache memories

Example: Xeon Gold 6140 (18 cores):

- Level 1:
  - 18×32 KB 8-way set associative instruction caches
  - 18×32 KB 8-way set associative data caches
- Level 2: 18×1 MB 16-way set associative caches
- Level 3: 1×24.75 MB 11-way set associative shared cache

Exercise: for each cache, compute how the address bits are split into tag, index and offset (consider 64bit addressing)

Performance modeling and evaluation

How to model and evaluate the performance of a sequential and/or multithreaded code?

Architecture (AMD Opteron 8431):

- clock frequency 2.4GHz $\rightarrow$ peak performance in DP
  $s = 2.4 * 2 * 2 = 9.6$ Gflop/s (the first 2 is for SSE units, the second is for dual-ALU);
- memory frequency = 667 MHz $\rightarrow$ peak memory bandwidth
  $b = 0.667 * 2 * 8 = 10.6$ GB/s (the 2 is for dual-channel and the 8 is for the bus width).

Assumptions:

- An operation is a succession of data transfers and computations.
- Computations and data transfers can be done at the same time: we will assume that while doing some computations, we can prefetch data for the computations step.
- Once in the cache, the access to data costs nothing.

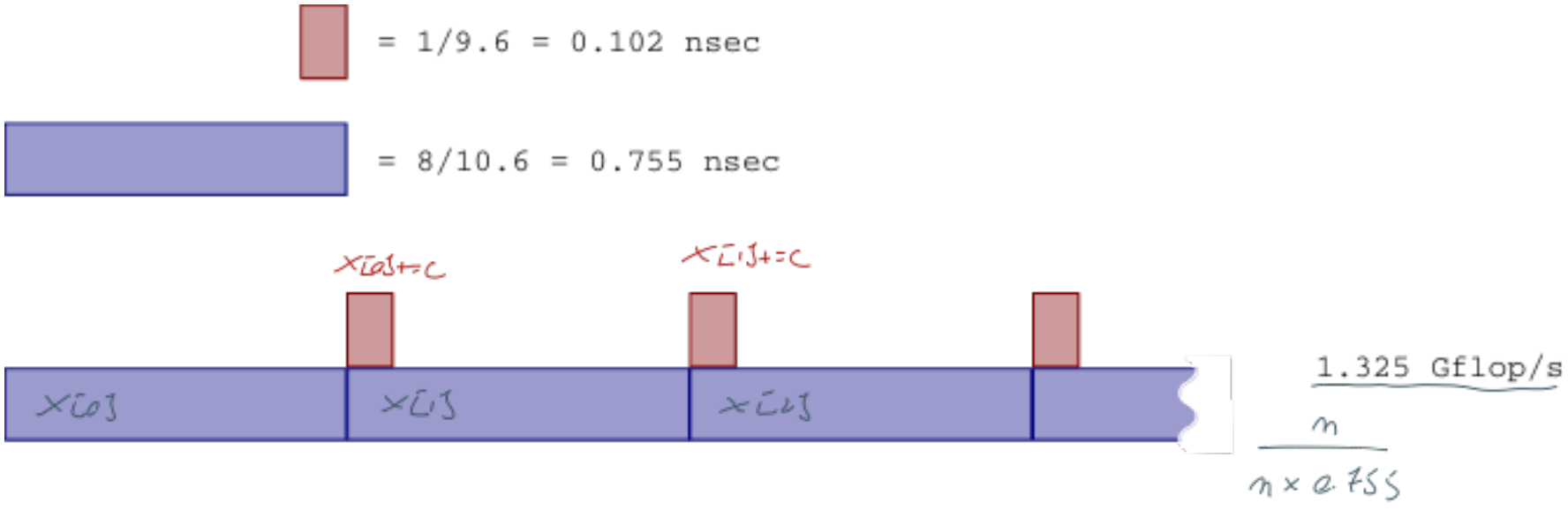
Performance modeling and evaluation

```
for (i=0; i<n; i++)
  x[i] += c;
```

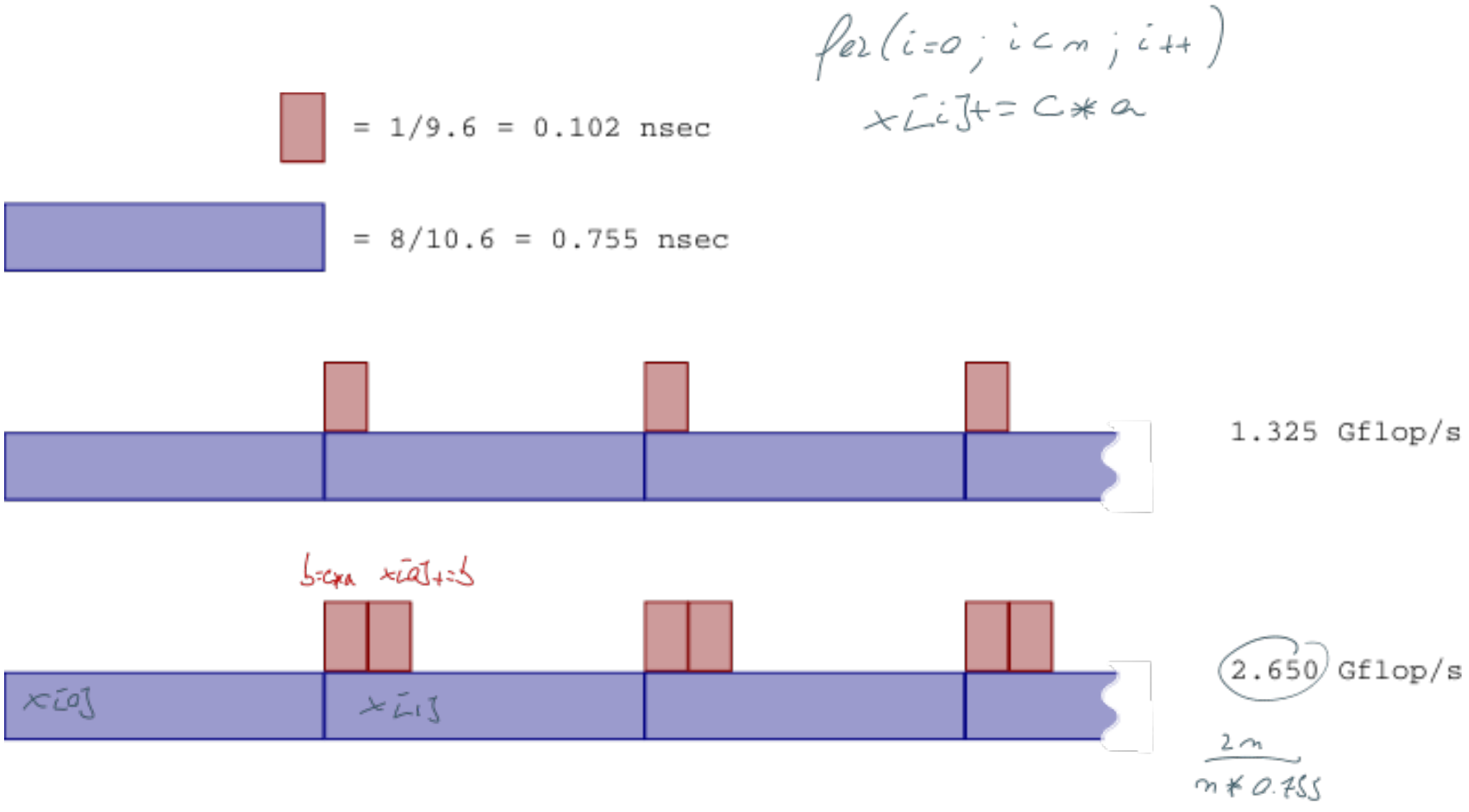
= 1/9.6 = 0.102 nsec

= 8/10.6 = 0.755 nsec

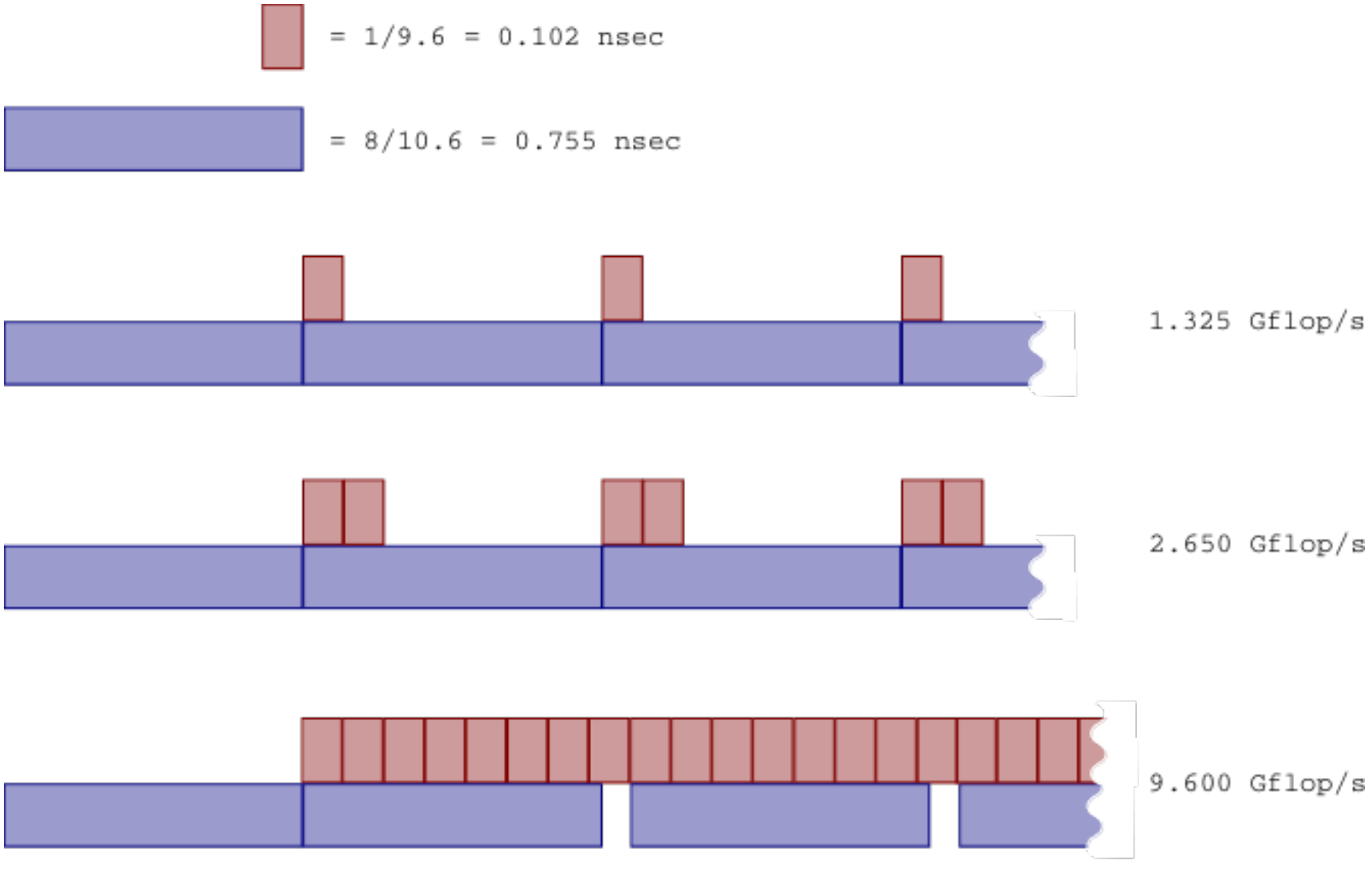
Performance modeling and evaluation



Performance modeling and evaluation



Performance modeling and evaluation



The roofline model

For a given computer, let

- s be the peak performance (speed) of one core in number of floating point operations (flops) per second flop/s
- b the memory bandwidth of one socket in GB/s

For a given algorithm, let

- W be the total workload in number of flops
- D be the volume of data, in bytes, transferred to/from the memory

Under the assumption that computation and memory transfers are done concurrently, the time to execute this algorithm on one core is

$$t = max\left(\frac{W}{s}, \frac{D}{b}\right)$$

The roofline model

What is the maximum speed (in flop/s) at which the algorithm can be executed?

$$\text{max speed} = \frac{W}{t} = min \begin{cases} s \\ \\ b \times \frac{W}{D} \end{cases}$$

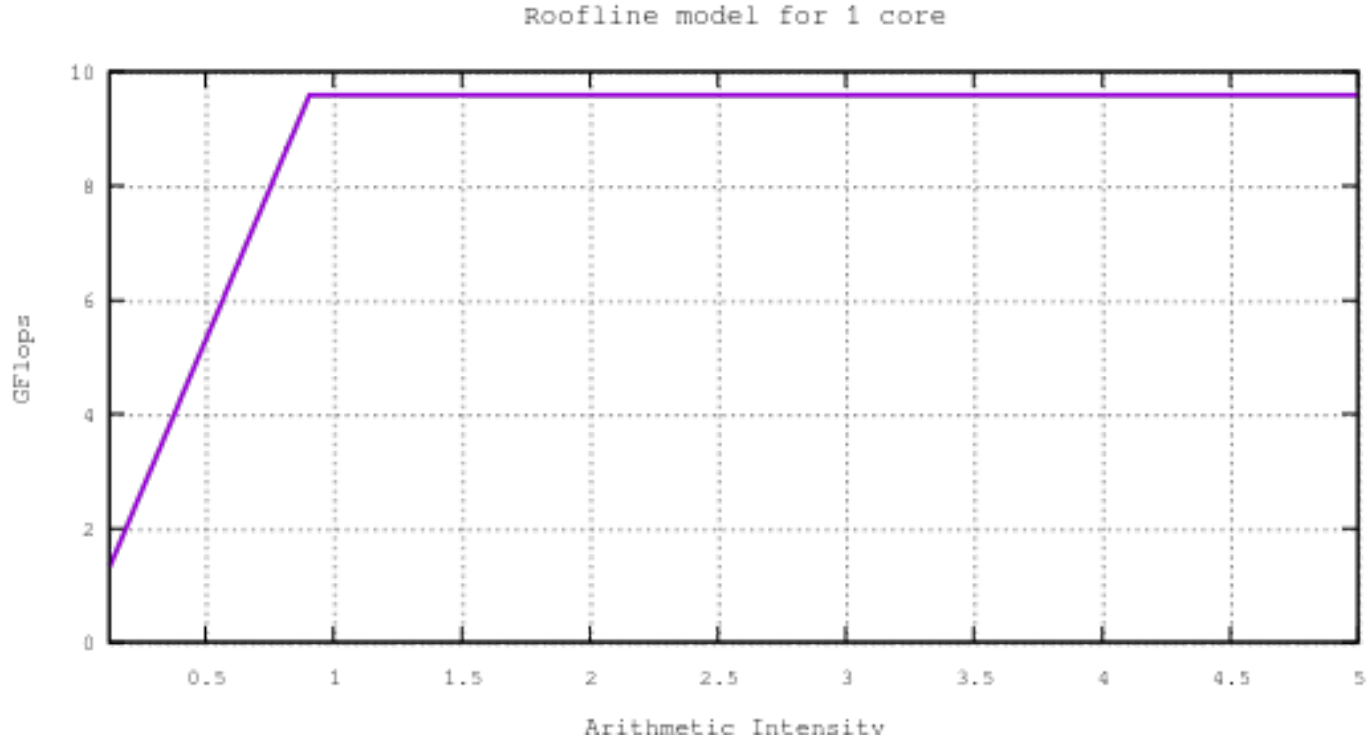
This is called the Roofline model [15]. The quantity W/D is called arithmetic intensity and denotes the amount of work done per byte transferred to/from memory.

The roofline model

What is the maximum speed (in flop/s) at which the algorithm can be executed?

$$\text{max speed} = \frac{W}{t} = min \begin{cases} 9.6 \\ \\ 10.6 \times \frac{W}{D} \end{cases}$$

This is called the Roofline model [15]. The quantity W/D is called arithmetic intensity and denotes the amount of work done per byte transferred to/from memory.



BLAS routines

netlib.org/blas

The BLAS [13, 5, 6] library offers subroutines for basic linear algebra operations (it's called Basic Linear Algebra Subroutines for a reason...). These routines are grouped in three levels

`Level-1`

vector or vector-vector operations such as

- `_axpy`: $y = \alpha x + y$
- `_dot`: $dot = x^T y$
- `_nrm2`: $nrm2 = \|x\|$

BLAS routines

Level-2

matrix-vector operations such as

- _gemv: $y = \alpha Ax + \beta y$
- _ger: $A = \alpha xy^T + A$
- _trsv: $x = T^{-1}x \quad (T \text{ triangular})$

Level-3

matrix-matrix operations such as

- _gemm: $C = \alpha AB + \beta C$
- _trsm: $\alpha X = T^{-1}X \quad (T \text{ triangular})$

BLAS routines

From the point of view of performance, there is a considerable difference among the three levels and it is due to the different ratios between floating-point operations and memory-to-cpu data transfers:

- `Level-1` routines typically perform $\mathcal{O}(n)$ floating-point operations on $\mathcal{O}(n)$ values
- `Level-2` routines typically perform $\mathcal{O}(n^2)$ floating-point operations on $\mathcal{O}(n^2)$ values
- `Level-3` routines typically perform $\mathcal{O}(n^3)$ floating-point operations on $\mathcal{O}(n^2)$ values

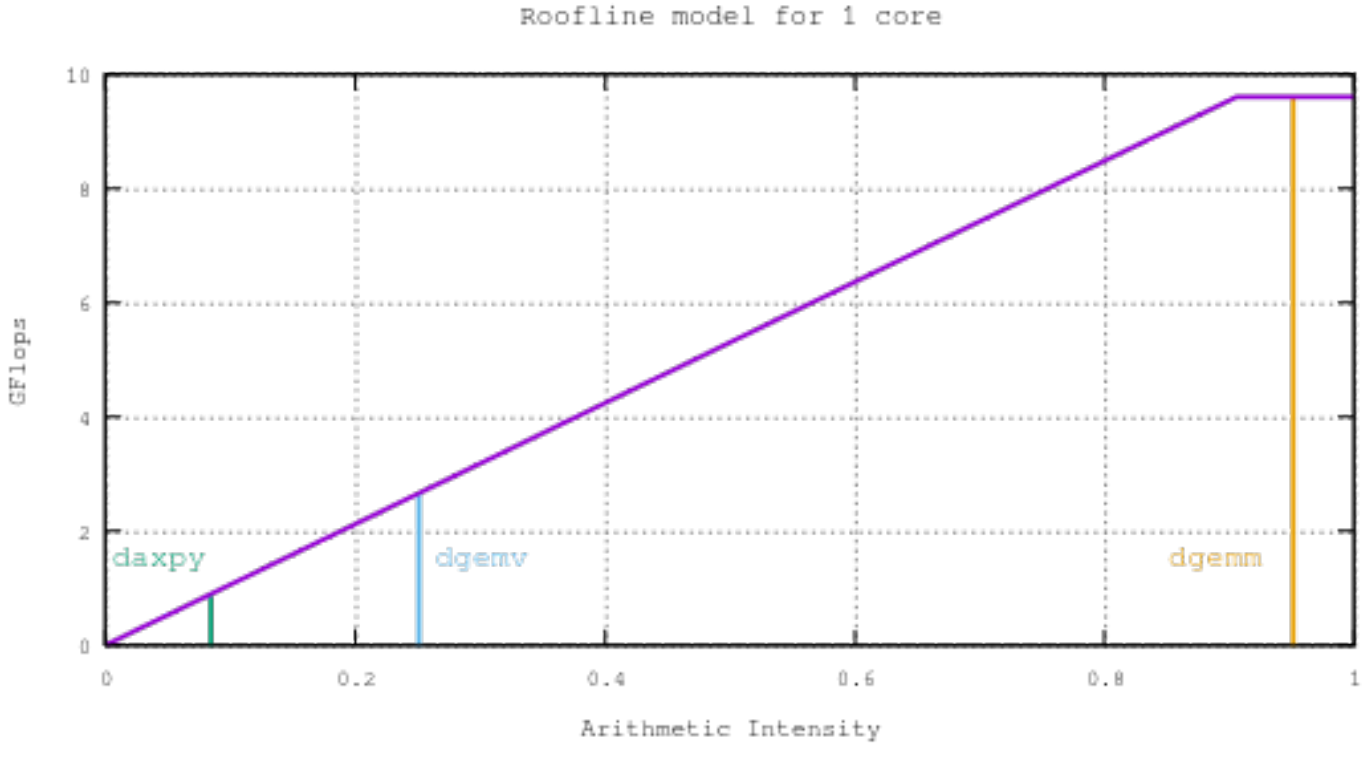
This means that, unlike in `Level-1` and `2` where each coefficient is used only once (no locality, operational intensity $\mathcal{O}(1)$), in `Level-3` routines each coefficient is used n times (lots of locality, operational intensity $\mathcal{O}(n)$).

BLAS routines

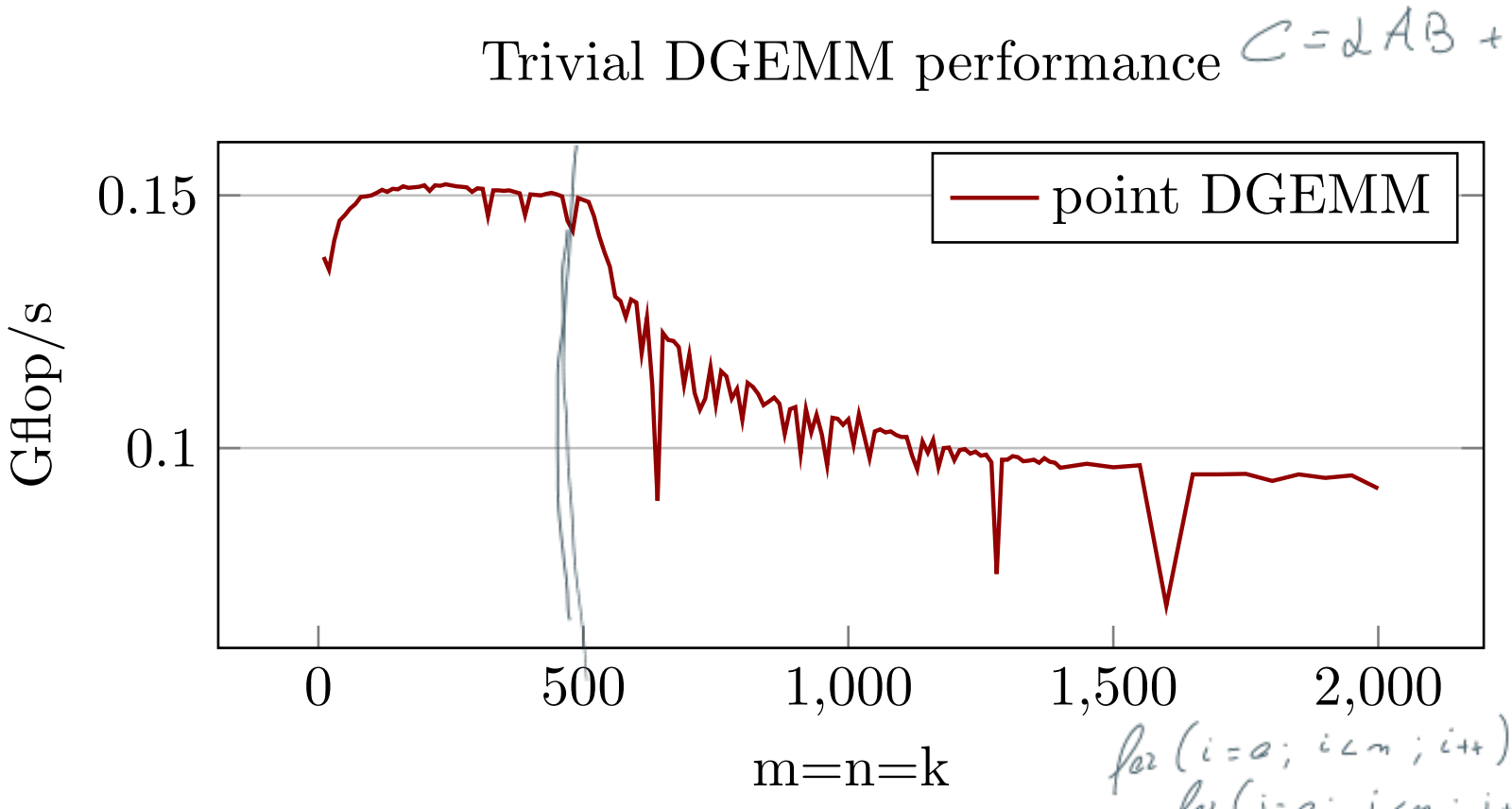
Here is how `Level-1`, `2` and `3` BLAS perform on our reference architecture:

$Y = 2X + Y$

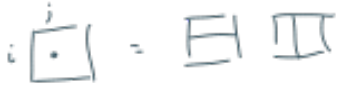
routine	#ops	#data	Meas.	Gflop/s	RM
daxpy	$2n$	$3n$		0.46	0.88
dgemv	$2n^2$	$n^2 + 3n$		1.18	2.65
dgemm	$2n^3$	$4n^2$		8.95	9.60



BLAS routines




```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i,j] = α A[i,k] * B[k,j] +
               β C[i,j]
```

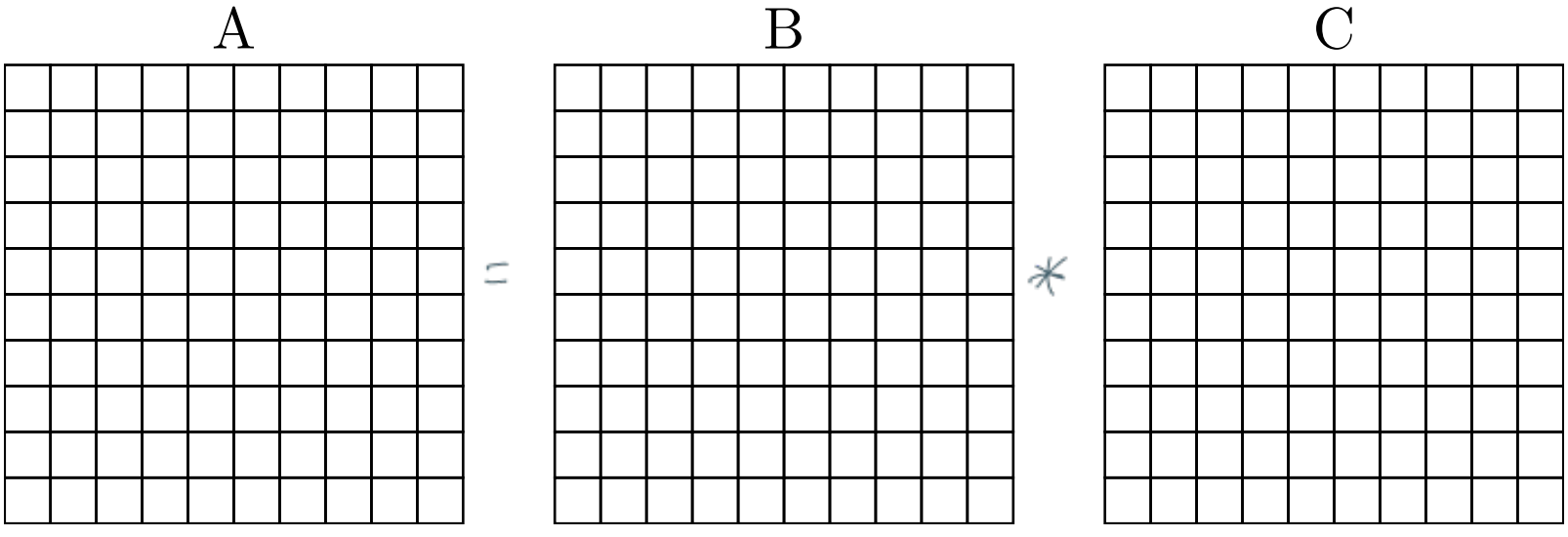


Performance on the AMD Opetron 8431

- much lower than expected (peak 9.6 Gflops/s)
- degrades as the matrix size grows
- very low when the matrix size is a multiple of some power of 2

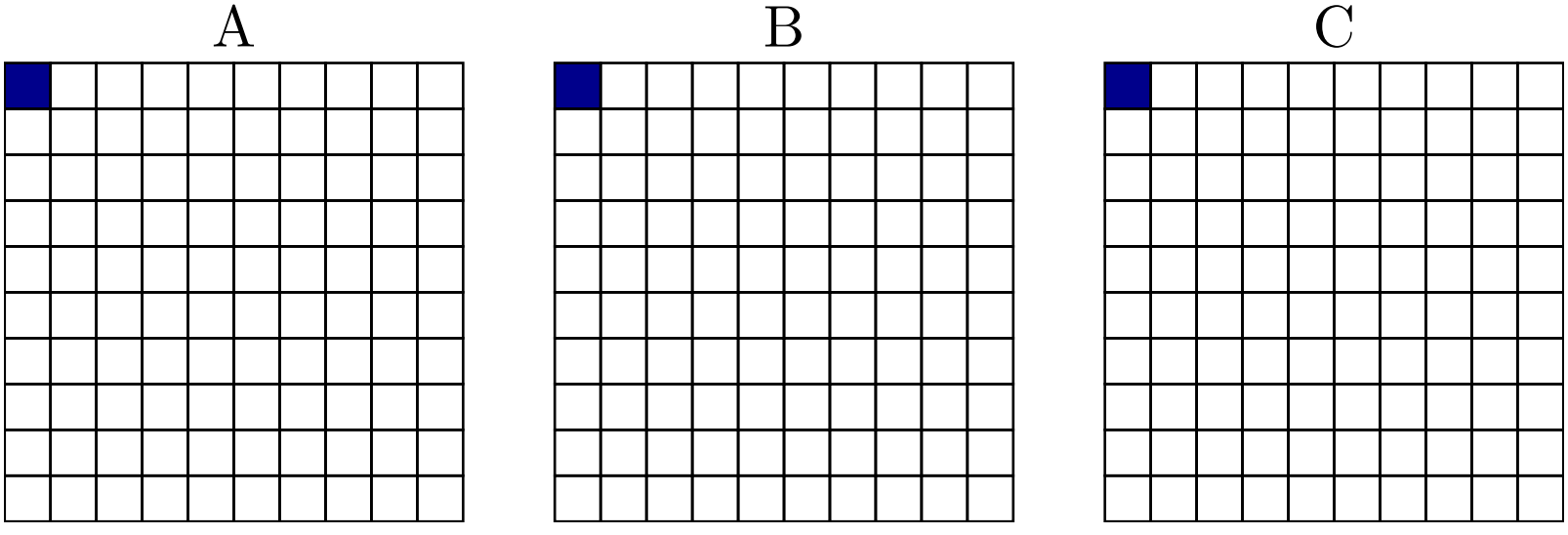
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



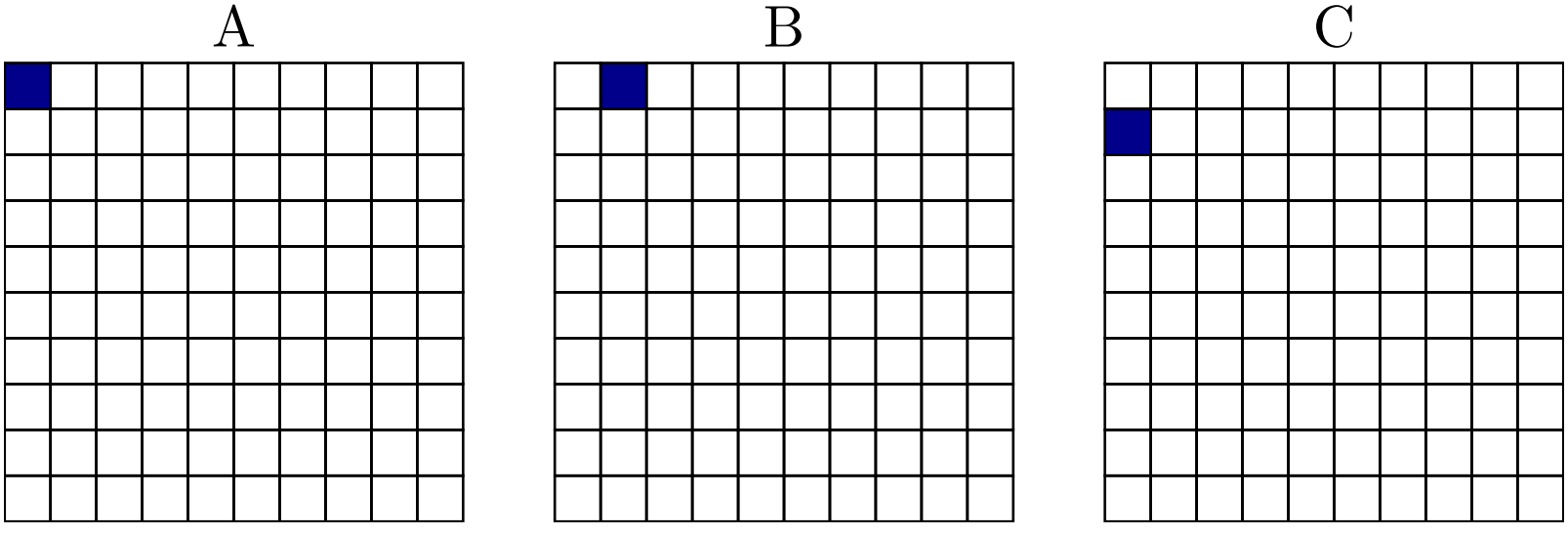
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



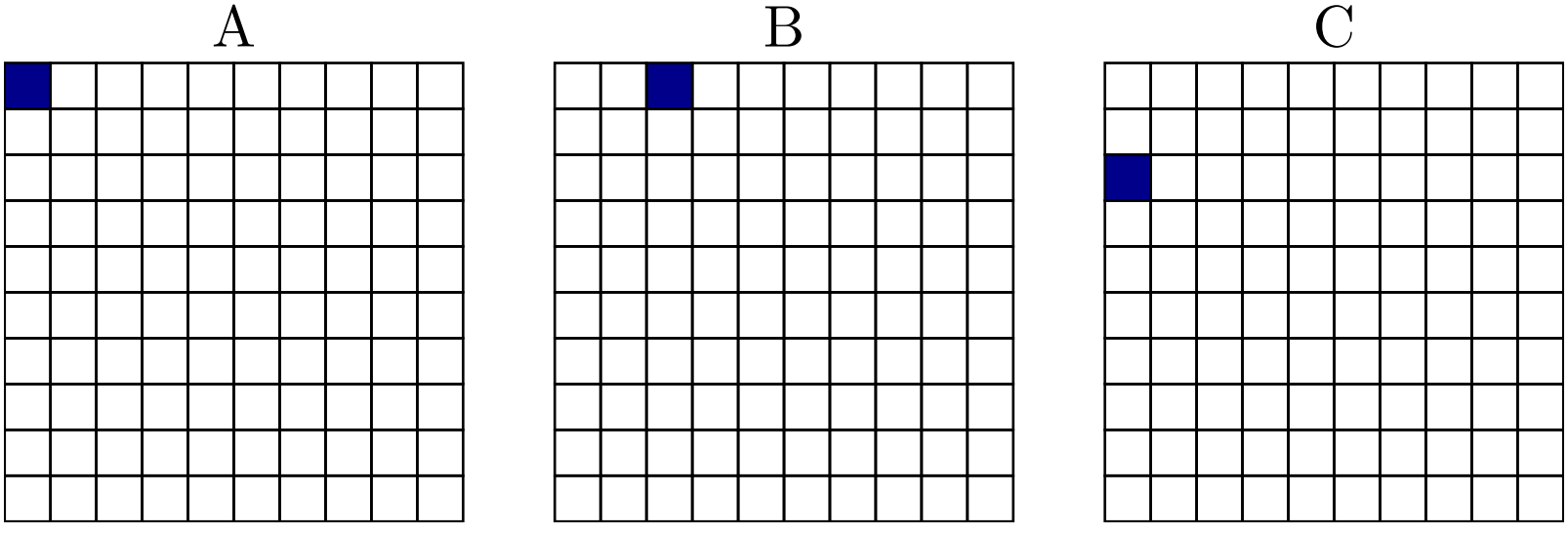
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



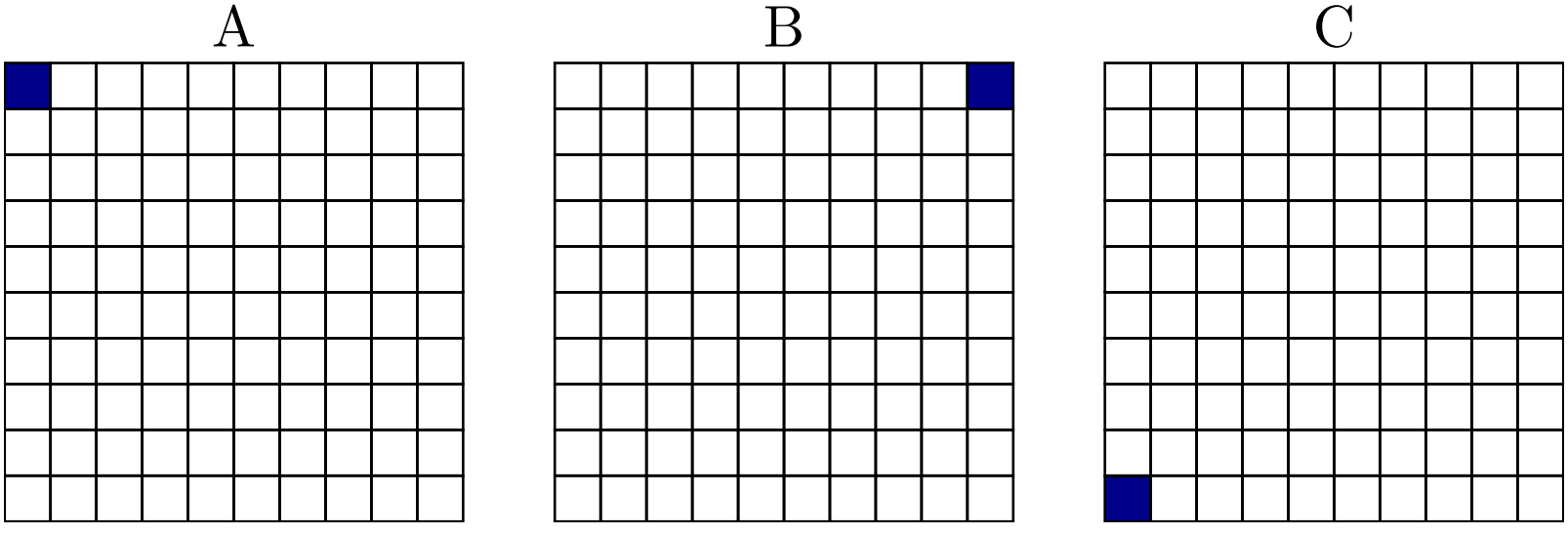
BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.



BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

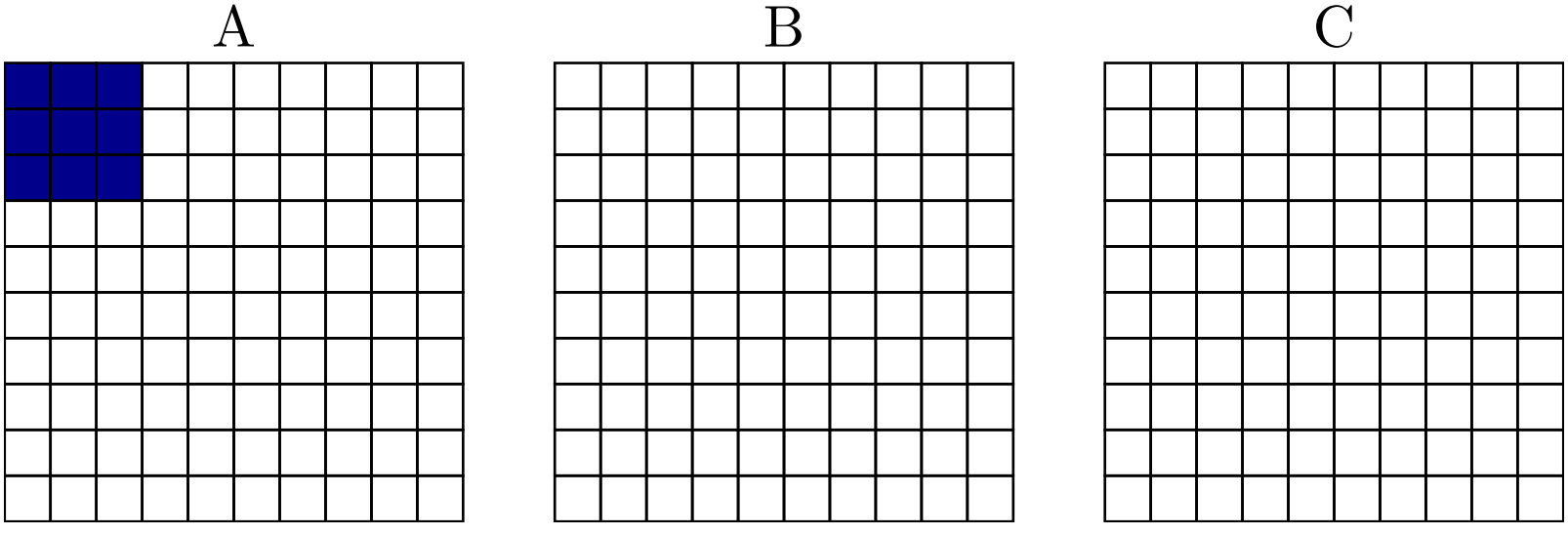


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

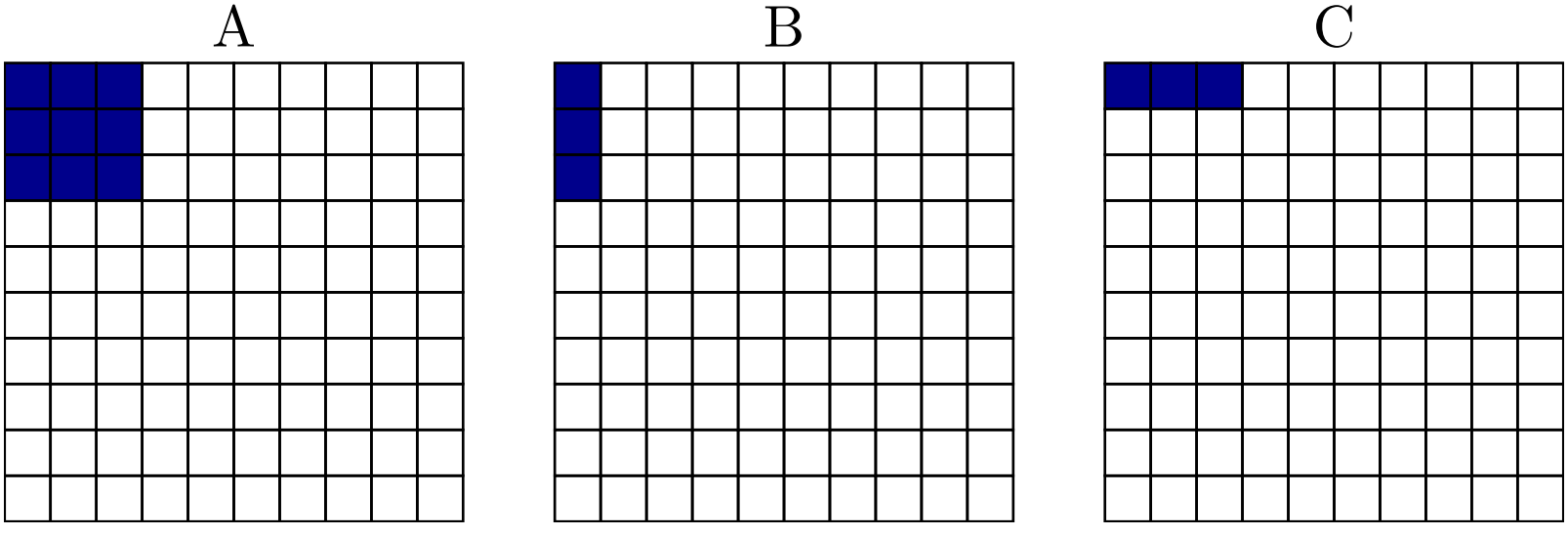


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

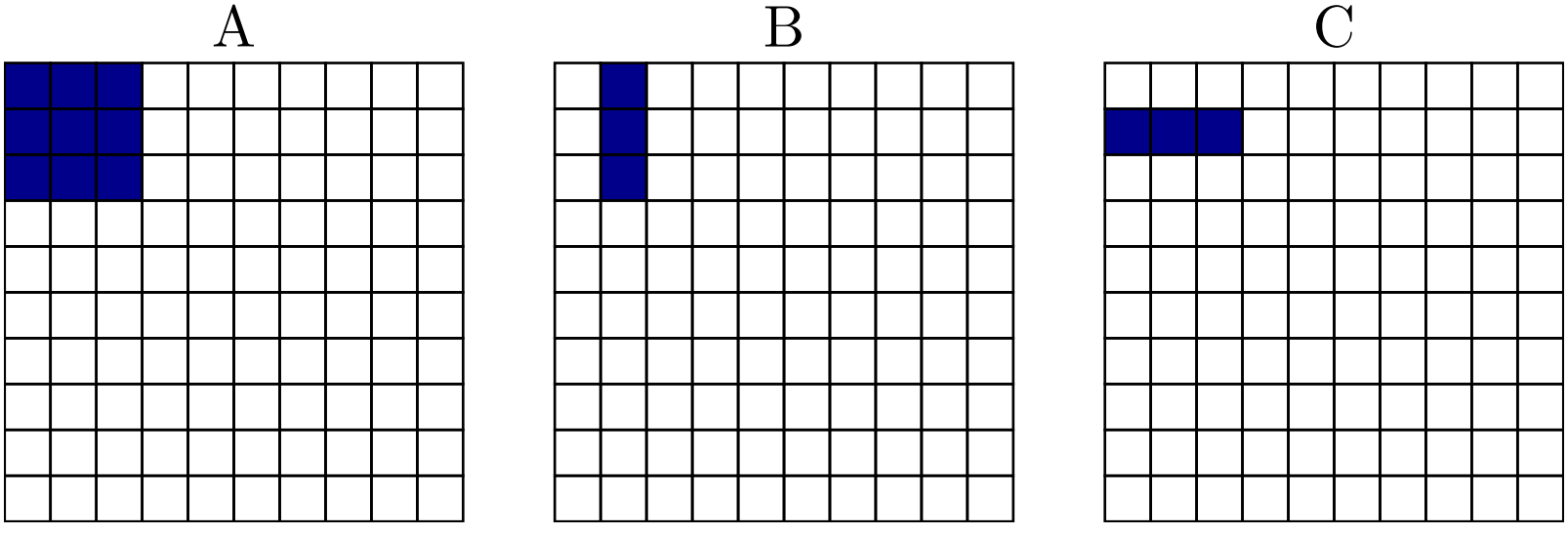


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

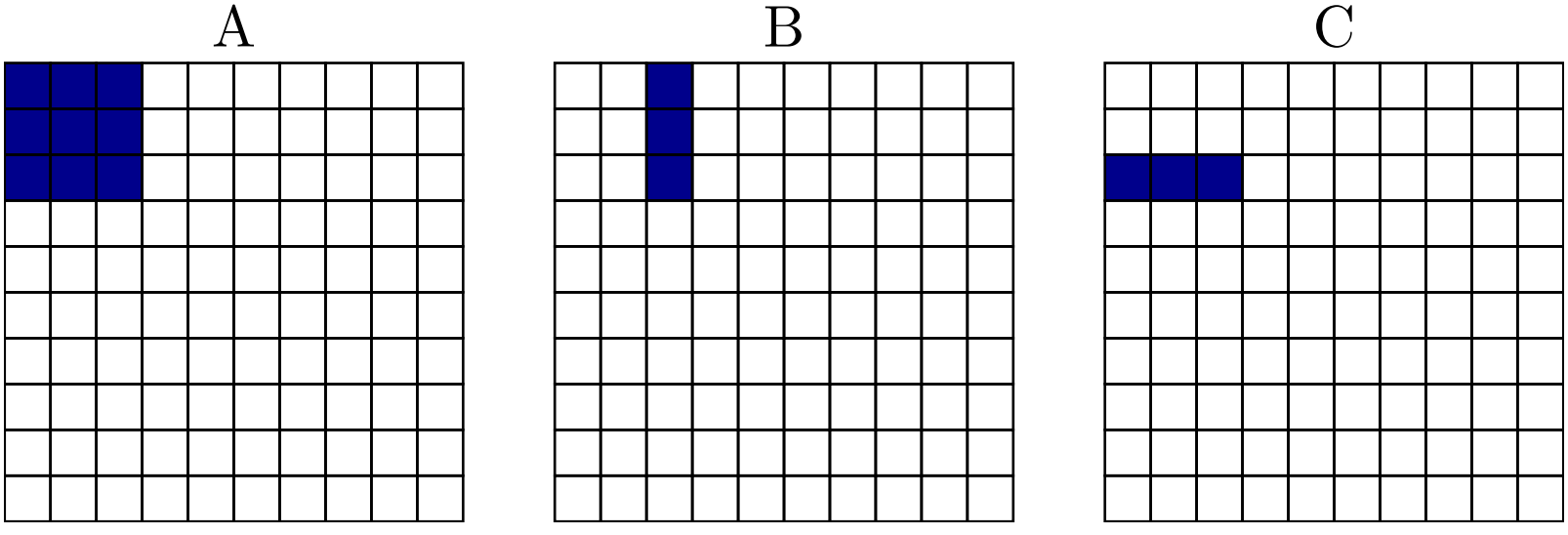


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

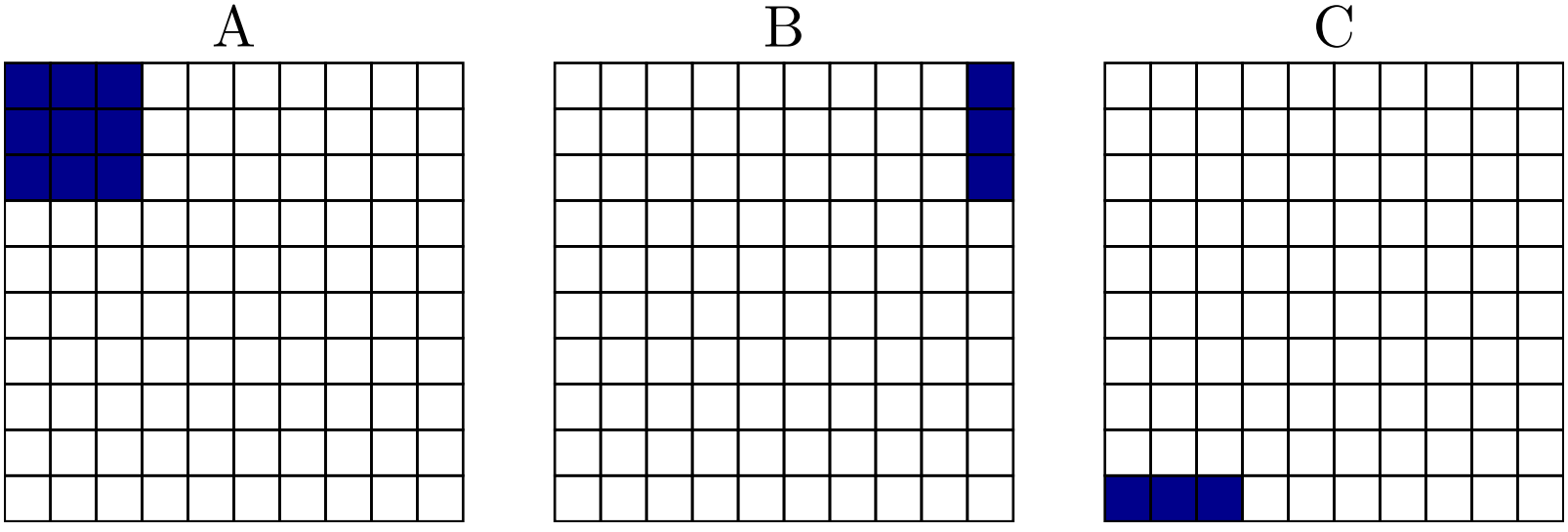


number of cache misses:

- unblocked $= 2n^3 = 2G$

BLAS routines

Although we normally handle matrices that don't fit in any cache level, the locality can be exploited with blocking. Assume the operation $A = A + BC$ with A, B and C square of size 1000 and a fully-associative cache of size 1000 coefficients with line size of 1 coefficient.

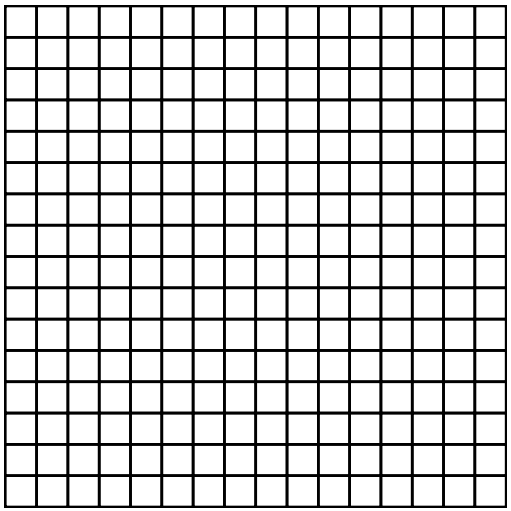


number of cache misses:

- unblocked $= 2n^3 = 2G$
- blocked $= (b^2 + 2bn) * ((n/b)^2) = 67M \quad (b = 30)$

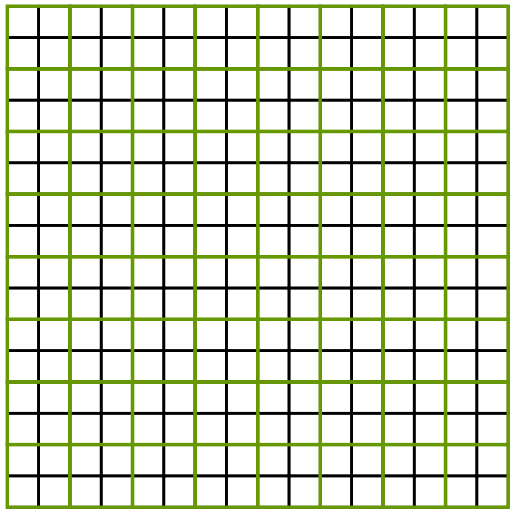
BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



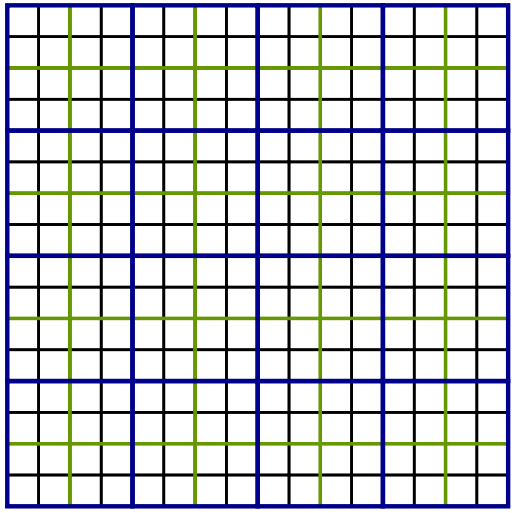
BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



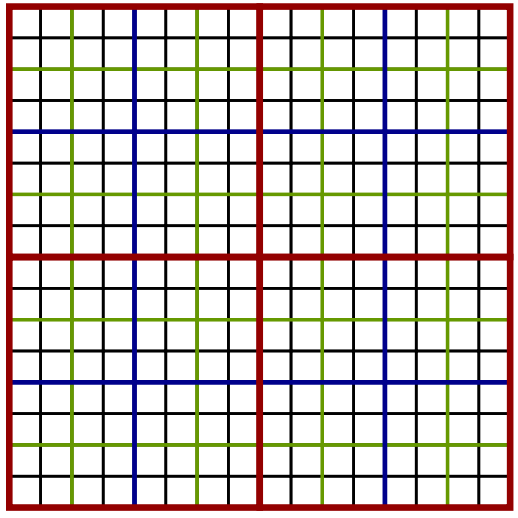
BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



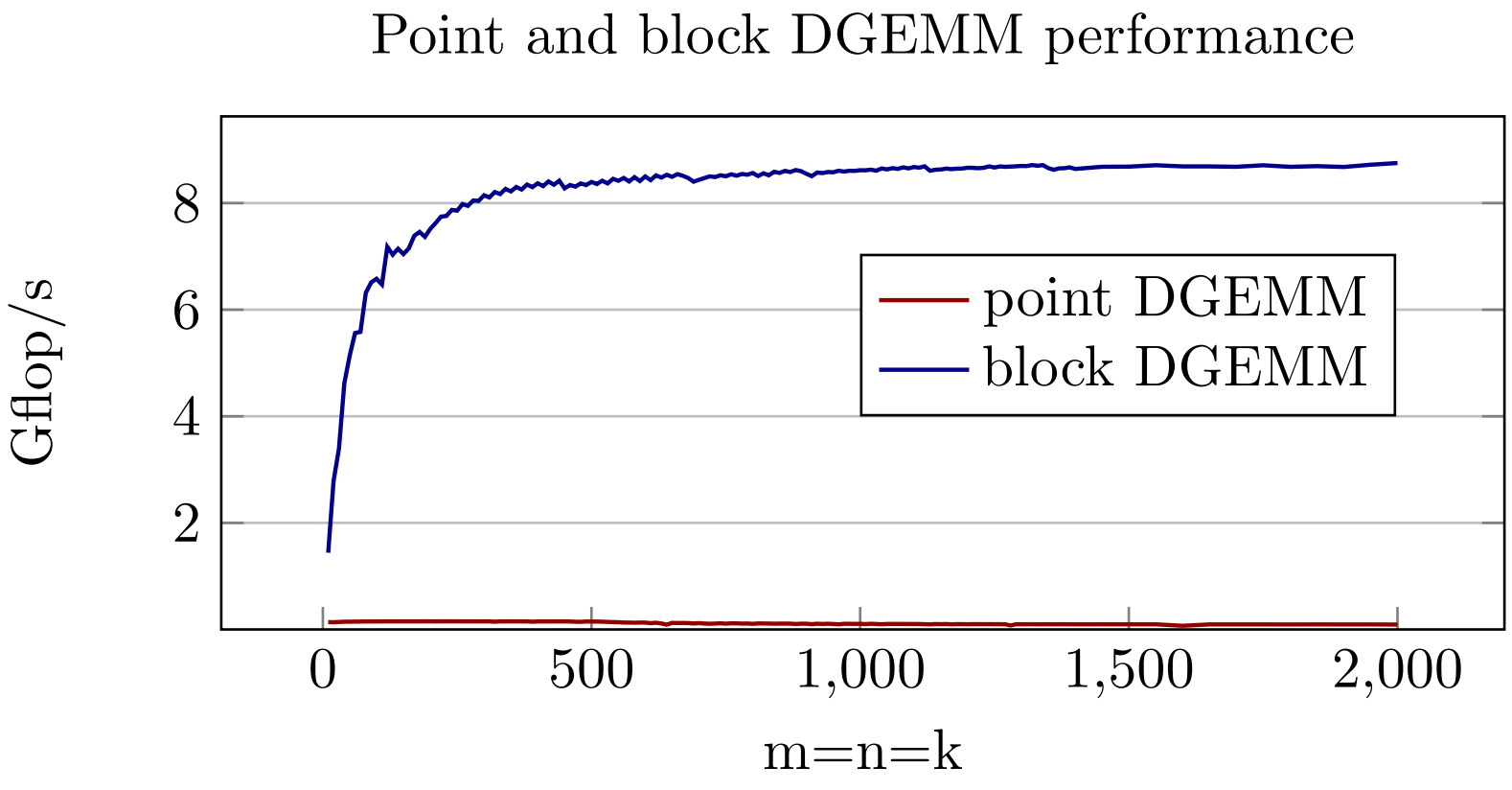
BLAS routines

Because modern processors are equipped with multiple levels of cache memories plus TLB, nested blocking is commonly used to achieve a good memory locality at each level:



More about BLAS optimization techniques can be found in ATLAS [14], or GotoBLAS [10, 9].

BLAS routines



Performance on the AMD Opetron 8431

- much lower than expected (peak 9.6 Gflops/s)
- degrades as the matrix size grows
- very low when the matrix size is a multiple of some power of 2

Efficient cache utilization: Exercise

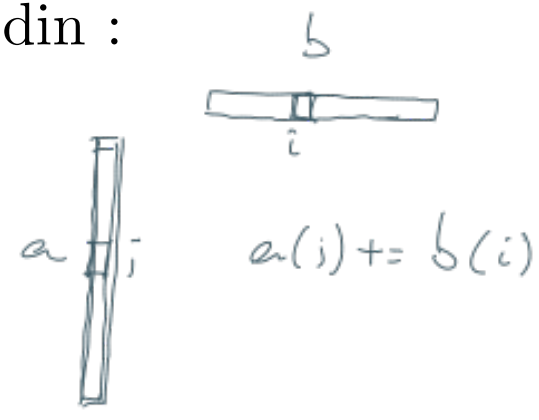
Courtesy of P. Amestoy, M. Daydé and J.-Y. L'Excellent

Reuse as much as possible data in cache $\leftrightarrow$ Improve cache hit ratio

- Cache : single block of CS (cache size) words
- When cache is full: LRU line returned to memory
- Write-back
- For simplicity, we assume cache line size L=1

Example from D. Gannon and F. Bodin :

```
do i=1,n
   do j=1,n
      a(j) = a(j) + b(i)
   enddo
enddo
```



1. Compute the cache hit ratio (assume n much larger than CS).
2. Propose a modification to improve the cache hit ratio.

Efficient cache utilization: Exercise

- Total number of memory references $= 3 \times n^2$ i.e. n^2 loads for `a`, n^2 stores for `a`, and n^2 loads for `b`.
- Total number of flops $= n^2$
- Cache empty at beginning of calculations.
- Inner loop:

```
do j=1,n
   a(j) = a(j) + b(i)
enddo
```

  Each iteration reads `a(j)` and `b(i)`, and writes `a(j)`

  For `i=1` $\rightarrow$ access to `a(1:n)`

  For `i=2` $\rightarrow$ access to `a(1:n)`

  As $n \gg$ CS, `a(j)` no longer in cache when accessed again:

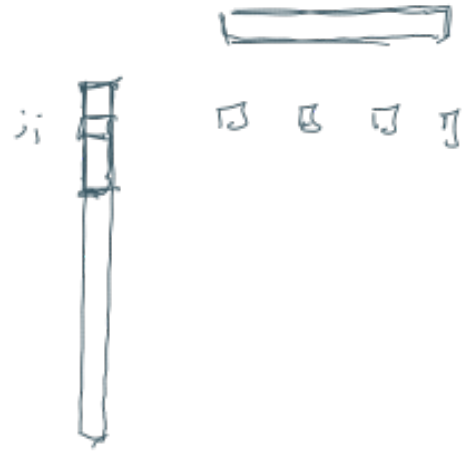
  - each read of `a(j)` $\rightarrow$ 1 miss
  - each write of `a(j)` $\rightarrow$ 1 hit
  - each read of `b(i)` $\rightarrow$ 1 hit (except the first one)

- **Hit ratio** $= \frac{\#\text{ of hits}}{\text{Mem.Refs}} = \frac{2}{3} = 66\%$

blocked version

The inner loop is blocked into blocks of size `nb` $<$ CS so that `nb` elements of a can be kept in cache and entirely updated with `b(1:n)`.

```
do j=1,n,nb
   do i=1,n
      do jj=j,j+nb-1
         a(jj) = a(jj) + b(i)
      enddo
   enddo
enddo
```



blocked version

To clarify we load the cache explicitly; it is managed as a 1D array : `CA(0:nb)`

```
do j=1,n,nb                                          nb miss
   CA(1:nb) = a(j:j+nb-1)
   do i=1,n
      CA(0) = b(i)  1 miss
      do jj=j,j+nb-1
         CA(jj-j+1) = CA(jj-j+1) + CA(0)   nb*3 hit
      enddo
   enddo
   a(j:j+nb-1) = CA(1:nb)   nb  miss
enddo
```

Each load into cache is a miss, each store to cache is a hit.

blocked version

- Total memory references $= 3n^2$
- Total misses:
  - load `a` $= \frac{n}{nb} \times nb$
  - load `b` $= \frac{n}{nb} \times n$
  - Total $= n + \frac{n^2}{nb}$
- Total hits $= 3n^2 - n - \frac{n^2}{nb} = (3 - \frac{1}{nb}) \times n^2 - n$

Hit ratio $= \frac{\text{hits}}{\text{Mem.Refs}} \approx 1 - \frac{1}{3nb} \approx 100\%$

if `nb` is large enough.

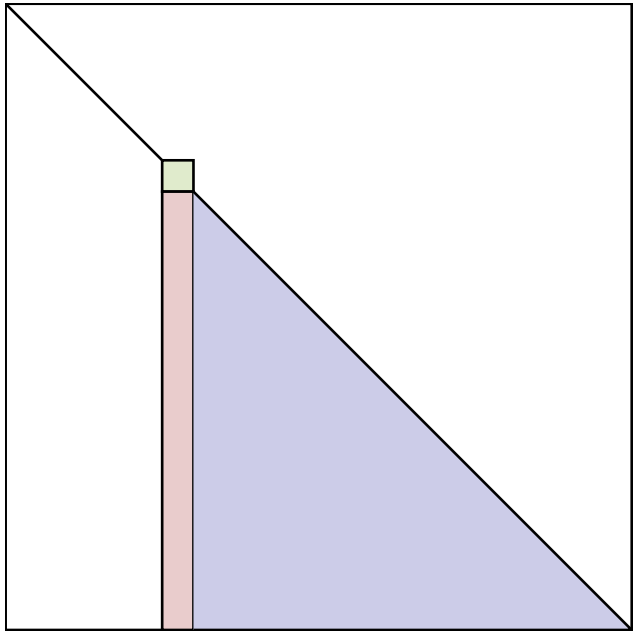
Other dense LA operations: the Cholesky factorization

The Cholesky factorization computes a lower-triangular matrix L such that $A = LL^T$ where A is a symmetric positive definite matrix. This allows to solve a linear system $Ax = b$ in three steps

$$1)\ A = LL^T, \qquad 2)\ Lz = b, \qquad 3)\ L^T x = z$$

Algorithm 1 Cholesky factorization

for $k = 1, ..., n$ **do**
  $l_{kk} = \sqrt{a_{kk}^{(k-1)}}$
  for $i = k+1, ..., n$ **do**
    $l_{ik} = a_{ik}^{(k-1)} / l_{kk}$
  end for
  for $i = k+1, ..., n$ **do**
    for $j = k+1, ..., i$ **do**
      $a_{ij}^{(k)} = a_{ij}^{(k-1)} - l_{ik} l_{jk}$
    end for
  end for
end for



Other dense LA operations: the Cholesky factorization

This is (pretty much) the point Cholesky factorization implemented in LAPACK [2]

```
subroutine dpotf2(a, n)
do k=1, n
   a(k,k) = sqrt(a(k,k))
   call dscal(a(k+1:n,k), a(k,k))
   call dsyr(a(k+1:n,k), a(k+1:n,k+1:n))
end do
```

- `dscal` computes $l_{k+1:n,k} = a^{(k-1)}_{k+1:n,k}/l_{kk}$
- `dsyrk` computes $a^{(k)}_{k+1:n,k+1:n} = a^{(k)}_{k+1:n,k+1:n} - l_{k+1:n,k} l^T_{k+1:n,k}$ (only lower half)
- factorization is in-place, i.e., L is stored in place of A

Exercise: What is the arithmetic intensity? Answ:

$$\left(\frac{1}{3}n^3\right) \;/\; \left(\frac{1}{2}n^2\right) = O(n)$$

Other dense LA operations: the Cholesky factorization

The Cholesky suffers from the same problem as the naive matrix multiply: if the matrix does not fit in cache the high arithmetic intensity cannot be exploited → blocking

Assume a block size $b \ll n$ and split A such that A_{11} is of size $b \times b$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \cdot \begin{bmatrix} L_{11}^T & L_{21}^T \\ & A_{22}^{(1)} \end{bmatrix}$$

$$A_{11} = L_{11}L_{11}^T, \quad A_{21} = L_{21}L_{11}^T, \quad A_{22}^{(1)} = A_{22} - L_{21}L_{21}^T$$

Iterate this procedure on $A_{22}^{(1)}$

The factorization is now written in terms of matrix-matrix BLAS3 operations which make efficient use of caches

Other dense LA operations: the Cholesky factorization

This is (pretty much) the block Cholesky factorization implemented in LAPACK [2]

```
subroutine dpotrf(a, n)
do k=1, n, b
   call dpotf2(a(k:k+b-1,k:k+b-1))
   call dtrsm(a(k+b:n,k:k+b-1), a(k:k+b-1,k:k+b-1))
   call dsyrk(a(k+b:n,k:k+b-1), (a(k+b:n,k+b:n))
end do
```

- `dpotf2` computes the Cholesky factorization of $A_{k:k+b-1,k:k+b-1}$
- `dtrsm` computes $L_{k+b:n,k:k+b-1} = A_{k+b:n,k:k+b-1} L^{-T}_{k:k+b-1,k:k+b-1}$
- `dsyrk` computes (only lower half)
  $A^{(k)}_{k+b:n,k+b:n} = A^{(k-1)}_{k+b:n,k+b:n} - L_{k+b:n,k:k+b-1} L^{T}_{k+b:n,k:k+b-1}$

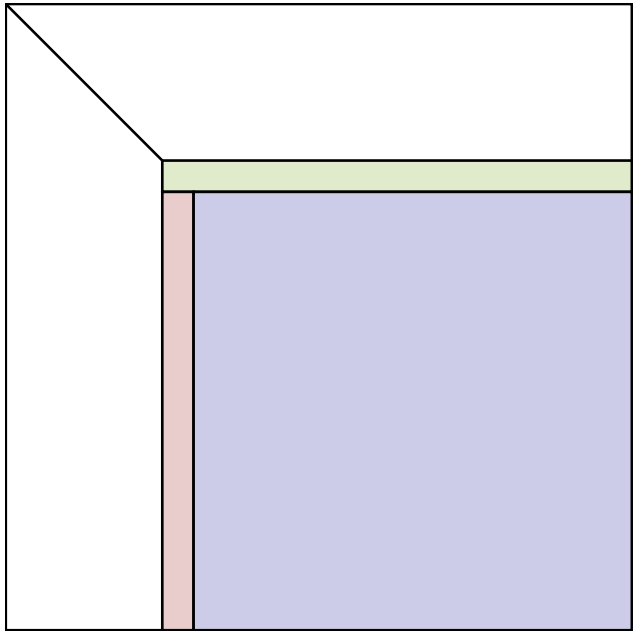
Even if `dpotf2` has low efficiency (point factorization) this doesn't really matter because $b \ll n$

Other dense LA operations: the LU factorization

For the LU factorization things get more complicated because of numerical issues:

Algorithm 2 LU factorization

for $k = 1, ..., n$ **do**
  $u_{k,k:n} = a^{(k-1)}_{k,k:n}$
  for $i = k+1, ..., n$ **do**
    $l_{ik} = a^{(k-1)}_{ik} / u_{kk}$
  end for
  for $i = k+1, ..., n$ **do**
    for $j = k+1, ..., i$ **do**
      $a^{(k)}_{ij} = a^{(k-1)}_{ij} - l_{ik} u_{kj}$
    end for
  end for
end for



Other dense LA operations: the LU factorization

Assume a computer with 3 decimal digits floating point representation of the form $\pm 0.d_1 d_2 d_3 \cdot 10^i$ and the system

$$\begin{pmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5.20 \\ 4.57 \end{pmatrix}$$

to be solved by factorization.
Because $-1.58 - 2420 = -2420$ in the three digits representation, the LU factorization results in

$$L = \begin{pmatrix} 1 & \\ 1000 & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 0.001 & 2.42 \\ & -2420 \end{pmatrix}$$

and the computed solution

$$\tilde{x} = \begin{pmatrix} 0.00 \\ 2.15 \end{pmatrix} \neq \text{ true solution } = \begin{pmatrix} 1.18 \\ 2.15 \end{pmatrix}$$

These errors are due to element growth in the factors

Other dense LA operations: the LU factorization

The previous example gives us the intuition that errors occur when coefficients grow too large. This can be better formalized.

It can be shown [12, Th. 9.4] that the approximate solution $\hat{x}$ of a linear system computed through Gaussian Elimination, with approximate factors $\hat{L}$ and $\hat{U}$ satisfy the following relations

$$(A + \Delta A)\hat{x} = b, \quad |\Delta A| \leq \gamma_{3n}|\hat{L}||\hat{U}|$$

Wilkinson [12, Th. 9.5] shows that

$$\||\hat{L}||\hat{U}|\|_\infty \leq c_{n^2}\rho\|A\|_\infty \qquad \rho = \frac{\max_{i,j}|a_{ij}^{(k)}|}{\max_{i,j}|a_{ij}|}$$

where ρ is the so-called growth factor. Therefore Gaussian elimination is *backward stable* provided that $\||\hat{L}||\hat{U}|\| = O(\|A\|)$, or, equivalently, if ρ is small. This cannot be guaranteed, unless pivoting is used

Other dense LA operations: the LU factorization

Partial pivoting determines a rows permutation such that at each elimination step k the pivot with biggest absolute value in the k-th column is permuted in pivotal position.

$$pivot_k = \max_{i=k+1\ldots n} |a_{ik}^{(k-1)}|$$

Applying partial pivoting to the previous example yields

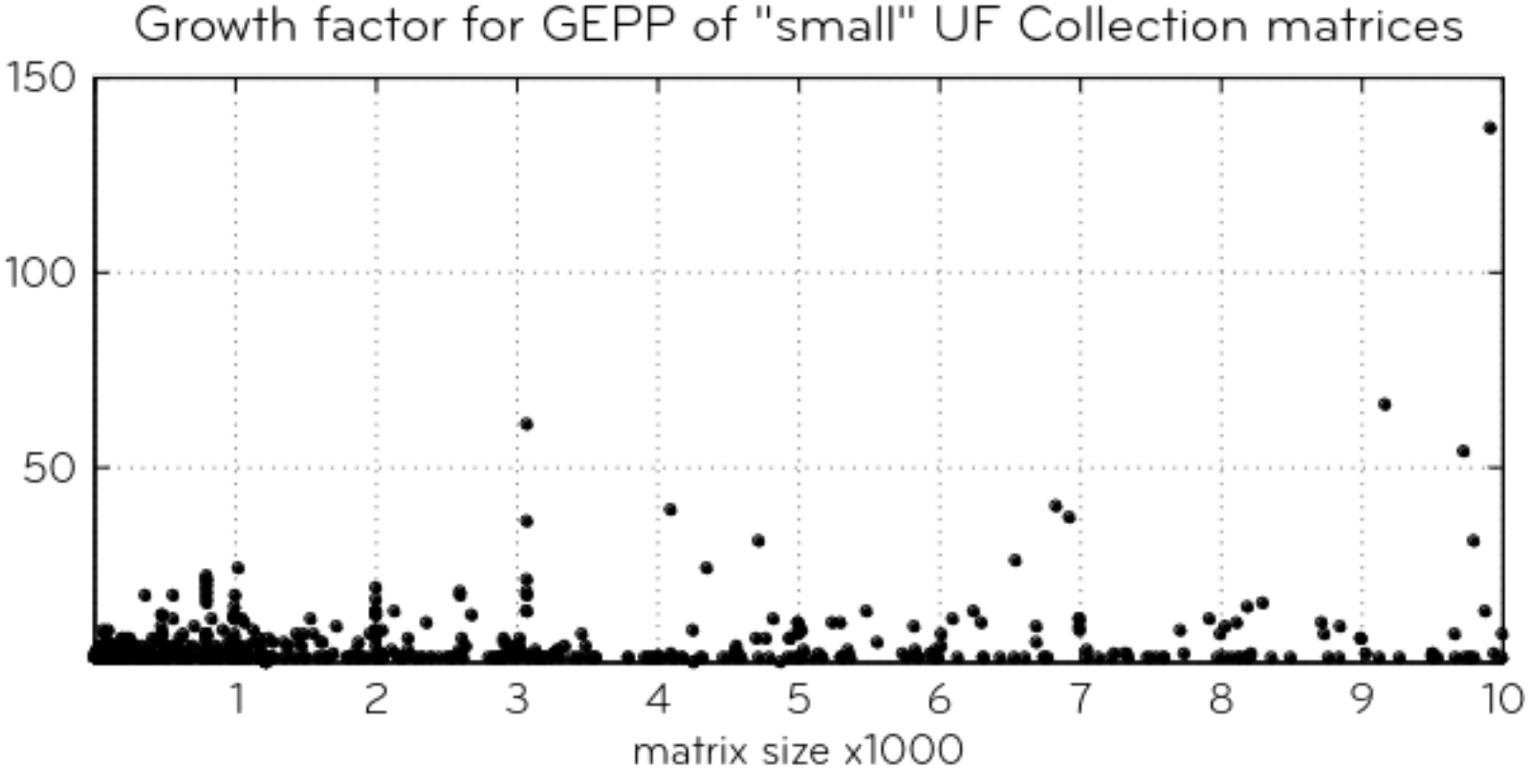
$$PA = \begin{pmatrix} 1.00 & 1.58 \\ 0.001 & 2.42 \end{pmatrix}$$

$$L = \begin{pmatrix} 1 & \\ 0.001 & 1 \end{pmatrix} \qquad U = \begin{pmatrix} 1.00 & 1.58 \\ & 2.42 \end{pmatrix}$$

$$\tilde{x} = \begin{pmatrix} 1.17 \\ 2.15 \end{pmatrix} \simeq \begin{pmatrix} 1.18 \\ 2.15 \end{pmatrix}$$

Partial pivoting

The previous result says that, in theory, GEPP is not stable because $|\hat{L}| = O(1)$ but $|\hat{U}|$ is not bound. The growth factor, in fact, can be as big as 2^{n-1}. In practice, however, GEPP always works since large growth factors never appear in real applications.



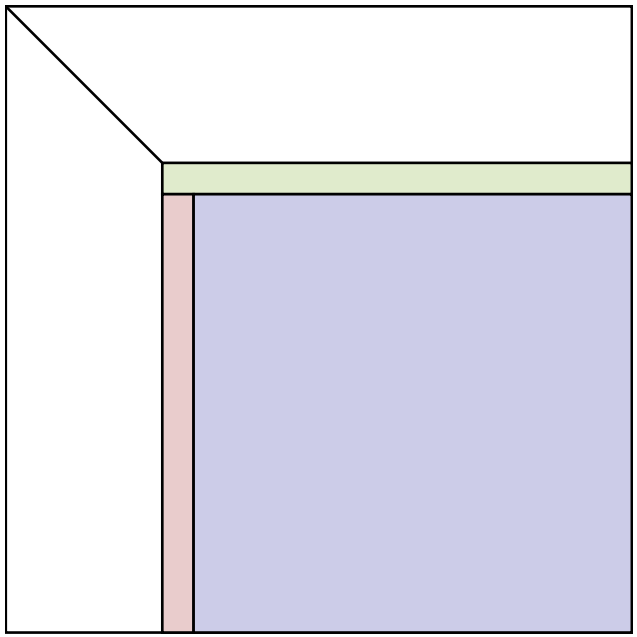
We say that GEPP is practically stable.

Other dense LA operations: the LU factorization

For the LU factorization things get more complicated because of pivoting. We compute $PA = LU$ where P is a permutation

Algorithm 3 LU factorization

for $k = 1, ..., n$ **do**
  find $\text{argmax}_i \left(\left| a_{ik}^{(k-1)} \right| \right)$, $i = k, ..., n$
  swap row i and row k
  $u_{k,k:n} = a_{k,k:n}^{(k-1)}$
  for $i = k+1, ..., n$ **do**
    $l_{ik} = a_{ik}^{(k-1)} / u_{kk}$
  end for
  for $i = k+1, ..., n$ **do**
    for $j = k+1, ..., i$ **do**
      $a_{ij}^{(k)} = a_{ij}^{(k-1)} - l_{ik} u_{kj}$
    end for
  end for
end for



Other dense LA operations: the LU factorization

This is (pretty much) the point LU factorization implemented in LAPACK [2]

```
subroutine dgetrf2(a, n)
do k=1, n
   i = idamax(a(k:n,k))
   call dswap(a(i,:), a(k,:))
   call dscal(a(k+1:n,k), a(k,k))
   call dger(a(k+1:n,k), a(k,k+1:n), a(k+1:n,k+1:n))
end do
```

- `idamax` finds the index `i` of the max coefficient in column `k`
- `dswap` swaps rows `k` and `i`
- `dscal` computes $l_{k+1:n,k} = a^{(k-1)}_{k+1:n,k}/u_{kk}$
- `dger` computes $a^{(k)}_{k+1:n,k+1:n} = a^{(k-1)}_{k+1:n,k+1:n} - l_{k+1:n,k}u_{k,k+1:n}$
- factorization is in-place, i.e., L and U are stored in place of A

Exercise: What is the arithmetic intensity? Answ:

$$\left(\frac{2}{3}n^3\right) \ / \ \left(n^2\right) = O(n)$$

Other dense LA operations: the LU factorization

Blocking for the LU factorization follows the same principle as for Cholesky: compute b transformations and then apply them at once to the trailing submatrix using BLAS3 operations

Assume a block size $b \ll n$ and split A such that A_{11} is of size $b \times b$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ & A_{22}^{(1)} \end{bmatrix}$$

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} [U_{11}], \quad A_{12} = L_{11}U_{12}, \quad A_{22}^{(1)} = A_{22} - L_{21}U_{12}$$

Iterate this procedure on $A_{22}^{(1)}$

The factorization is now written in terms of matrix-matrix BLAS3 operations which make efficient use of caches

Other dense LA operations: the LU factorization

This is (pretty much) the block LU factorization implemented in LAPACK [2]

```
subroutine dgetrf(a, n)
do k=1, n, b
   call dgetrf2(a(k:n,k:k+b-1))
   call dswap(a(i,k+b:n), a(k,k+b:n))
   call dtrsm(a(k:k+b-1,k+b:n), a(k:k+b-1,k:k+b-1))
   call dgerk(a(k+b:n,k:k+b-1), a(k:k+b-1,k+b:n),
              a(k+b:n,k+b:n))
end do
```

- `dgetrf2` computes the LU factorization of $A_{k:n,k:k+b-1}$
- `dtrsm` computes $L^{-1}_{k:k+b-1,k:k+b-1} U_{k:k+b-1,k+b:n} = A_{k:k+b-1,k+b:n}$
- `dsyrk` computes
  $A^{(k)}_{k+b:n,k+b:n} = A^{(k-1)}_{k+b:n,k+b:n} - L_{k+b:n,k:k+b-1} U_{k:k+b-1:k+b:n}$

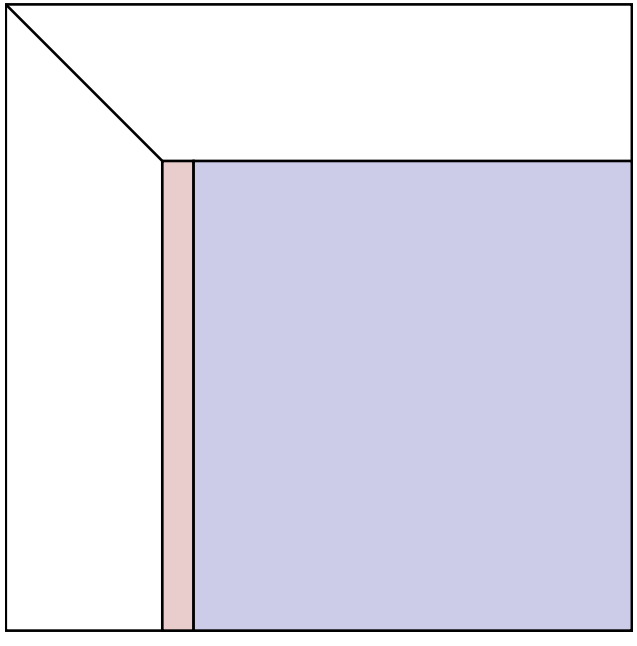
Even if `dgetrf2` has low efficiency (point factorization) this doesn't really matter because $b \ll n$

Other dense LA operations: the QR factorization

This algorithm decomposes the matrix A of size $m \times n$into the product of an orthogonal matrix Q and a triangular one R and can be used to solve linear systems of any shape. It is commonly used to solve least-squares or least-norm problems.

Algorithm 4 QR factorization

for $k = 1, ..., n$ **do**
$[v_k, \tau_k] = \textbf{house}(A_{k:m,k})$
$A_{k:m,k:n} = (I^{m-k+1} - \tau_k v_k v_k^T) A_{k:m,k:n}$
end for



The Q matrix is implicitly represented through the v vectors and τ scalars

Other dense LA operations: the QR factorization

This is (pretty much) the point QR factorization implemented in LAPACK [2]

```
subroutine dgeqr2
do   k = min(m,n)
   call dlarfp(a(k:m,k), tau( k ) )
   call dlarf(a(k:m,k), tau(k), a(k:m,k+1:n))
end do
```

- `dlarfp` computes a Householder reflector to annihilate the subdiagonal coefficients in the k-th column
- `dlarf` applies the Householder reflector to the trailing submatrix, i.e., computes $A_{k:m,k:n} = (I^{m-k+1} - \tau_k v_k v_k^T) A_{k:m,k:n}$
- factorization is in-place, i.e., V and R are stored in place of A

Other dense LA operations: the QR factorization

The `dlarfp` routine computes a Householder reflection $H = (I - \tau v v^T)$ with

$$v = \frac{x + sign(x(1))\|x\|_2 e_1}{x(1) + sign(x(1))\|x\|_2}, \quad \tau = \frac{sign(x(1))(x(1) - \|x\|_2)}{\|x\|_2}.$$

The `dlarf` routine **does not** explicitly compute the Householder matrix $H = (I - \tau v v^T)$ but rather it applies it as follows

$$(I - \tau v v^T)A = (A - \tau(v(v^T A)))$$

Exercise: What is the arithmetic intensity? Answ:

$$\left(\frac{4}{3}n^3\right) \ / \ (n^2) = O(n)$$

Other dense LA operations: the QR factorization

Theorem (Compact WY representation [4])

Let $Q = H_1 ... H_{k-1} H_k$, with $H_i \in \mathbb{R}^{m \times m}$ an Householder transformation

$$H_k = (I - \tau_k v_k v_k^T)$$

and $k \leq m$. Then, there exist an upper triangular matrix $T \in \mathbb{R}^{k \times k}$ and a matrix $V \in \mathbb{R}^{m \times k}$ such that

$$Q = I + V_k T_k V_k^T.$$

Where:

$$T_k = \begin{bmatrix} T_{k-1} & -\tau_k T_{k-1} V_{k-1}^T v_k \\ 0 & -\tau_k \end{bmatrix}, \quad V_k = \begin{bmatrix} V_{k-1} & v_k \end{bmatrix}$$

Other dense LA operations: the QR factorization

Take a matrix A of size $m \times n$ and partition it as below with A_{11} of size $b \times b$, $b \ll m, n$ and A_{22} of size $(m-b) \times (n-b)$
The basic step of the block QR factorization is

$$Q_1{}^T \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] = \left[\begin{array}{c|c} R_{11} & R_{12} \\ \hline & \tilde{A}_{22} \end{array}\right]$$

achieved through the following operations [4]

$$\left[\begin{array}{c} A_{11} \\ A_{21} \end{array}\right] \rightarrow \left[\begin{array}{c} V_{11} \\ V_{21} \end{array}\right], \; [R_{11}], \; [T_1]$$

$$\left[\begin{array}{c} R_{12} \\ \tilde{A}_{22} \end{array}\right] = \left(I - \left[\begin{array}{c} V_{11} \\ V_{21} \end{array}\right] \cdot [T_{11}^T] \cdot [V_{11}^T V_{21}^T]\right) \left[\begin{array}{c} A_{12} \\ A_{22} \end{array}\right]$$

Other dense LA operations: the QR factorization

This is (pretty much) the block QR factorization implemented in LAPACK [2]

```
subroutine dgeqrf
do k=1, n, b
   call dgeqrt (A(k:m,k:k+b-1), T)
   call dgemqrt(A(k:m,k:k+b-1), T, A(k:m,k+b:n))
end do
```

- `dgeqrt` computes the QR factorization of the $A_{k:m,k:k+b-1}$ block-column
- `dgemqrt` aplies the corresponding reflectors to $A_{k:m,k+b:n}$

Outline

Parallelism: definitions

Assume a problem of size n (sequential execution time is a linear function of n) and a parallel system with p processors

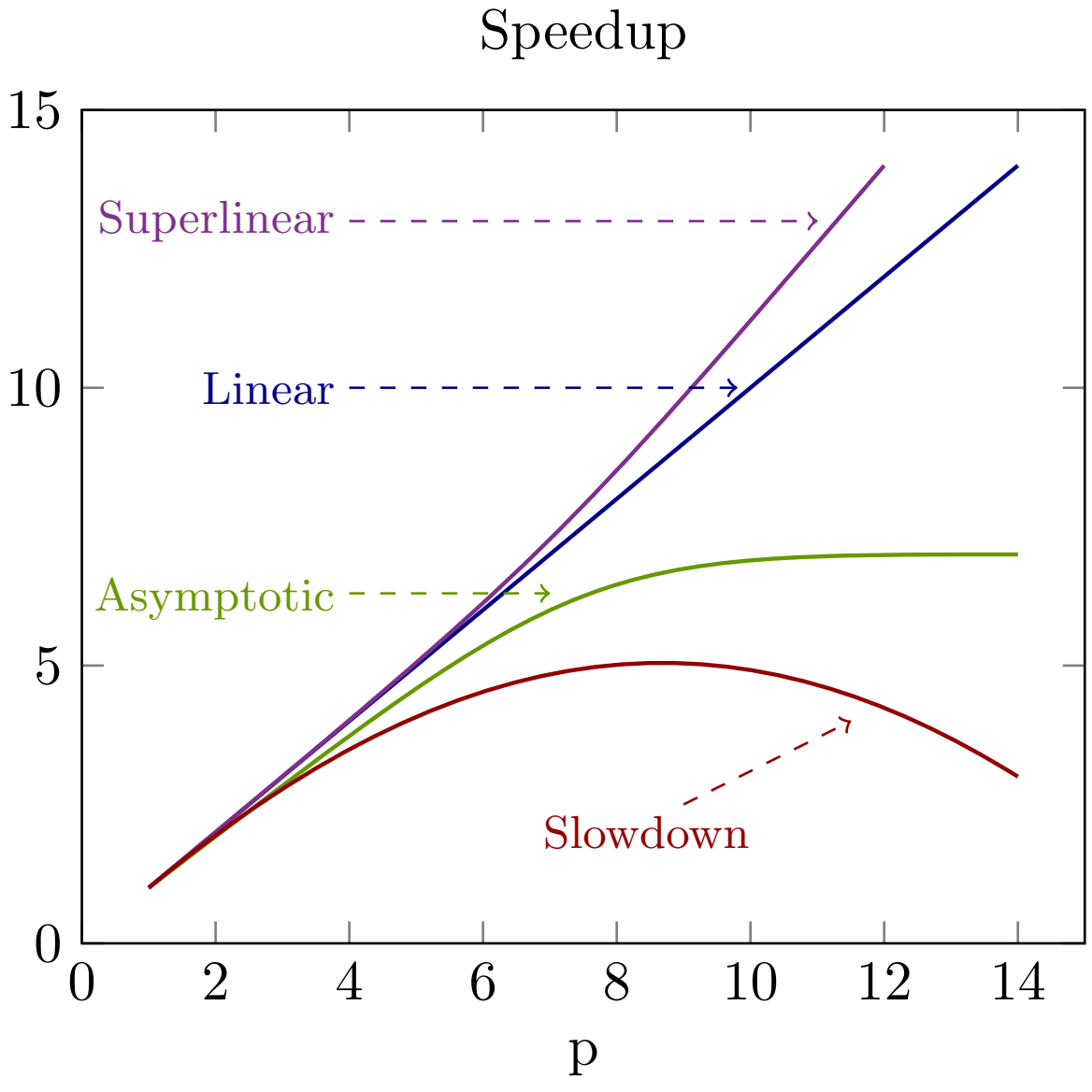
- $T(n, p)$ is the time needed to solve the problem using p processors. $T(n, 1)$ is the time for a sequential execution and is set as a reference for evaluating performance
- Speedup: the speedup measures the acceleration, i.e., the improvement with respect to the sequential case

$$S(n, p) = \frac{T(n, 1)}{T(n, p)}$$

- Efficiency: the efficiency measures the improvement with respect to the available resources

$$E(n, p) = \frac{S}{p} = \frac{T(n, 1)}{pT(n, p)}$$

Parallelism: speedup



A model for speedup and efficiency

Two very well known laws to evalulate performance can be formulated using the following model:

$$pT(n,p) = T(n,1) + O(n,p)$$

where $O(n,p)$ is the overhead associated with parallelism; this can be, for example, used to model the part of the workload which is not parallelizable or the communications.

Therefore

$$S(n,p) = \frac{pT(n,1)}{T(n,1) + O(n,p)} = \frac{p}{1 + \frac{O(n,p)}{T(n,1)}}$$

A model for speedup and efficiency

Speedup:

- Superlinear: $O(n,p)$ is a negative (decreasing) function of p. This is extremely rare in practice and may happen, e.g., due to larger aggregated cache size

- Linear: $O(n,p) = 0$. This is also very rare in parallel computing (less in Cloud computing) where processes cooperate for achieving a common task

- Asymptotic: $O(n,p)$ grows linearly with p. This may, e.g., correspond to the case where part of the workload is not parallelizable which is very common (see Amdahl's law in the next slide).

- Slowdown: $O(n,p)$ grows faster than p. This is the most common case.

Amdahl's and Gustafson's laws, iso-efficiency

- Amdahl's law: the workload can be split into a part which is parallelizable and one which is not. This amounts to saying that $O(n,p)/p$ has a constant value c as p increases. As a result, the speedup is upper-bounded by the value $T(n,1)/c$

- Gustafson's law: as p increases, the workload can be scaled up in such a way that the speedup grow linearly with p or, equivalently, the efficiency stays constant.

$$E(n,p) = \frac{1}{1 + \frac{O(n,p)}{T(n,1)}}$$

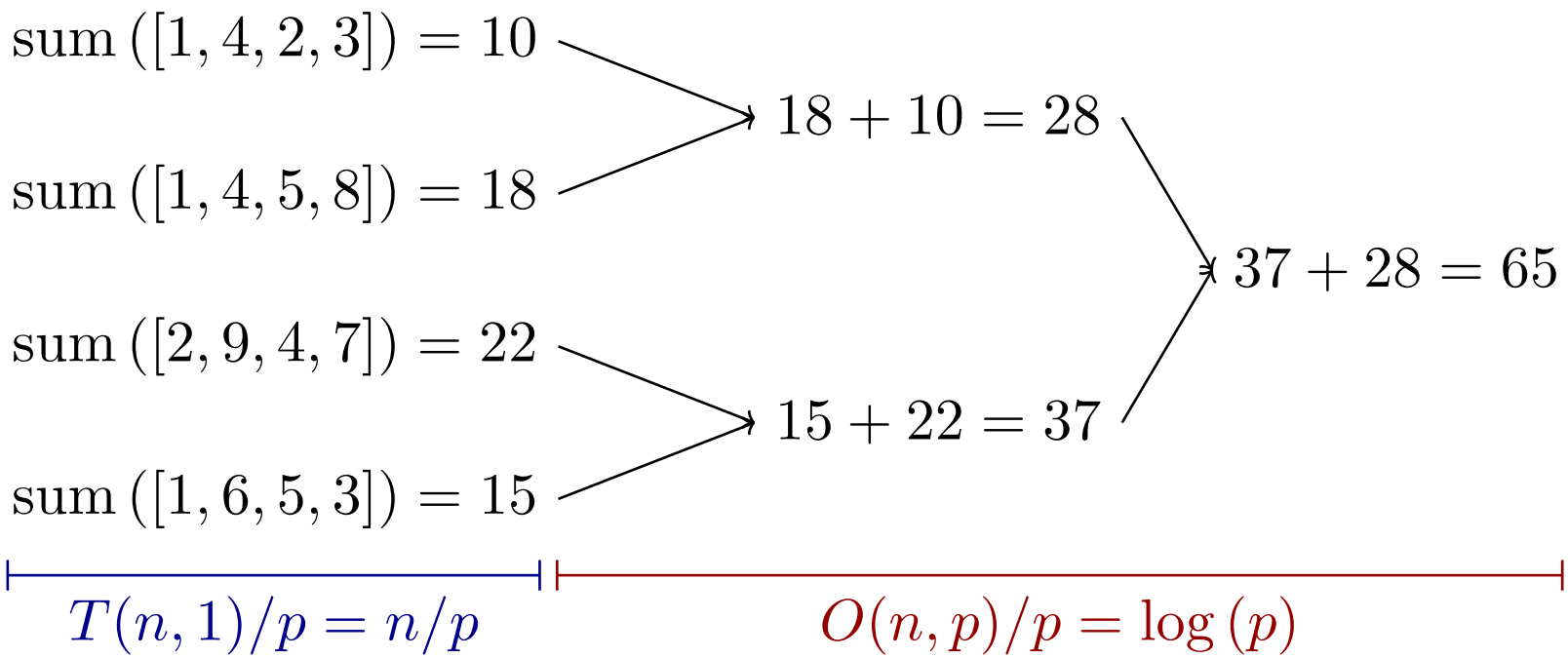
This is the so-called iso-efficiency function and tells us how much the workload should be scaled up to keep the efficiency constant

Iso-efficiency: example

Sum all the elements of an array x of size n. Algorithm: split the array in p parts, each processor computes the sum of one part, a binary reduction assembles all the partial sums

Example with $n = 16$ and $p = 4$:

$\text{sum}([1, 4, 2, 3, 1, 4, 5, 8, 2, 9, 4, 7, 1, 6, 5, 3]) = 65$



Because $O(n, p) = p\log(p)$, if the number of processors increases from p to p', n must increase of $(p'\log(p'))/(p\log(p))$

Scalability

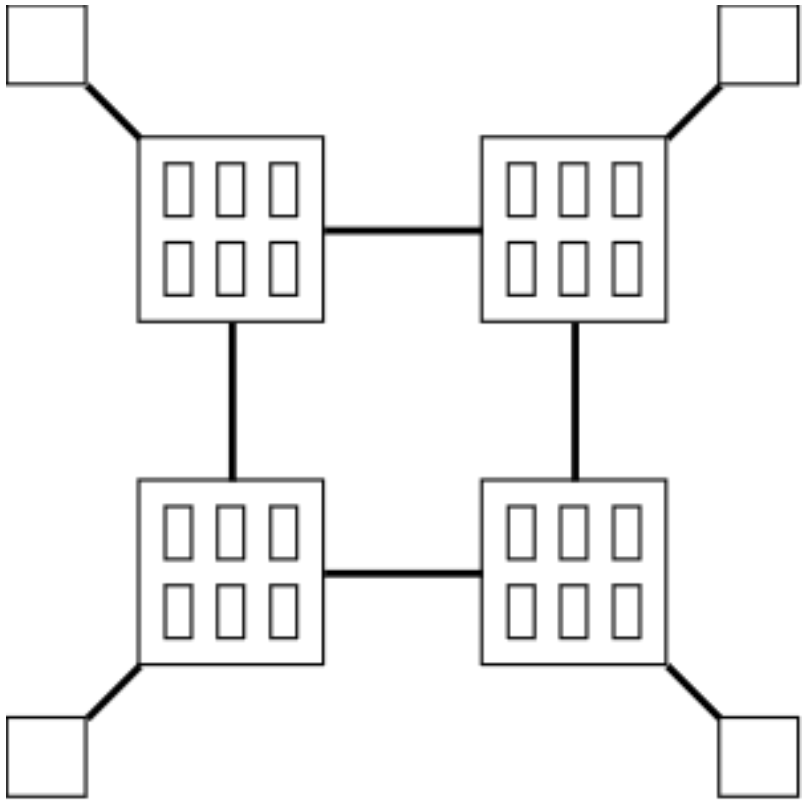
- Strong scalability: ability of an algorithm/code to reduce the execution time as p increases. This can only happen if $O(n, p)$ is constant, which never happens

- Weak scalability: ability of an algorithm/code to keep the efficiency constant as p increases by adjusting n. If n and, therefore, $T(n, 1)$ must be increased a lot (e.g., exponentially), the algorithm is not weakly scalable. If it is increased linearly, therefore $O(np, p)/T(np, 1)$ is constant, the algorithm is weakly scalable

Outline

The roofline model: multicores

Let's assume a system with 4 AMD Opteron 8431 processors in a non-uniform memory access (NUMA) configuration



This means that, although each core can access directly data in any memory bank, the efficiency of memory transfers will depend on distance and contention

The roofline model: multicores

Assume the algorithm is executed in parallel by p processes, each running on a different core:

$$t = \max_{i=0}^{p-1} \left(t^i\right) = \max_{i=0}^{p-1} \left(\max \left(\frac{W^i}{s^i}, \frac{D^i}{b^i} \right) \right)$$

i.e., the total execution time corresponds to the time taken by the slowest and/or most loaded process.

Assume the algorithm is embarrassingly parallel: each processes

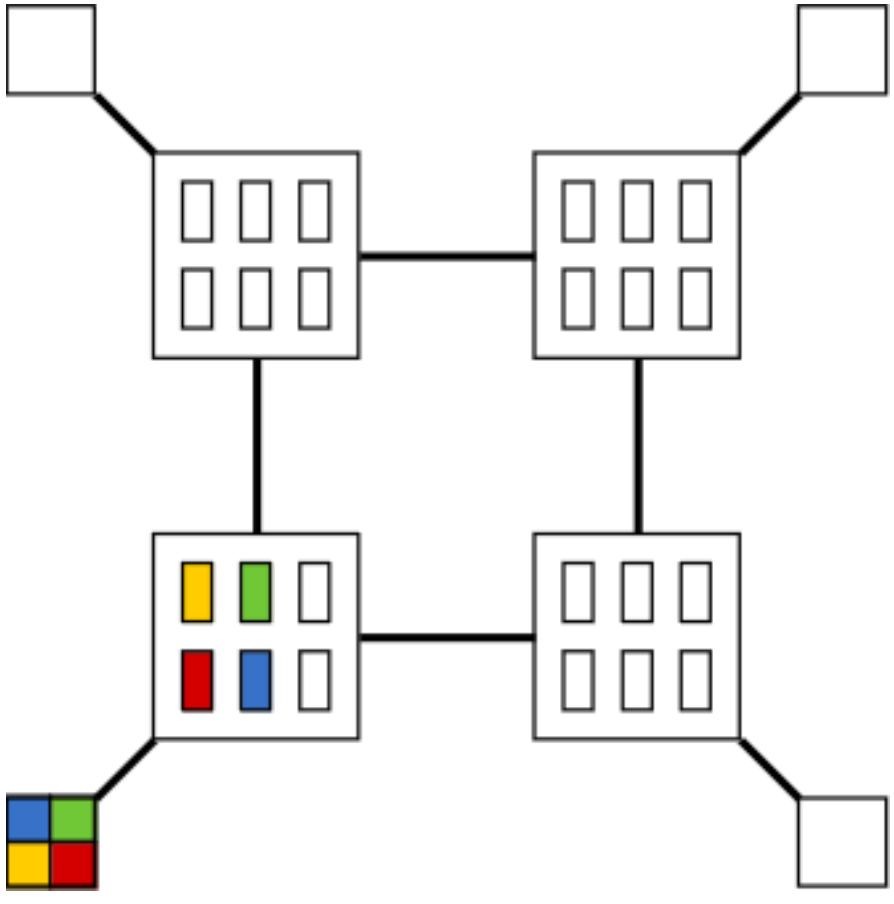
- executes $W^i = W/p$ operations
- and uses $D^i = D/p$ data

$$t = \max_{i=0}^{p-1} \left(t^i\right) = \max_{i=0}^{p-1} \left(\max \left(\frac{W}{ps^i}, \frac{D}{pb^i} \right) \right)$$

The roofline model: multicores

Case 1: bad data placement

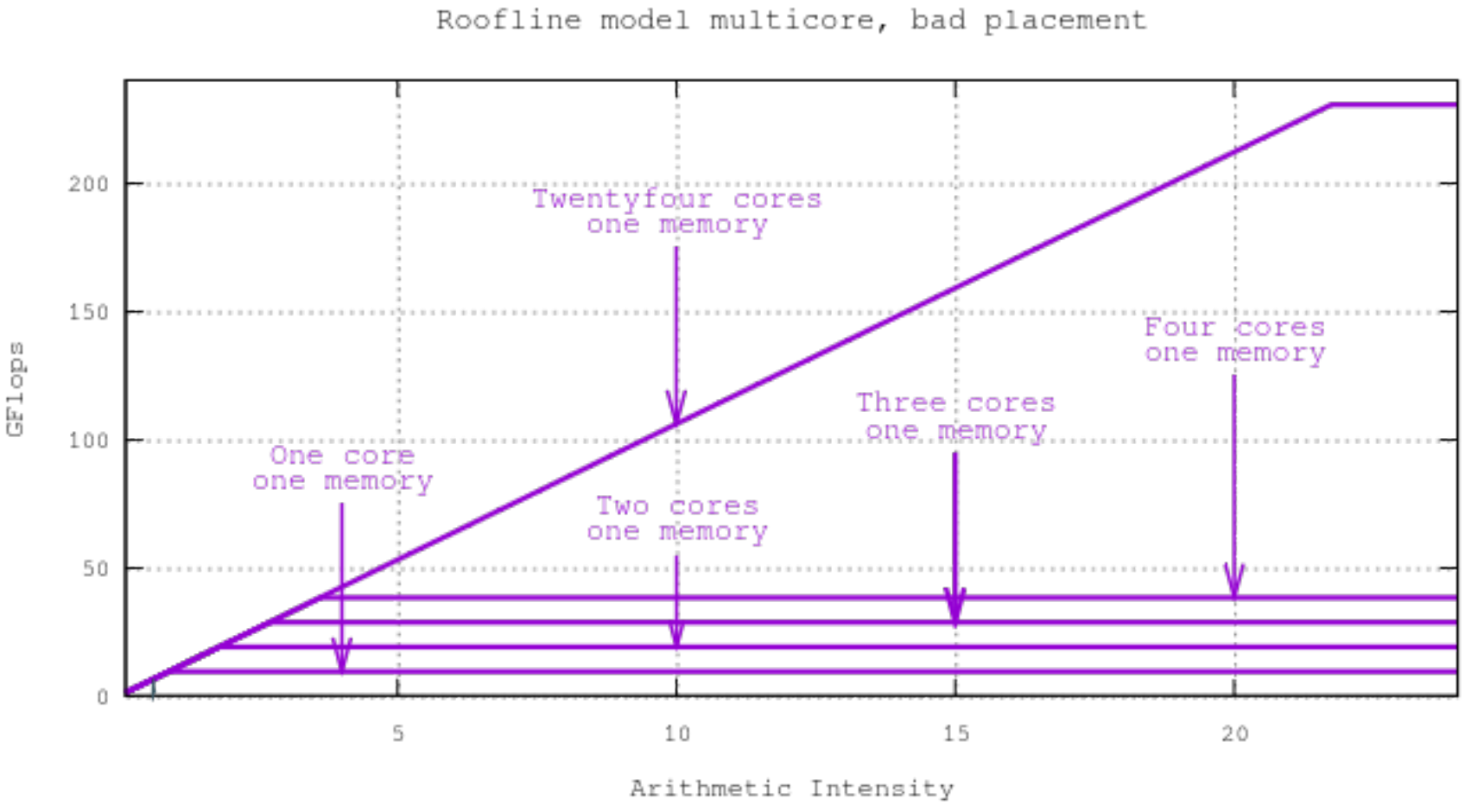
In this case all the processes read/write data from/to the same memory and, thus, the bandwidth is shared among all the processes. Therefore for each process the bandwidth is $b^i = b/p$



$$t = \max_{i=0}^{p-1} \left(\max \left(\frac{W}{ps^i}, \frac{D}{pb^i} \right) \right) = \max \left(\frac{W}{ps}, \frac{Dp}{pb} \right)$$

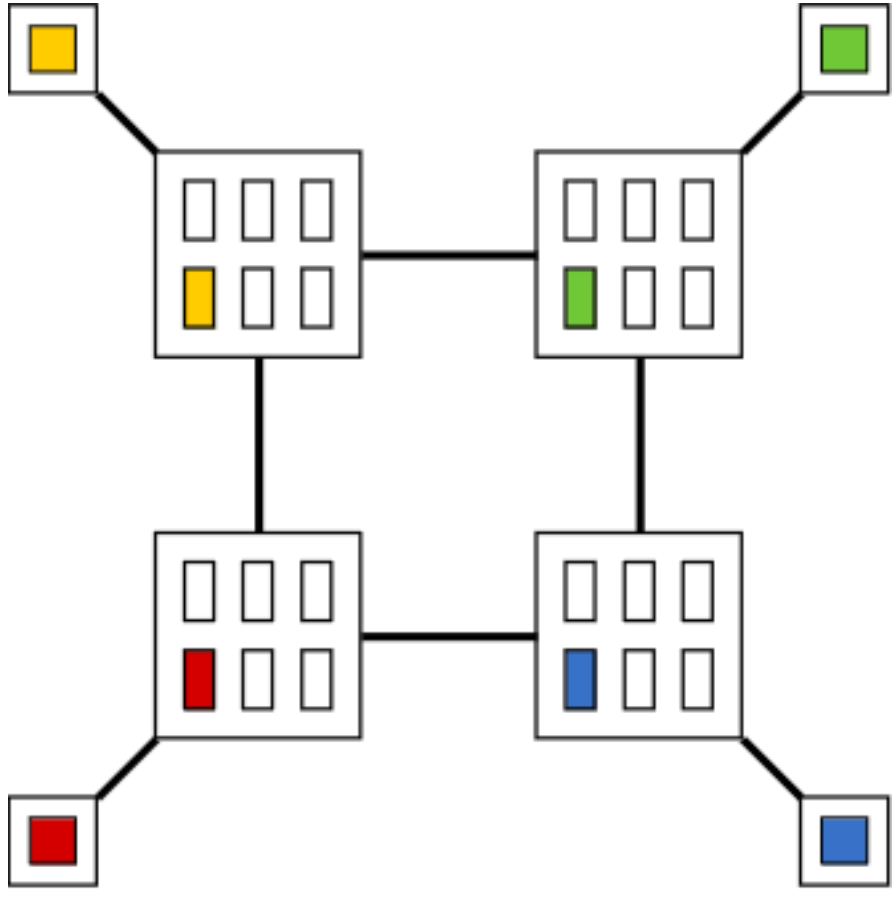
The roofline model: multicores

Case 1: bad data placement



Case 2: good data placement

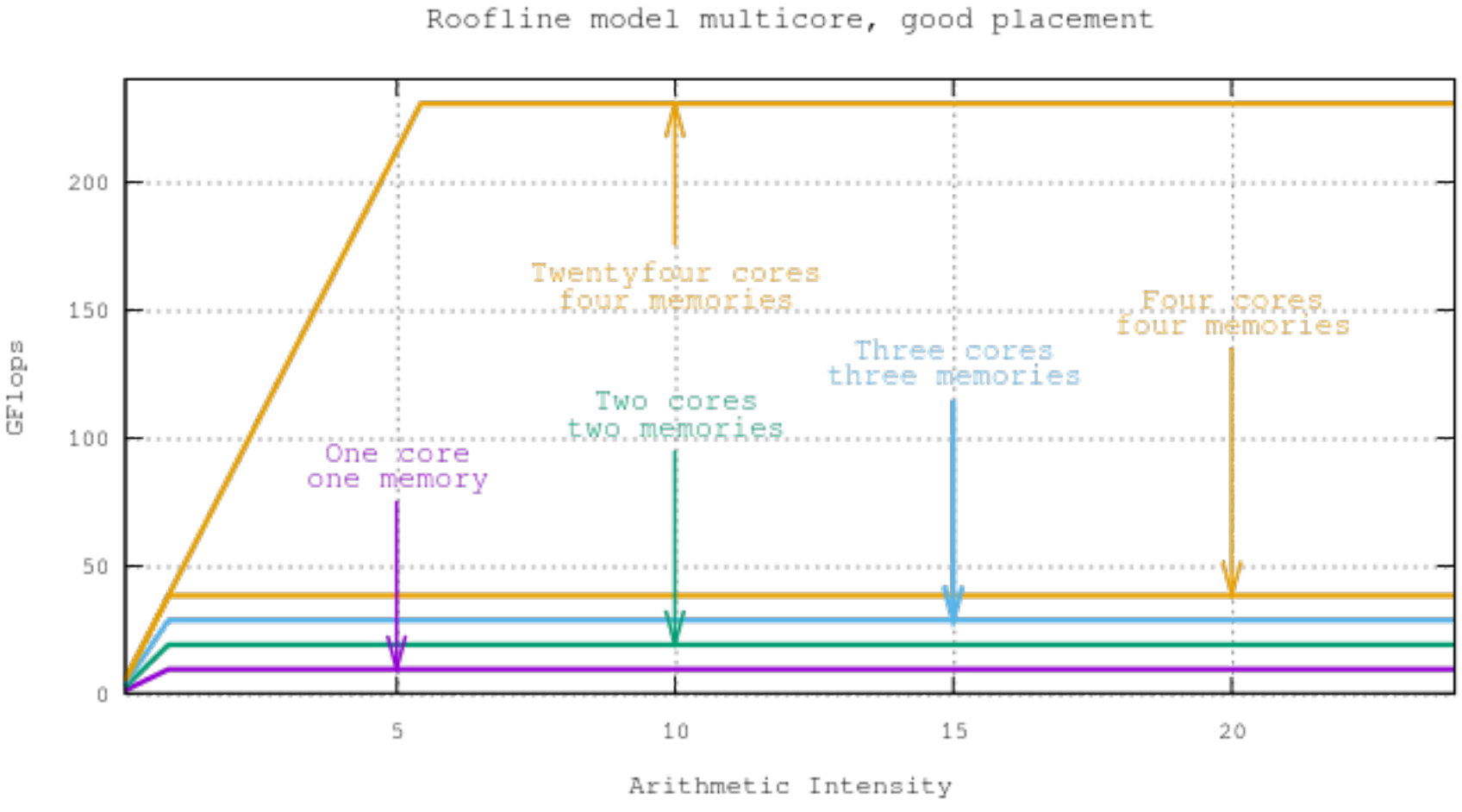
In this case each process reads/writes data from a different memory and, thus, has bandwidth $b^i = b$. If p is greater than the number of sockets, the bandwidth will be somehow shared depending on the placement of data and processes.



$$t = \max_{i=0}^{p-1} \left(\max \left(\frac{W}{ps^i}, \frac{D}{pb^i} \right) \right) = \max \left(\frac{W}{ps}, \frac{D}{pb} \right)$$

The roofline model: multicores

Case 2: good data placement

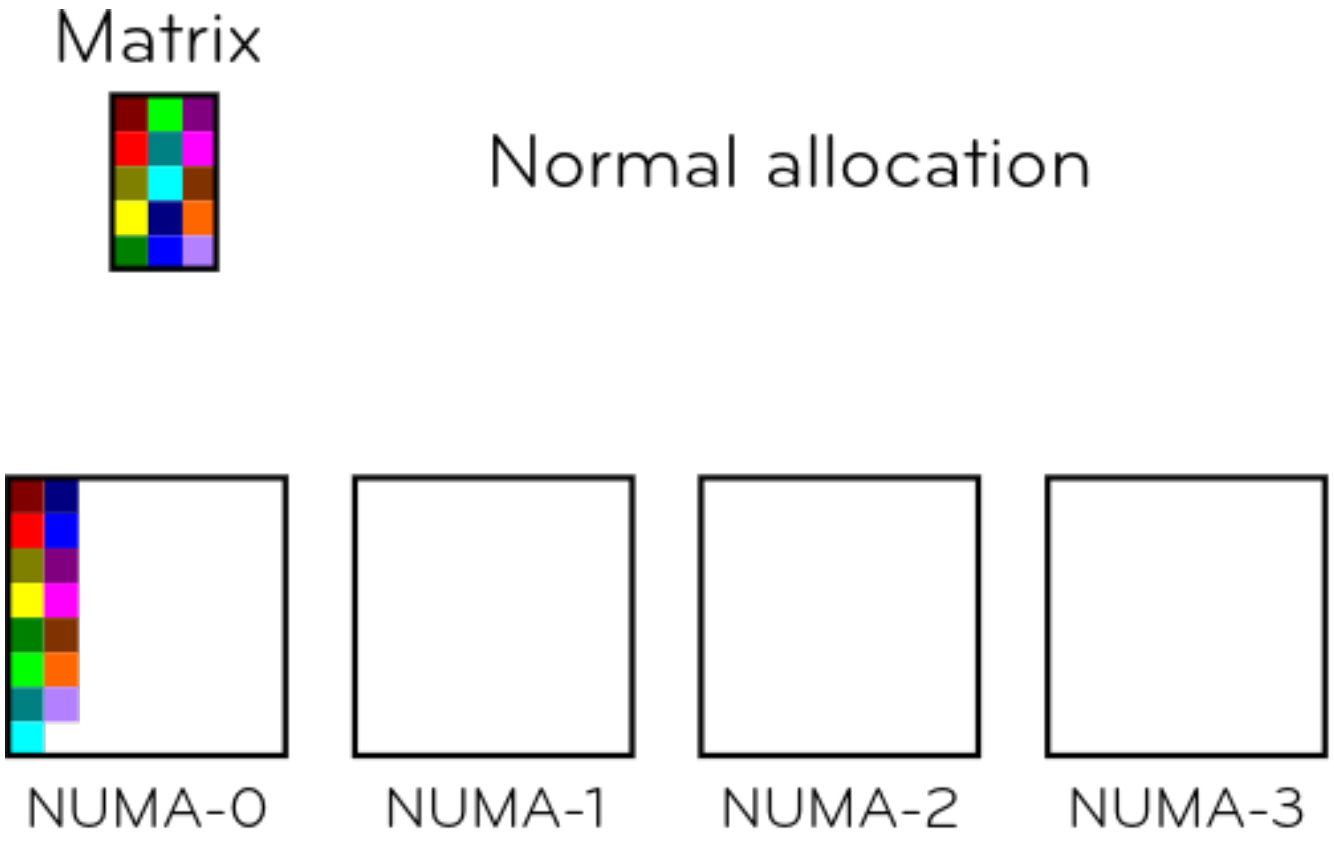


Threads and data placement

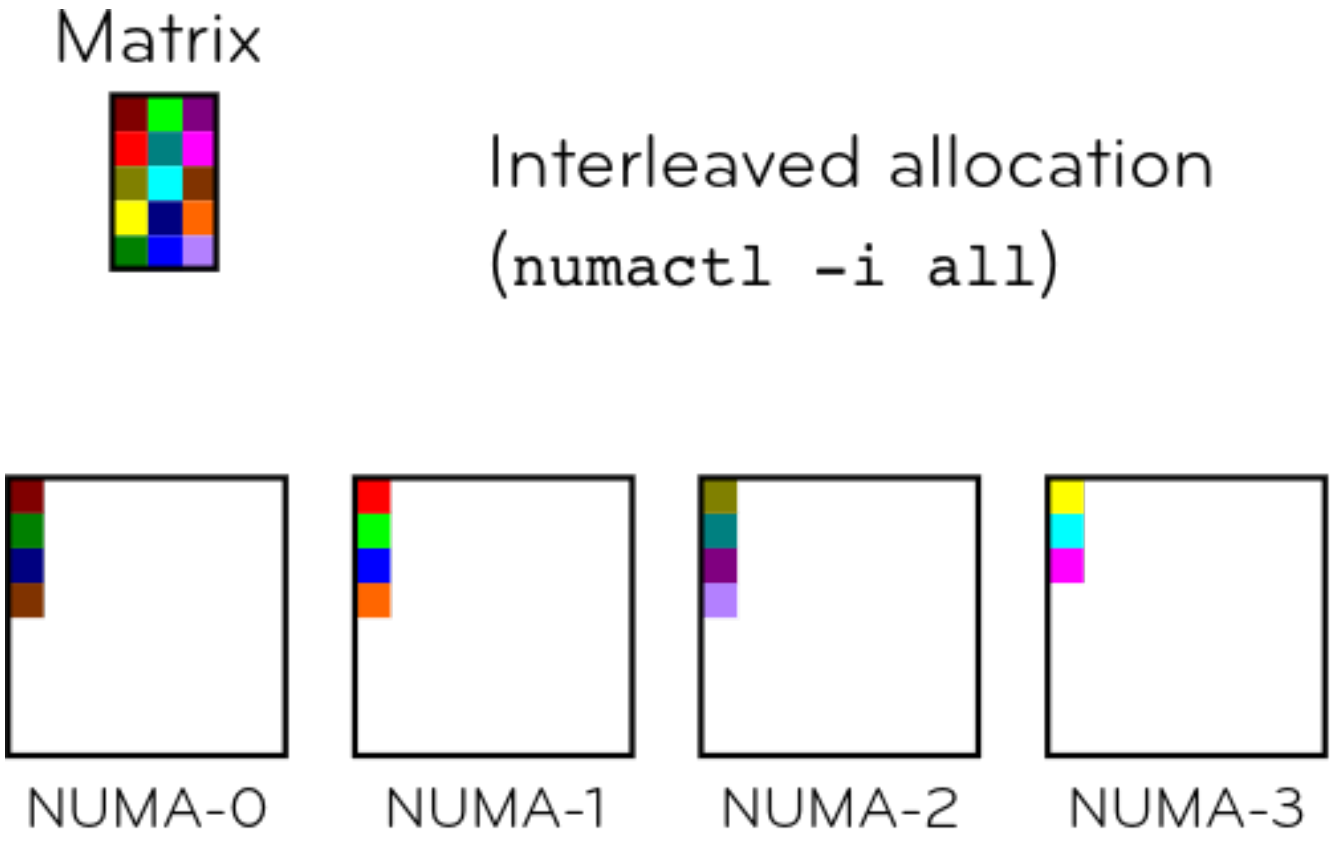
- threads: controlling the placement of threads depends on the threading technology in use: for example, with OpenMP the `OMP_PROC_BIND` and `OMP_PLACES` can be used.

- data: the data placement can be controlled in (at least) three different ways:
  - implicit: through the first touch rule (e.g., in linux). Data is allocated in the memory module closer to the first thread that touches it
  - explicit: using libraries such as `numactl`
  - interleaved: allocated memory is spread over multiple memory modules in pages in a round-robin fashion

Interleaved memory allocation



Interleaved memory allocation

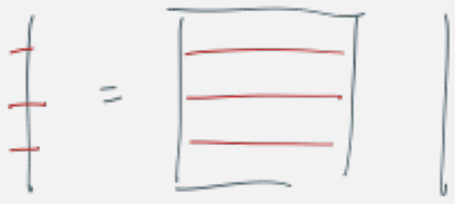


Memory interleaving should provide a more uniform distribution of data that (supposedly) reduces conflicts and increases the average memory bandwidth

Multithreaded BLAS routines

Assume **gemv** computes a sequential matrix-vector product

```
init_x( x(1:n) )
init_y( y(1:m) )
init_a( a(1:m,1:n) )

!$omp parallel
iam = omp_get_thread_num()
nth = omp_get_num_threads()
b   = m/nth

y(iam*b+1:(iam+1)*b-1) = gemv( a(iam*b+1:(iam+1)*b-1,1:n
    ), x(1:n) )
!$omp end parallel
```



in this case the placement is bad because one thread initializes all the data which is thus placed next to it

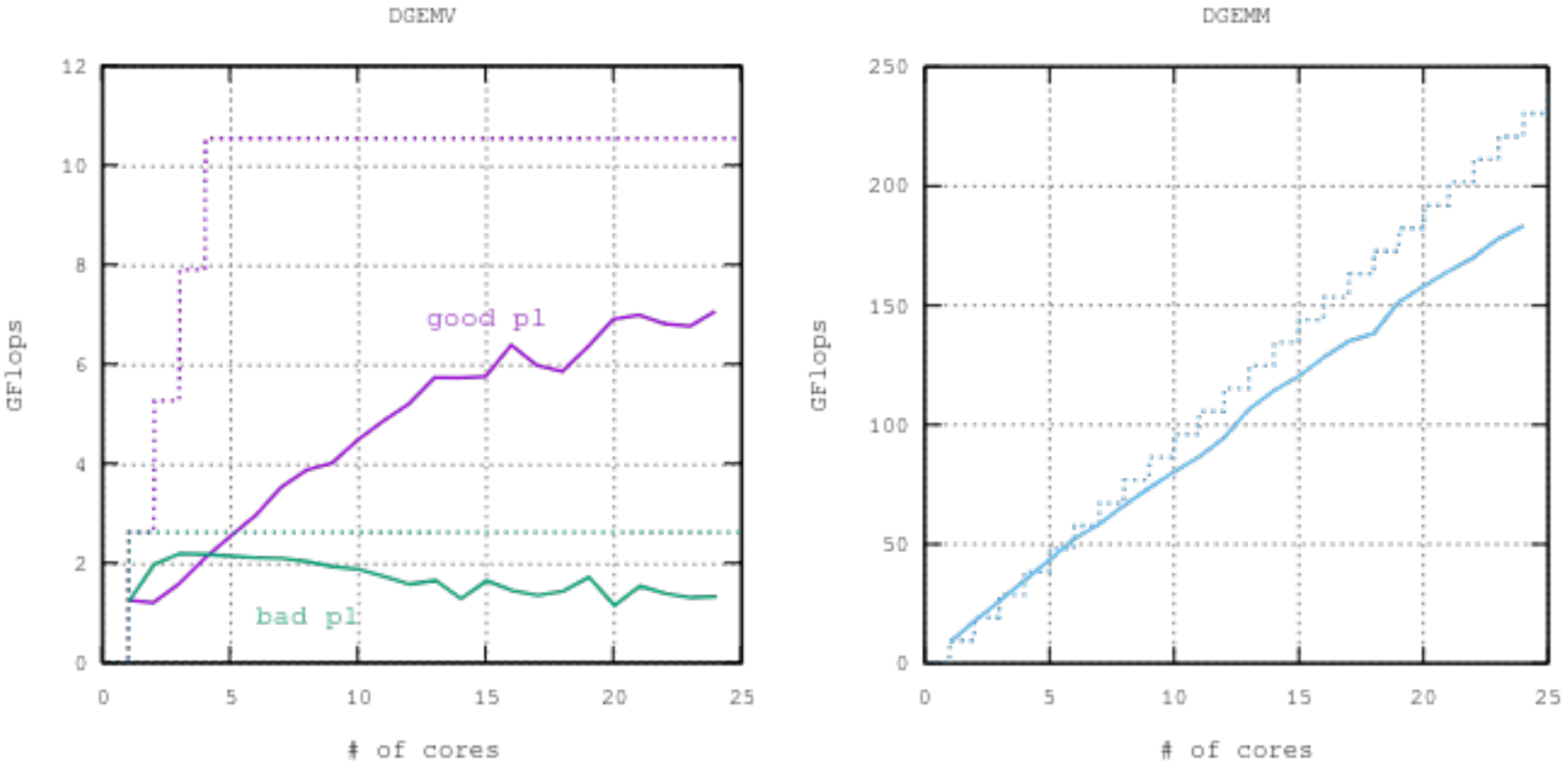
Multithreaded BLAS routines

Assume `gemv` computes a sequential matrix-vector product

```
!$omp parallel
iam = omp_get_thread_num()
nth = omp_get_num_threads()
b   = m/nth

!$omp single
init_x( x(1:n) )
!$omp end single

init_y( y(iam*b+1:(iam+1)*b-1) )
init_a( a(iam*b+1:(iam+1)*b-1,1:n) )

y(iam*b+1:(iam+1)*b-1) = gemv( a(iam*b+1:(iam+1)*b-1,1:n
    ), x(1:n) )
!$omp end parallel
```

Here each thread initializes the part of `a` and `y` which it is going to use

Multithreaded BLAS routines

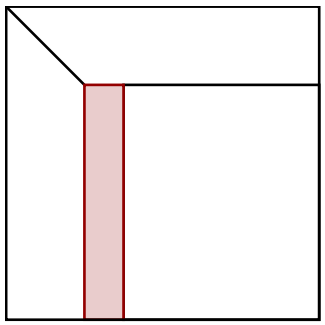


If the data is not correctly placed, it doesn't help much to use more cores for memory bound operations like `Level-2` BLAS. In any case performance is limited by the (aggregated) bandwidth for memory-bound operations.

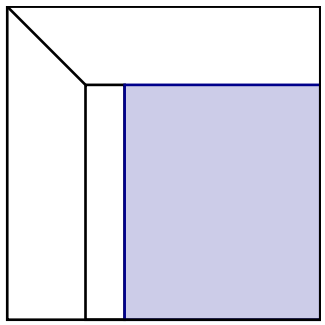
Parallel dense matrix factorizations

Take the block LU factorization (implemented in the `dgetrf` routine). This can be described as an alternating sequence of a

- panel operation which factorizes a block-column of size $(n-(k-1)b) \times b$ (this is the `dgetrf2` routine) and an...
- update operation which applies the corresponding transformation to the trailing submatrix of size $(n-(k-1)b) \times (n-(k-1)b)$ (this is the ensemble of the `dswap`, `dtrsm` and `dgerk` routines).



Panel



Update

Parallel dense matrix factorizations

- panel is mostly based on BLAS-2 operations and, thus, poorly or not parallelizable
- update is based on BLAS-3 operations (`dtrsm` and `dgerk` or `dgemm`) and, thus, very efficiently parallelizable

Exercise: what is the cost of the panel and update operations ?
Answer (excluding some lower order terms):

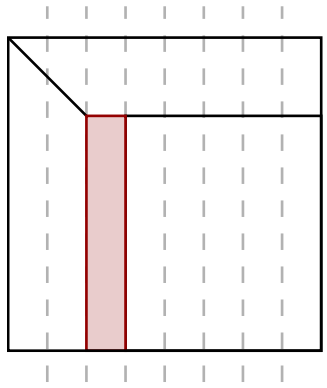
- panel at step k: $(n-(k-1)b)\,b^2 - \frac{1}{3}b^3$
- update at step k: $2\,(n-kb)^2\,b$
- all panels: $n^2 b - \frac{n^2}{2} - \frac{nb^2}{3}$
- all updates: $\frac{2}{3}n^3$

Exercise: use the above results to apply Amdahl's law to the blocked LU factorization.

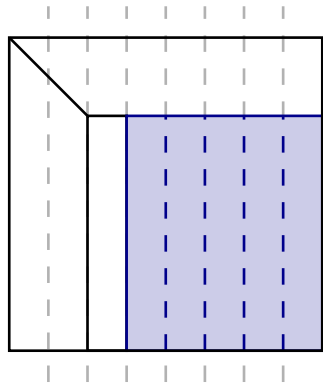
Parallel dense matrix factorizations

A naive approach to parallelizing the LU and QR factorization:

- choose b as small as possible without degrading the performance of operations
- partition the matrix into block-columns of size b
- at each step, execute updates on each block-column in parallel



Panel



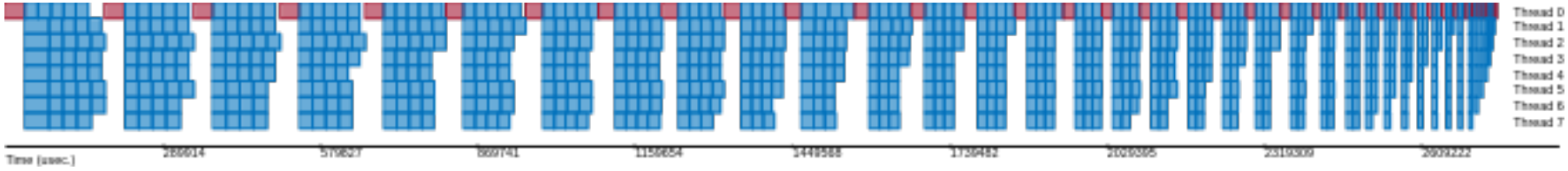
Update

Parallel dense matrix factorizations

The actual code may look something like this (`A[:,k]` denotes block-column k and $N = n/b$):

```
do k=1, N
   call panel (A[:,k])
   !$omp parallel do
   do j=k+1, N
      call update(A[:,k], A[:,j])
   end do
end do
```

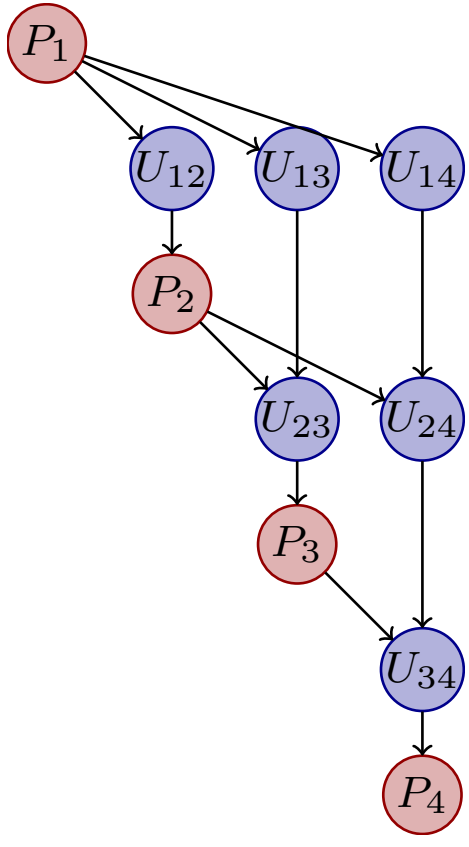
this is commonly referred to as a *fork-join* parallelization because it is an interleaved sequence of sequential and parallel regions



The main limit of the fork-join model is that in sequential regions all cores except on are idle because of the synchronization in the for and join

Parallel dense matrix factorizations

A better parallelization can be achieved if the previous algorithm is represented in the for of a Directed Acyclic Graph (DAG) of tasks: each node represent and operation/task and each edge a dependency. A dependency $A \rightarrow B$ means that task B depends on task A, i.e., it cannot be started before A is finished (for example because it needs data produced by A)

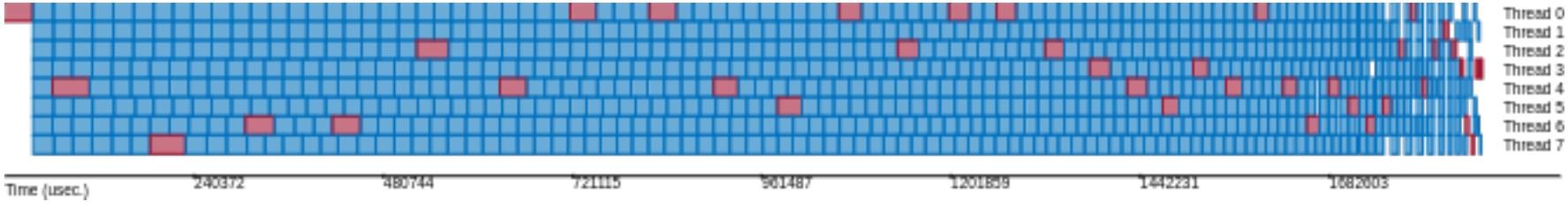


In this case, the DAG reveals that a panel operation P_i can be executed as soon as update $U_{i-1,i}$ is finished even if all updates $U_{i-1,j}, j > i$ are not done. This technique is called lookahead

Parallel dense matrix factorizations

Lookahead can be used with a fixed depth (i.e., no more than d factorization steps are pipelined) or arbitrary depth (i.e., a panel is executed as soon as its ready.

```
!$omp parallel
!$omp master
do k=1, N
   !$omp task depend(inout:A[:,k]) priority(2)
   call panel (A[:,k])
   do j=k+1, N
      !$omp task depend(in:A[:,k]) depend(inout:A[:,j])
    priority(1)
      call update(A[:,k], A[:,j])
   end do
end do
```

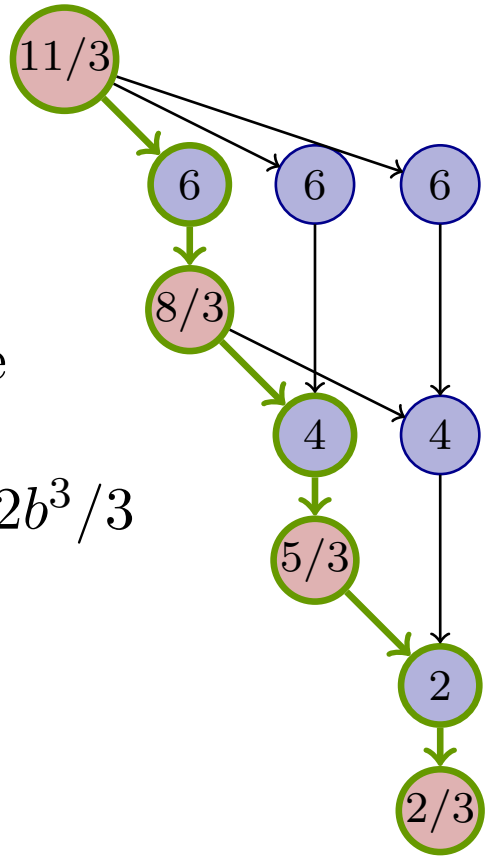


Parallel dense matrix factorizations

What is the shortest time I can achieve with this approach?
Critical Path Analysis: the shortest execution time is the time needed to traverse the longest path in the DAG.

Assume $n = N * b$ the time of tasks (for LU) is

- time(P_k) : $(N - (k-1) - 1/3)b^3$
- time(U_{k*}) : $2(N-k)b^3$



The critical path is the one that connects all the panel operations. Its length is
$T(n,\infty) = (11/3+6+8/3+4+5/3+2+2/3)b^3 = 62b^3/3$
Therefore, in this case, the highest speedup is

$$S(n,\infty) = \frac{T(n,1)}{T(n,\infty)} = \frac{110b^3/3}{62b^3/3} = 1.77$$

For a matrix of size $n \times n = N * b \times N * b$ the speedup is

$$S(n,\infty) = \frac{\sum_{k=1}^{N} \text{time}(P_k) + (N-k)\text{time}(U_{k*})}{\sum_{k=1}^{N} \text{time}(P_k) + \text{time}(U_{k*})} \simeq \frac{4}{9}N$$

Parallel dense matrix factorizations

Assume that matrix A is split into blocks of size $b \times b$ and `A[i,j]` identifies block (i, j); rewrite the Cholesky factorization like this:

```
subroutine dpotrf(a, n)
do k=1, N
   call dpotf2(A[k,k]))
   do i=k+1, N
      call dtrsm(A[i,k], A[k,k])
      do j=k+1, i
         call dgemm(A[i,k], A[j,k], A[i,j])
      end do
   end do
end do
```

- `dpotf2(A[k,k]))` computes the Cholesky factorization of a block and costs $1/3b^3$
- `dtrsm(A[i,k], A[k,k])` computes $A_{ik}A_{kk}^{-T}$ (here A_{kk} denotes the L_{kk} triangular block resulting from the `dpotf2`) and costs b^3
- `dgemm(A[i,k], A[j,k], A[i,j])` computes $A_{ij} = A_{ij} - A_{ik} * A_{jk}^T$ and costs $2b^3$

Exercise: compute the critical path

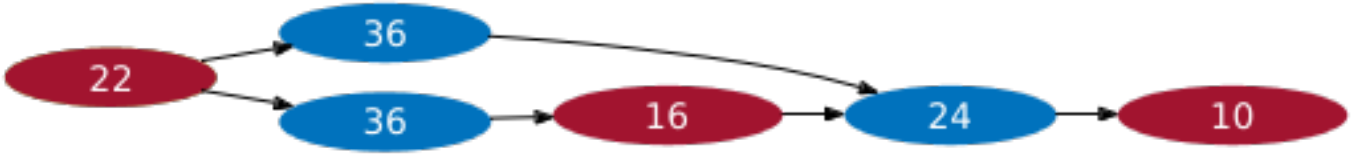
Parallel dense matrix factorizations

Apply the critical path analysis to the QR factorization of a $m \times n = M * b \times N * b$ matrix with $m \geq n$ knowing that

- $\text{time}(P_k)$: $2(M - k - 1/3)b^3$
- $\text{time}(U_{k*})$: $4(M - k - 1)b^3$

$$S(m, n, \infty) = \frac{\sum_{k=1}^{N} \text{time}(P_k) + (N - k)\text{time}(U_{k*})}{\sum_{k=1}^{N} \text{time}(P_k) + \text{time}(U_{k*})} \simeq \frac{2M - \frac{2}{3}N}{6M - 3N} N$$

Assuming $M = 4$ and $N = 3$:



Think of iso-efficiency: what happens if we only increase M?

Parallel dense matrix factorizations

Resume:

- Panel operations need access to the whole column:
  - in the LU factorization for searching the pivot
  - in the QR factorization for computing the column norm for the Householder reflection
- this makes the panel operation not (or hardly) parallelizable
- fork-join parallelization suffers from synchronisations and does not fully take advantage of parallelism
- lookahead or task-based approaches make better use of concurrency
- iso-efficiency is hard to achieve, especially in the case of overdetermined matrices (i.e., $m \gg n$) where the panel operations account for a large portion of the execution time

Can we break down the panel operation in such a way that it does not need access to whole columns at once?

Tiled QR factorization

In the tiled QR factorization the matrix is decomposed into blocks (or tiles) of size $b \times b$. The sub-diagonal blocks in a column are annihilated in successive steps.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix}$$

Tiled QR factorization

In the tiled QR factorization the matrix is decomposed into blocks (or tiles) of size $b \times b$. The sub-diagonal blocks in a column are annihilated in successive steps.

$$A_{11} = Q_{11}^{11} R_{11}^{11}$$

$$\begin{bmatrix} {Q_{11}^{11}}^T & & & \\ & I & & \\ & & I & \\ & & & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix} = \begin{bmatrix} R_{11}^{11} & A_{12}^{11} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix}$$

Tiled QR factorization

In the tiled QR factorization the matrix is decomposed into blocks (or tiles) of size $b \times b$. The sub-diagonal blocks in a column are annihilated in successive steps.

$$\begin{bmatrix} R_{11}^{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} Q_{11}^{21} & Q_{12}^{21} \\ Q_{21}^{21} & Q_{22}^{21} \end{bmatrix} \begin{bmatrix} R_{11}^{21} \\ \end{bmatrix}$$

$$\begin{bmatrix} {Q_{11}^{21}}^T & {Q_{21}^{21}}^T & & \\ {Q_{12}^{21}}^T & {Q_{22}^{21}}^T & & \\ & & I & \\ & & & I \end{bmatrix} \begin{bmatrix} R_{11}^{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix} = \begin{bmatrix} R_{11}^{21} & A_{12}^{21} \\ & A_{22}^{21} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix}$$

Tiled QR factorization

In the tiled QR factorization the matrix is decomposed into blocks (or tiles) of size $b \times b$. The sub-diagonal blocks in a column are annihilated in successive steps.

$$\begin{bmatrix} R_{11}^{21} \\ A_{31} \end{bmatrix} = \begin{bmatrix} Q_{11}^{31} & Q_{12}^{31} \\ Q_{21}^{31} & Q_{22}^{31} \end{bmatrix} \begin{bmatrix} R_{11}^{31} \\ \end{bmatrix}$$

$$\begin{bmatrix} {Q_{11}^{31}}^T & & {Q_{21}^{31}}^T & \\ & I & & \\ {Q_{12}^{31}}^T & & {Q_{22}^{31}}^T & \\ & & & I \end{bmatrix} \begin{bmatrix} R_{11}^{21} & A_{12}^{21} \\ & A_{22}^{21} \\ A_{31} & A_{32} \\ A_{41} & A_{42} \end{bmatrix} = \begin{bmatrix} R_{11}^{31} & A_{12}^{31} \\ & A_{22}^{21} \\ & A_{32}^{31} \\ A_{41} & A_{42} \end{bmatrix}$$

Tiled QR factorization

In the tiled QR factorization the matrix is decomposed into blocks (or tiles) of size $b \times b$. The sub-diagonal blocks in a column are annihilated in successive steps.

$$\begin{bmatrix} R_{11}^{31} \\ A_{41} \end{bmatrix} = \begin{bmatrix} Q_{11}^{41} & Q_{12}^{41} \\ Q_{21}^{41} & Q_{22}^{41} \end{bmatrix} \begin{bmatrix} R_{11}^{41} \\ \end{bmatrix}$$

$$\begin{bmatrix} {Q_{11}^{41}}^T & & {Q_{21}^{41}}^T \\ & I & \\ & & I \\ {Q_{12}^{41}}^T & & {Q_{22}^{41}}^T \end{bmatrix} \begin{bmatrix} R_{11}^{31} & A_{12}^{31} \\ & A_{22}^{21} \\ & A_{32}^{31} \\ A_{41} & A_{42} \end{bmatrix} = \begin{bmatrix} R_{11}^{41} & R_{12}^{41} \\ & A_{22}^{21} \\ & A_{32}^{31} \\ & A_{42}^{41} \end{bmatrix}$$

Tile QR factorization

$$\tilde{R},\ Q = \texttt{dtsqrt}(R,\ A),\ \text{ where } Q\tilde{R} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} \tilde{R}_1 \\ \end{bmatrix} = \begin{bmatrix} R \\ A \end{bmatrix}$$

Computes the QR factorization of a matrix formed by a triangle on top of a square. The Q matrix is stored implicitly in the form of Householder reflectors.

$$\tilde{B},\ \tilde{C} = \texttt{dtsmqrt}(B,\ C,\ Q),$$

$$\text{where } \begin{bmatrix} \tilde{B} \\ \tilde{C} \end{bmatrix} = Q^T \begin{bmatrix} B \\ C \end{bmatrix} = \begin{bmatrix} Q_{11}^T & Q_{21}^T \\ Q_{12}^T & Q_{22}^T \end{bmatrix} \begin{bmatrix} B \\ C \end{bmatrix}$$

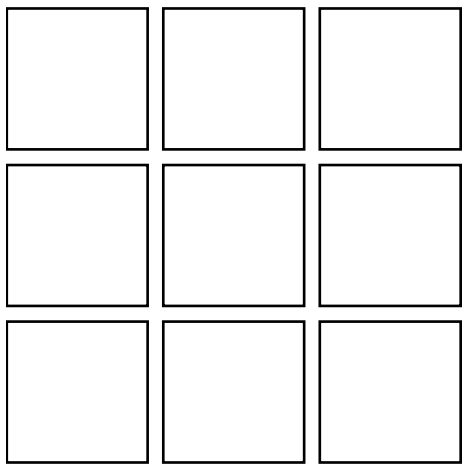
Applies the transformation computed by `dtsqrt` to a matrix formed by two squares, one on top of the other.

Both routines work in-place (i.e., the output overwrites the input) and are *structured* (i.e., skip computations with the zeros).

Tile QR

The (sequential) tile QR factorization proceeds like this

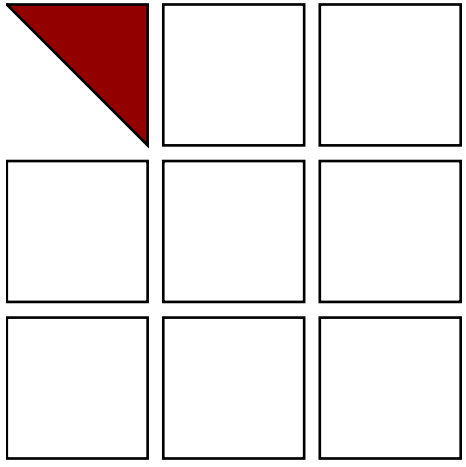
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

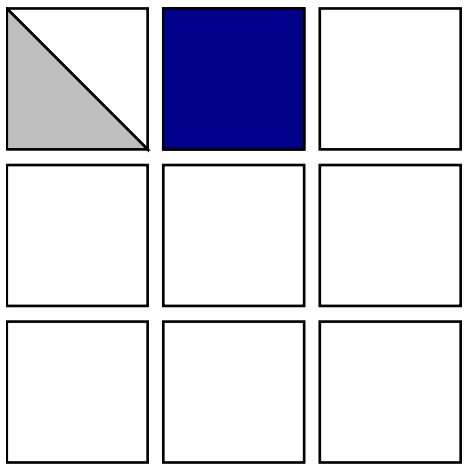
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

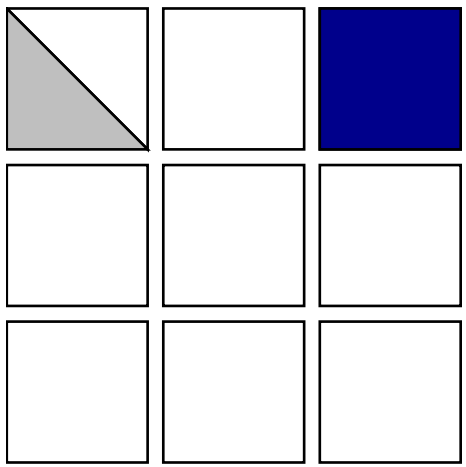
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

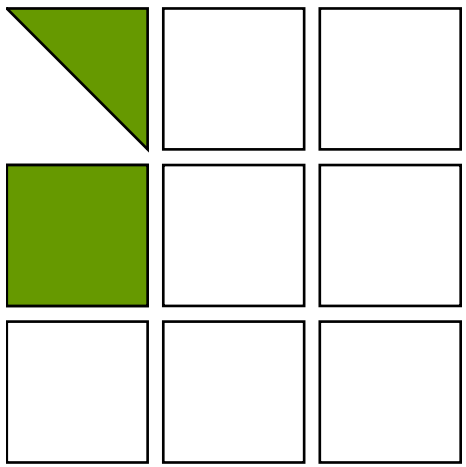
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

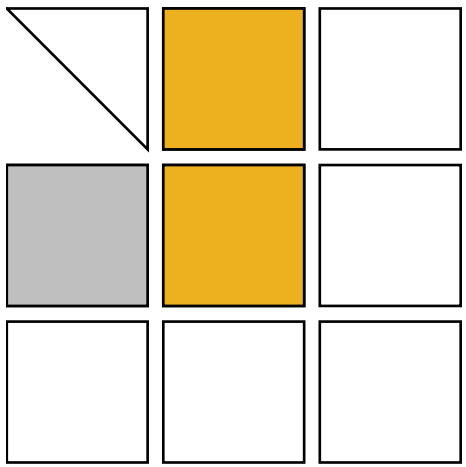
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

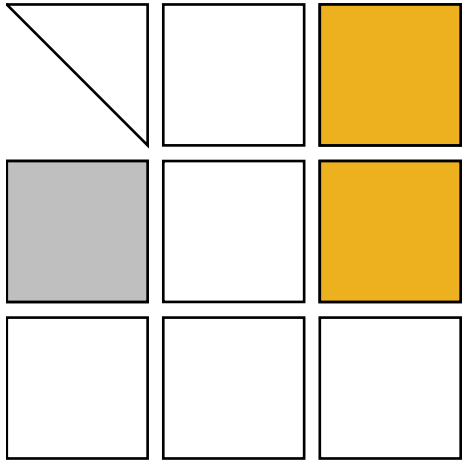
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

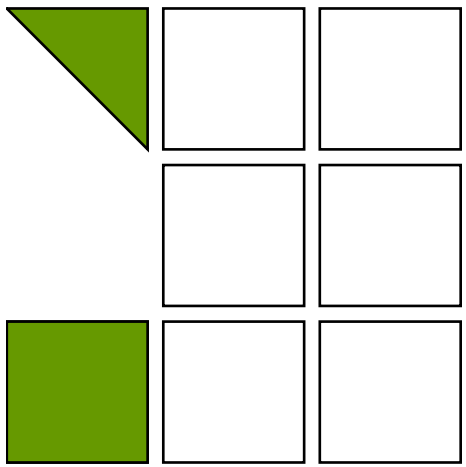
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

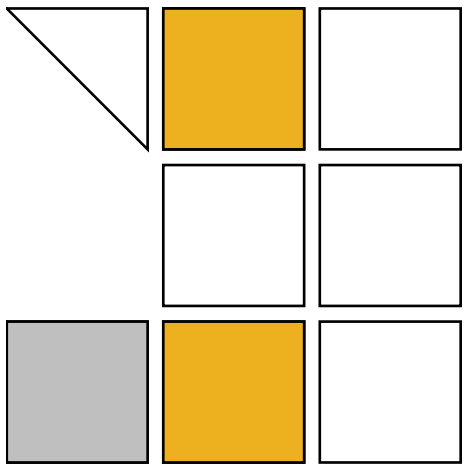
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

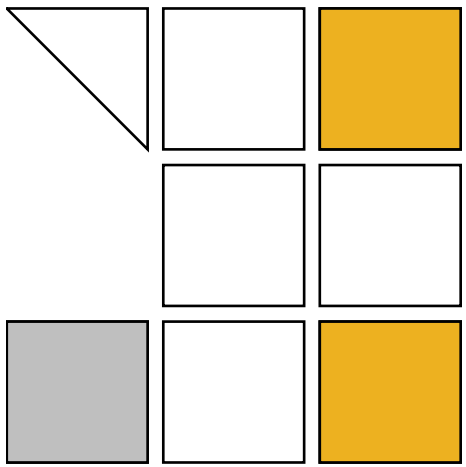
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

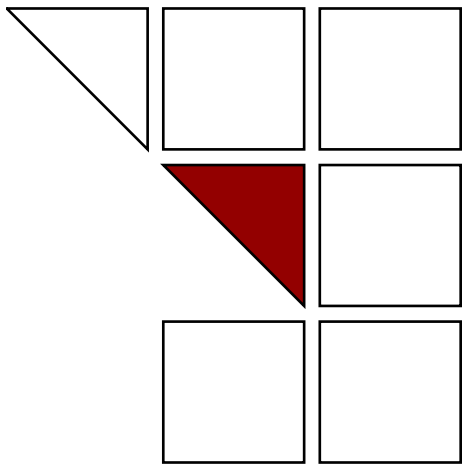
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

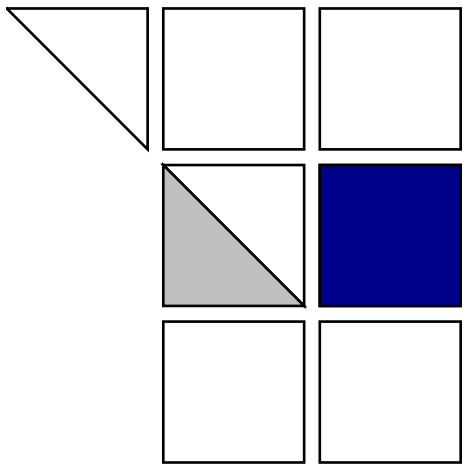
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

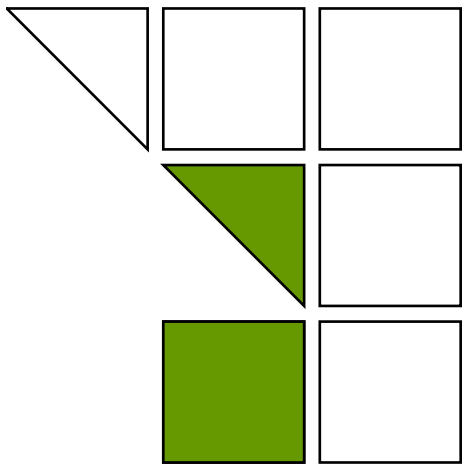
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

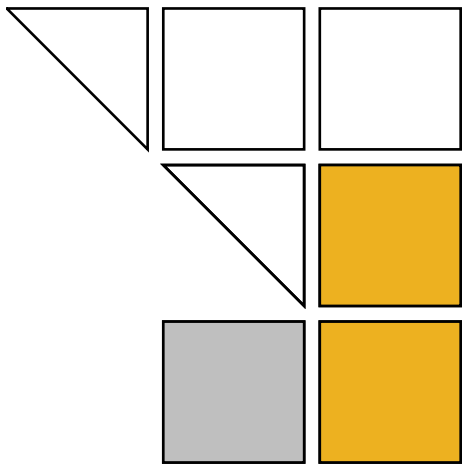
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

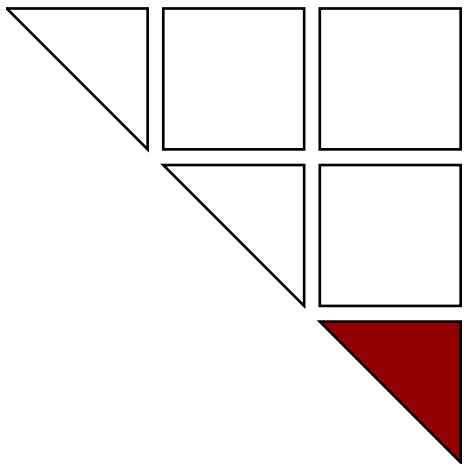
1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

The (sequential) tile QR factorization proceeds like this

1. `dgeqrt` : factorize the k-th diagonal tile
2. `dgemqrt`: update the k-th row wrt the `dgeqrt`
3. `dtsqrt` : use the diagonal tile to kill a subdiagonal one on row i, column k
4. `dtsmqrt`: update rows k and i wrt `dtsqrt`



Tile QR

This is the code of the tiled QR factorization, with a *flat tree panel reduction tree* because the sub-diagonal blocks in a column are annihilated sequentially (i.e., one after the other)

```
subroutine tiled_qr_flattree(A)
   do k=1, N
      !$omp task depend(inout:A[k,k])
      call dgemqrt(A[k,k])

      do j=k+1, N
         !$omp task depend(in:A[k,k]) depend(inout:A[k,j])
         call dgemqrt(A[k,j], A[k,k])
      end do

      do i=k+1, M
         !$omp task depend(in:A[k,k], A[i,k])
         !$omp&     depend(out:A[k,k], A[i,k])
         call dtsqrt(A[k,k], A[i,k])
         do j=k+1, N
            !$omp task depend(in:A[k,k], A[i,k])
            !$omp&     depend(inout:A[k,j],A[i,j])
            call dtsmqrt(A[k,j], A[i,j], A[k,k], A[i,k])
         end do
      end do
   end do
```

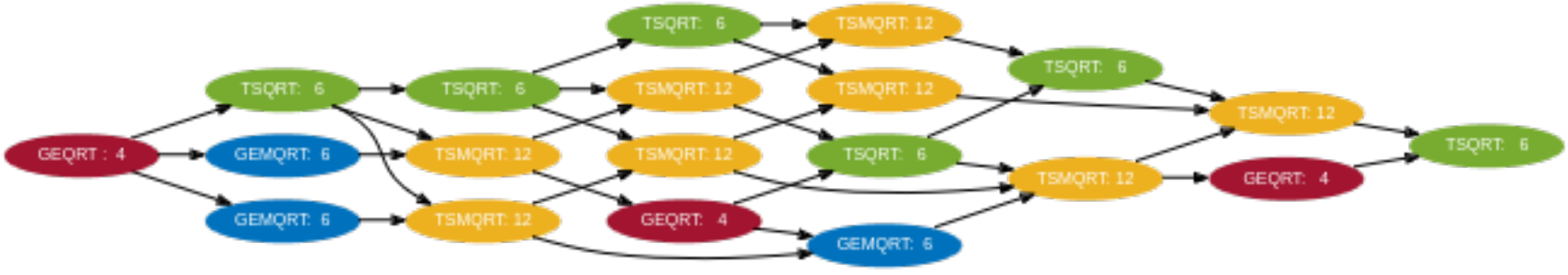
Parallelization can be achieved with a task-based approach

Tiled QR: critical path analysis

Tasks costs in $b^3/3$:

- `dgeqrt` : 4
- `dgemqrt`: 6
- `dtsqrt` : 6
- `dtsmqrt`: 12

For a matrix of size $m \times n = M * b \times N * b$ with $M = 4,\ N = 3$ the dependency graph is

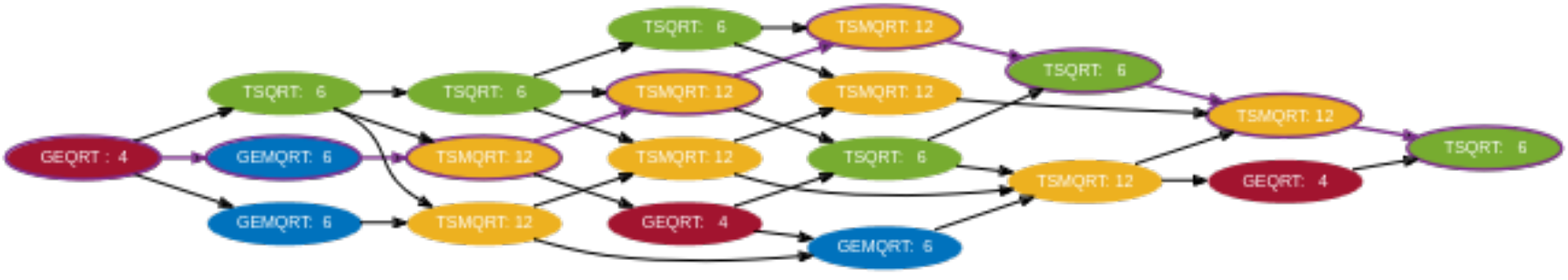


Tiled QR: critical path analysis

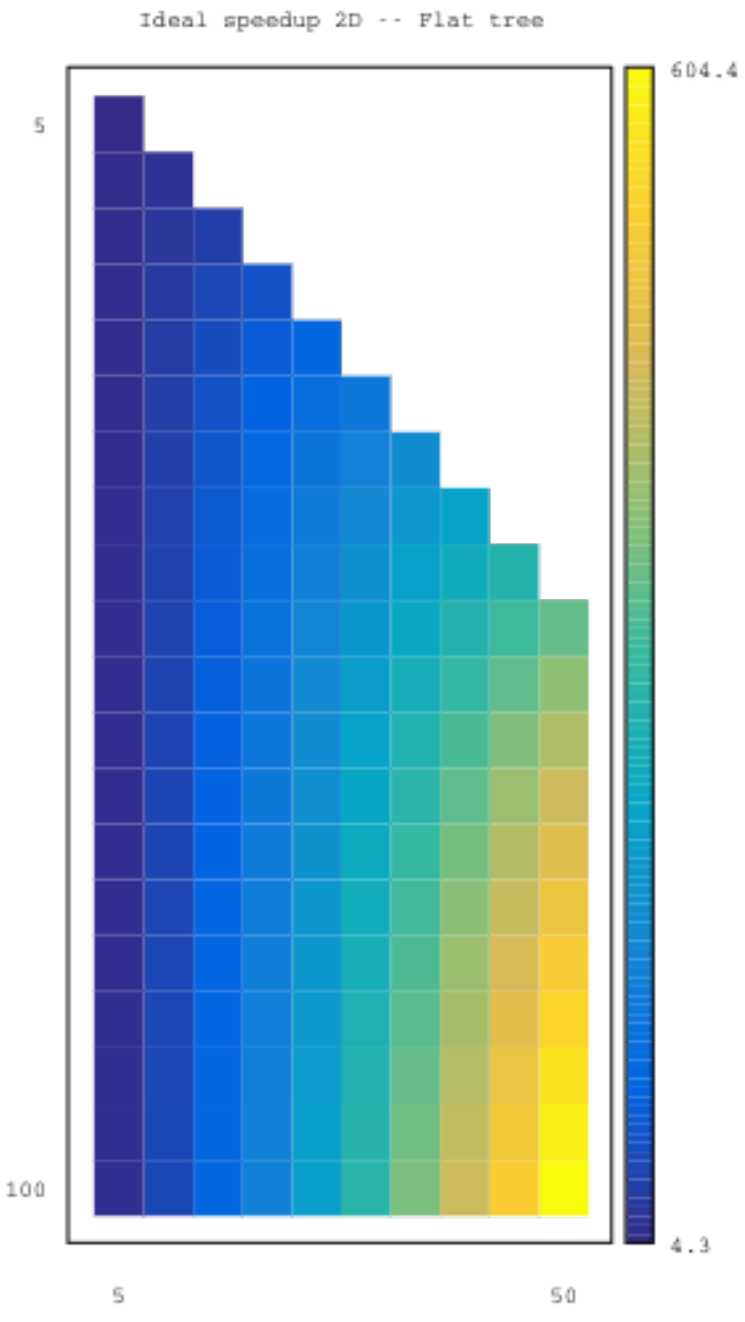
Tasks costs in $b^3/3$:

- `dgeqrt` : 4
- `dgemqrt`: 6
- `dtsqrt` : 6
- `dtsmqrt`: 12

For a matrix of size $m \times n = M * b \times N * b$ with $M = 4,\ N = 3$ the dependency graph is



Tiled QR: critical path analysis



$$\frac{\sum_{k=1}^{N} 4 + 6(N-k) + 6(M-k) + 12(M-k-1)(N-k)}{4 + 6 + 12(M-2) + (12+6)(N-1)}$$

Much better theoretical speedups with respect to the 1D panelwise parallelization.

Tiled QR

You may have noticed hat all the tasks related to tiles along a column form a chain in the DAG. This means that their execution is serialized. This is only slightly better than what we had with a 1D partitioning mostly because the reduction/update on a column is not parallelized but only decomposed in multiple tasks that can be interleaved with others.

Therefore, we can still expect that this version of the tile algorithm suffers on tall and skinny matrices.

Thanks to the great flexibility of Householder transformations, it is possible to devise different algorithms that are more suited to this kind of matrices [11]

Tile QR factorization

$$\tilde{R},\ Q = \texttt{dttqrt}(R_A,\ R_B),\ \text{where}\ Q\tilde{R} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{bmatrix} \tilde{R}_1 \\ \end{bmatrix} = \begin{bmatrix} R_A \\ R_B \end{bmatrix}$$

Computes the QR factorization of a matrix formed by a triangle on top of a square. The Q matrix is stored implicitly in the form of Householder reflectors.

$$\tilde{B},\ \tilde{C} = \texttt{dttmqrt}(B,\ C,\ Q),$$

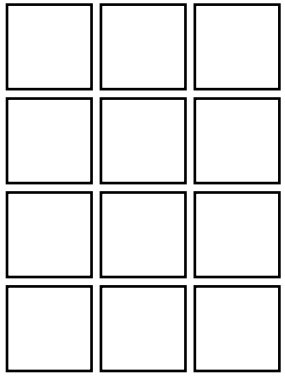
$$\text{where} \quad \begin{bmatrix} \tilde{B} \\ \tilde{C} \end{bmatrix} = Q^T \begin{bmatrix} B \\ C \end{bmatrix} = \begin{bmatrix} Q_{11}^T & Q_{21}^T \\ Q_{12}^T & Q_{22}^T \end{bmatrix} \begin{bmatrix} B \\ C \end{bmatrix}$$

Applies the transformation computed by `dttqrt` to a matrix formed by two squares, one on top of the other.

Both routines work in-place (i.e., the output overwrites the input) and are *structured* (i.e., skip computations with the zeros).

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



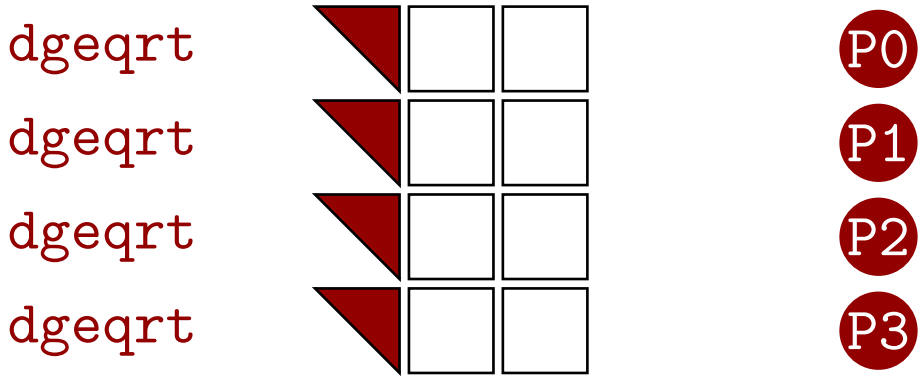
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



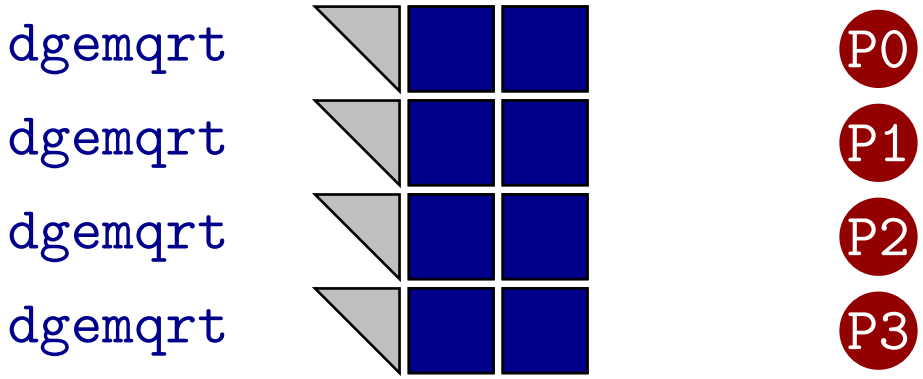
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



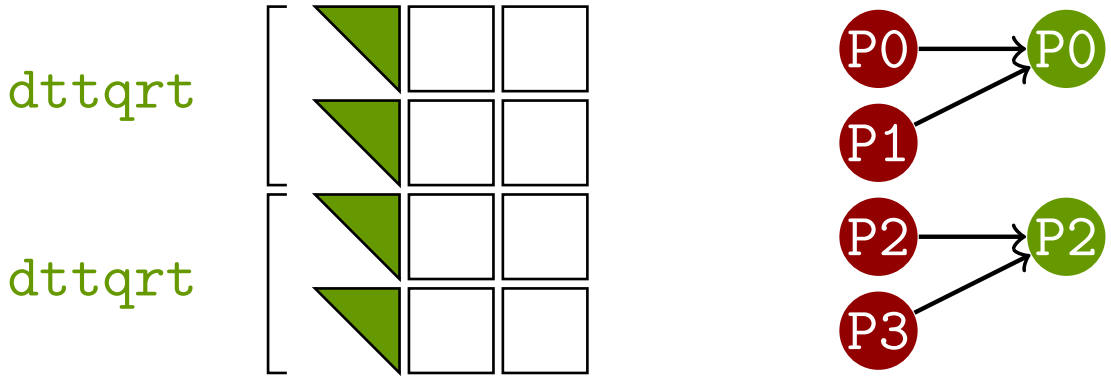
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



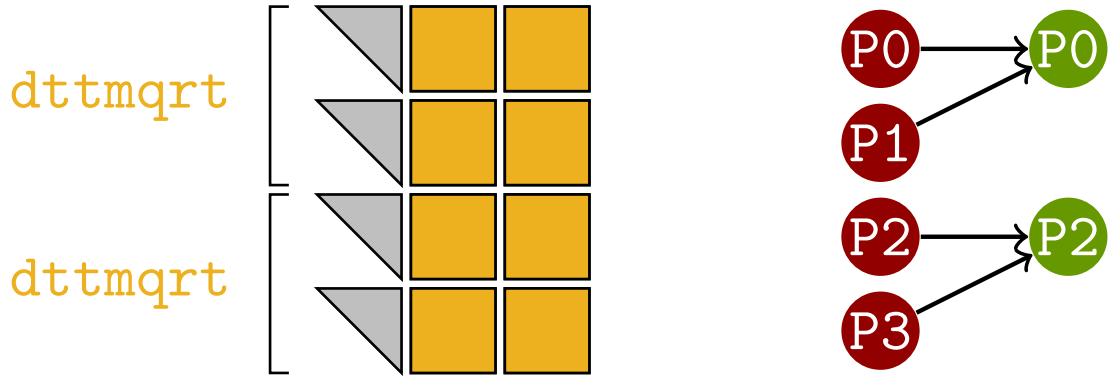
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



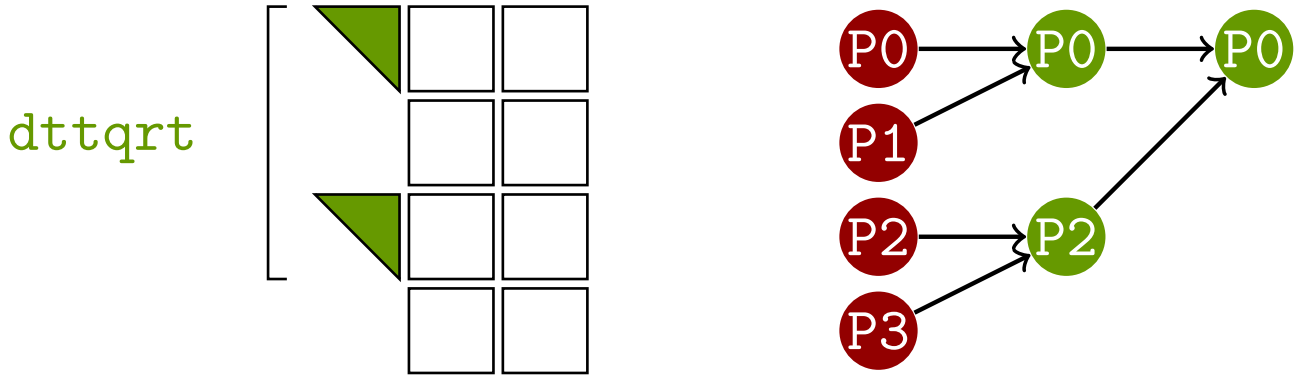
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



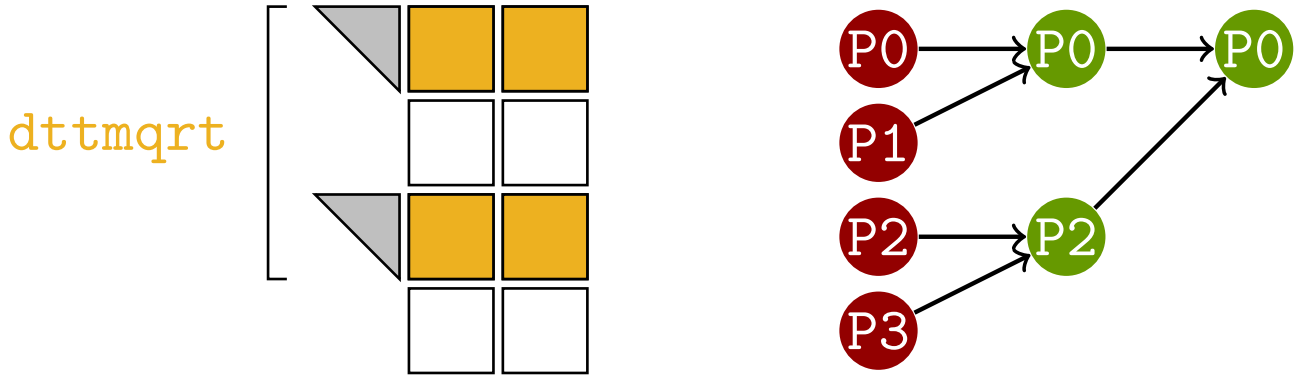
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



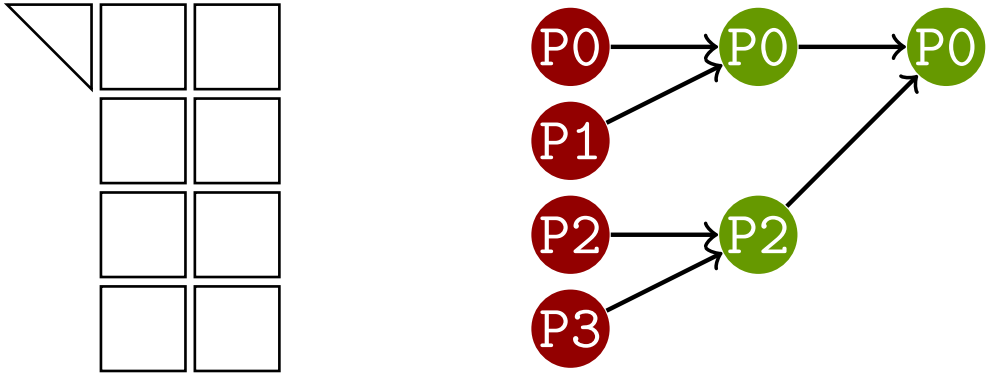
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

Tiled QR

It is possible to devise algorithms that are based on the idea that multiple tiles along a column can be annihilated independently



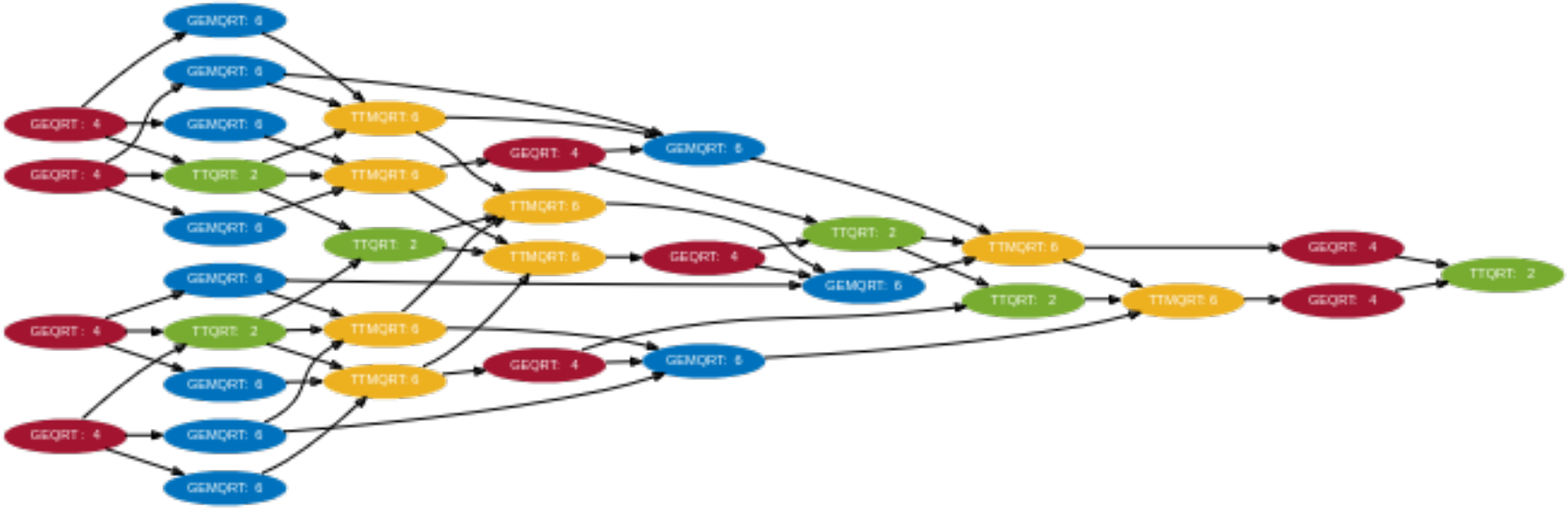
This allows for :

1. annihilating multiple tiles of the panel at the same time by calling `dttqrt` in parallel.
2. updating multiple tiles of the trailing submatrix at the same time by calling `dttmqrt` in parallel.

This delivers much more concurrency when the matrix is overdetermined (i.e., $M \gg N$)

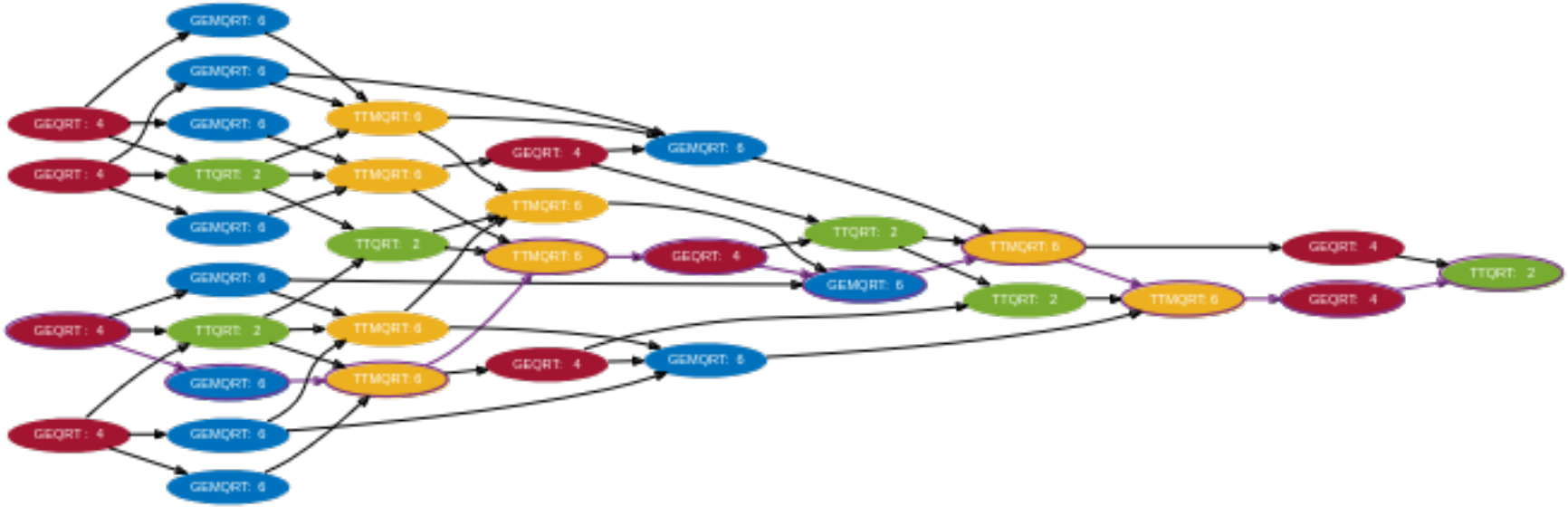
Critical path analysis

`dttqrt` and `dttmqrt` take, respectively 2 and 6 time units
For a matrix of size $m \times n = M * b \times N * b$ with $M = 4,\ N = 3$ the dependency graph is

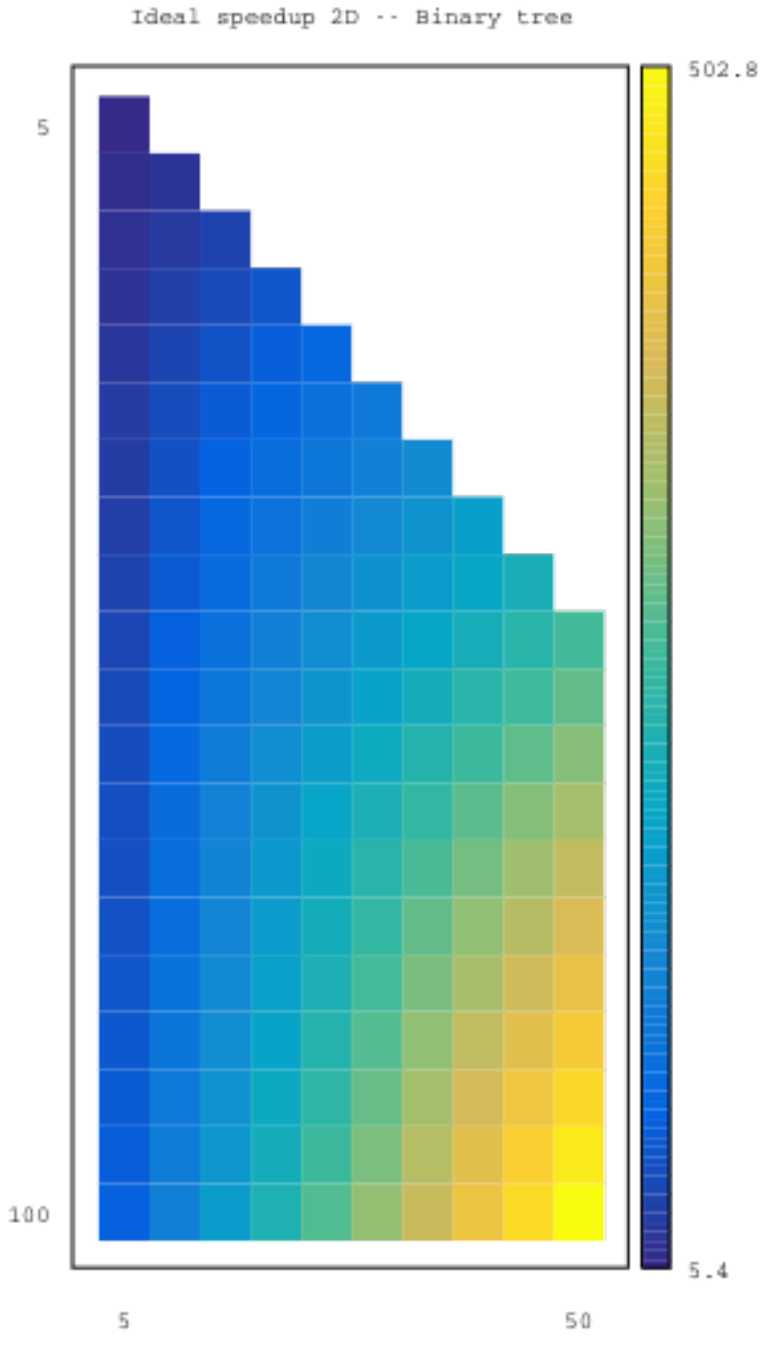


Critical path analysis

`dttqrt` and `dttmqrt` take, respectively 2 and 6 time units
For a matrix of size $m \times n = M * b \times N * b$ with $M = 4,\ N = 3$ the dependency graph is



Critical path analysis



The binary panel reduction provides better parallelism for extremely overdetermined matrices but suffer from poor pipelining of successive panel stages on less overdetermined ones. Moreover, the `dttqrt` and `dttmqrt` are less efficient than the `dtsqrt` and `dtsmqrt` ones because they involve two triangular blocks.

Outline

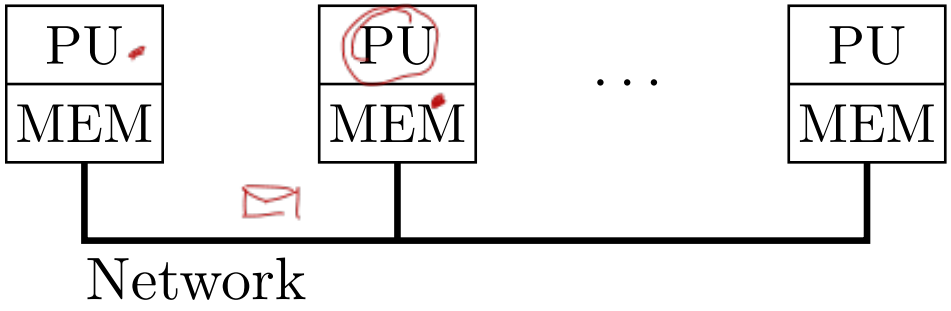
$T(n,1)$

$T(n,p)$

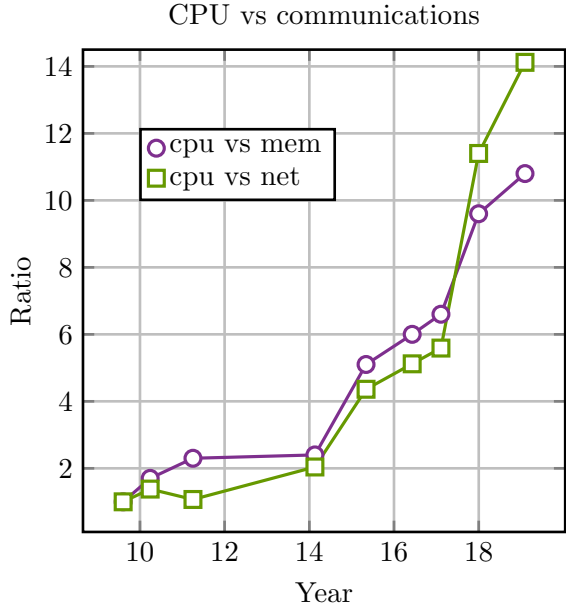
$p\ T(n,p) = T(n,1) + O(n,p)$

Distributed memory parallelism

A distributed memory parallel computer is one where processing units (PU) are attached to (logically) different memories. This means that one PU cannot directly access data in another memory but the data must be transferred beforehand.



Communications are extremely slow compared to the speed of computations $\rightarrow$ the $O(n, p)$ term (overhead) may end up dominating the execution time. Communications must be avoided or reduced as much as possible.



Distributed memory parallelism

Because of communications, numerous issues arise in distributed memory parallel programming:

- Data distribution: how is data distributed among the available nodes in such a way that communications are reduced/minimized?
- Operations distribution: how are operations assigned to processing units in order to reduce communications?
- Communication patterns: is it possible to use efficient communication approaches such as collectives or non-blocking communications to overlap communications and computations?

Accurate communication models help designing efficient algorithms

The Hockney model for communications

The Hockney model assumes that the time for exchanging a message of size m between two nodes is

γ = t pour calcul scalaire

$$t(m) = \alpha + \beta m$$

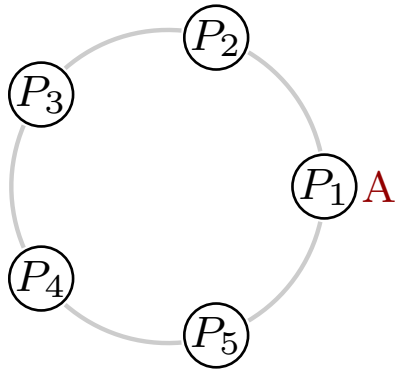
- α: is the so called *latency*. This term is independent of the size of the message
- β: how much time it takes to trasfer a unit of data (e.g., bit or byte). This is the reciprocal of the bandwidth

Many different network topologies exist but we assume a fully-connected graph, i.e., this time is the same for all nodes pairs

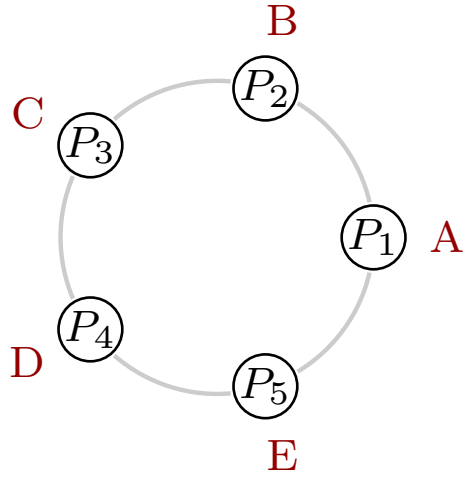
A node cannot send (receive) two messages simultaneously but can send one message and receive another at the same time.

Hockney model: example

Message sent in a ring: messages A of size m are sent sequentially along the ring

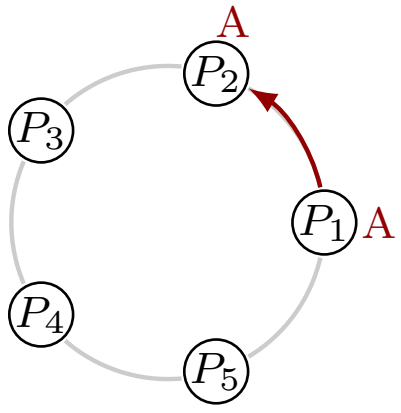


Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)

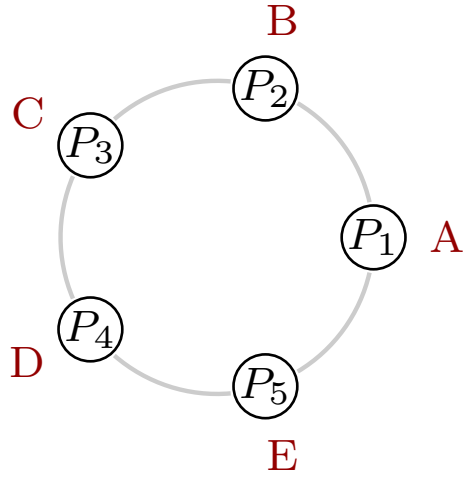


Hockney model: example

Message sent in a ring: messages A of size m are sent sequentially along the ring

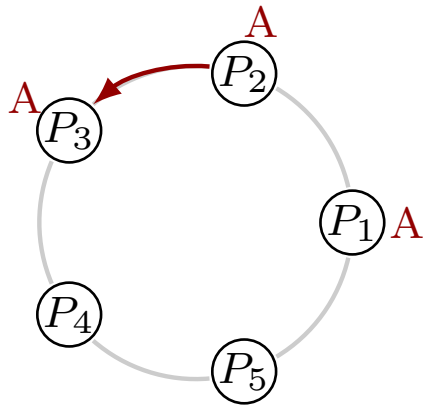


Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)

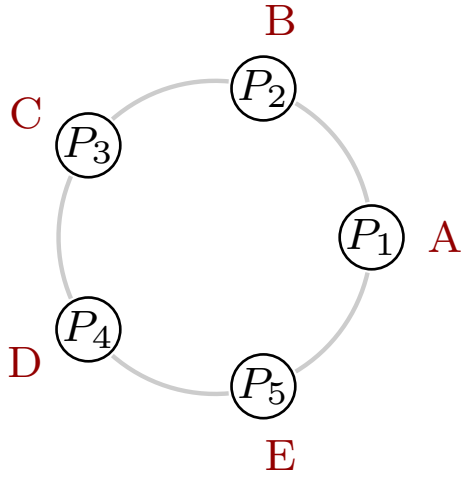


Hockney model: example

Message sent in a ring: messages A of size m are sent sequentially along the ring

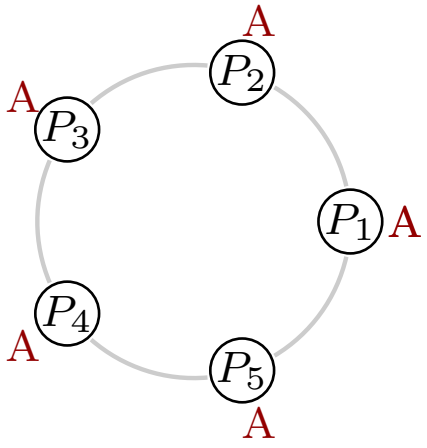


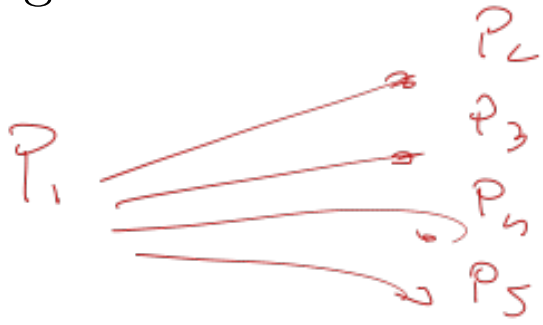
Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)



Hockney model: example

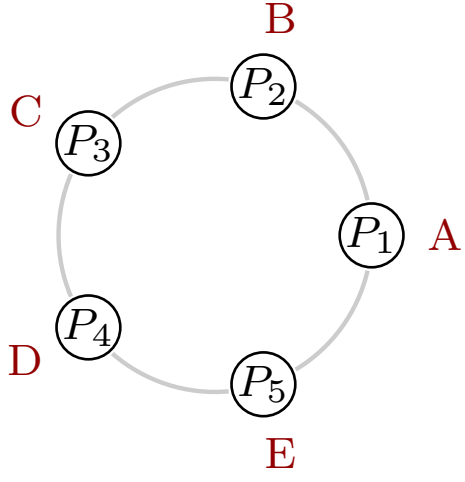
Message sent in a ring: messages A of size m are sent sequentially along the ring





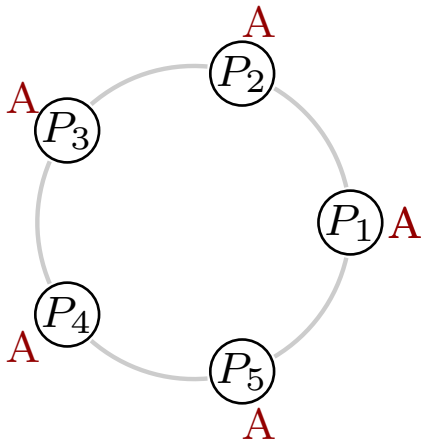
$$T_{bcast,rng}(p,m) = p(\alpha + \beta m)$$

Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)



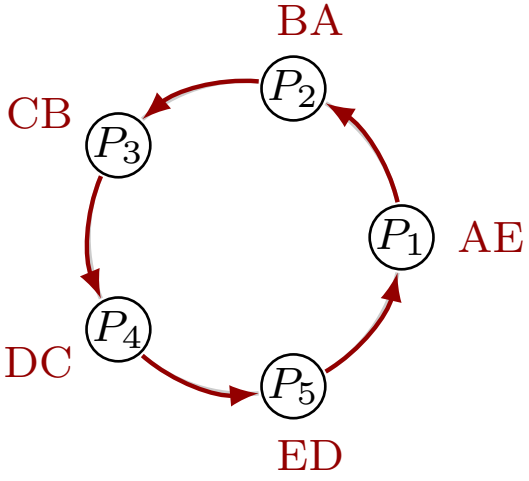
Hockney model: example

Message sent in a ring: messages A of size m are sent sequentially along the ring



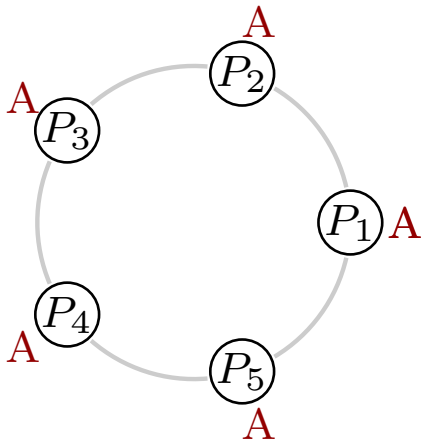
$$T_{bcast,rng}(p,m) = p(\alpha + \beta m)$$

Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)



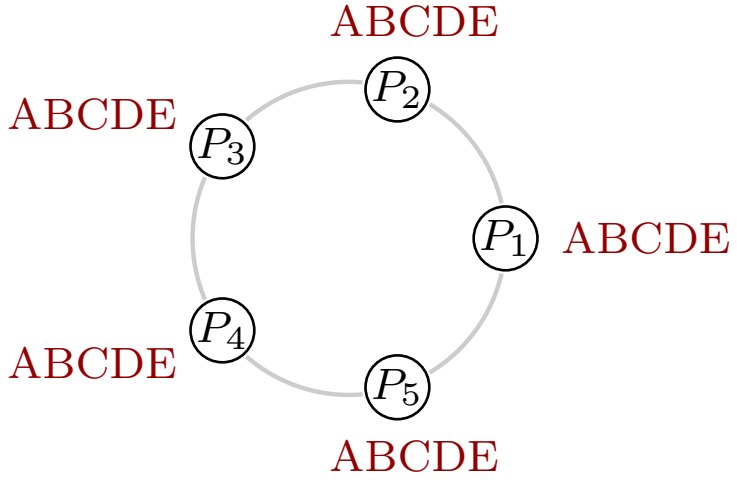
Hockney model: example

Message sent in a ring: messages A of size m are sent sequentially along the ring



$$T_{bcast,rng}(p,m) = p(\alpha + \beta m)$$

Note that for the same cost we can implement an *allgather* communication (note that m is the size of the final data)



$$T_{allg,rng}(p,m) = p(\alpha + \beta m)$$

Collective communications

Collective communications employ relatively complex patterns in order to reduce the overall time with respect to communication loops.

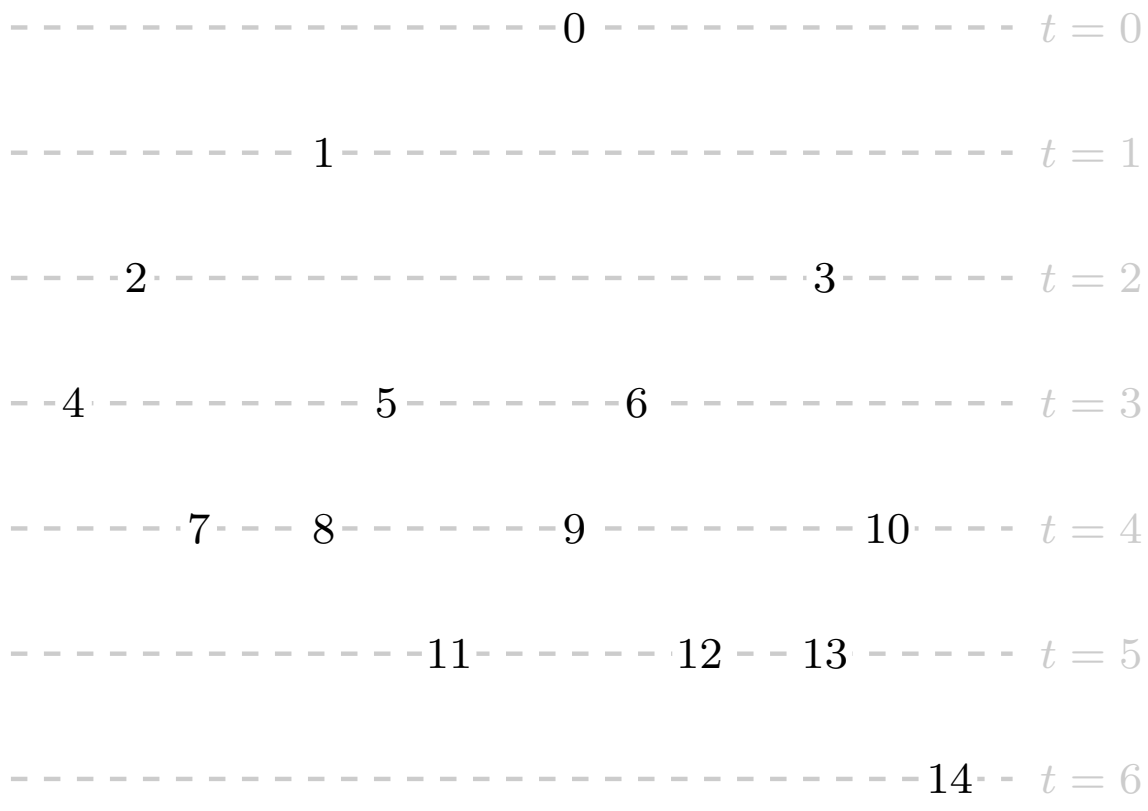
There are different types of collectives:

- **one-to-all**: broadcast, scatter
- **all-to-one**: gather, reduce
- **all-to-all**: allreduce, allgather, reduce-scatter, all-to-all

All of these, and more, are provided by the MPI standard

One-to-all: broadcast

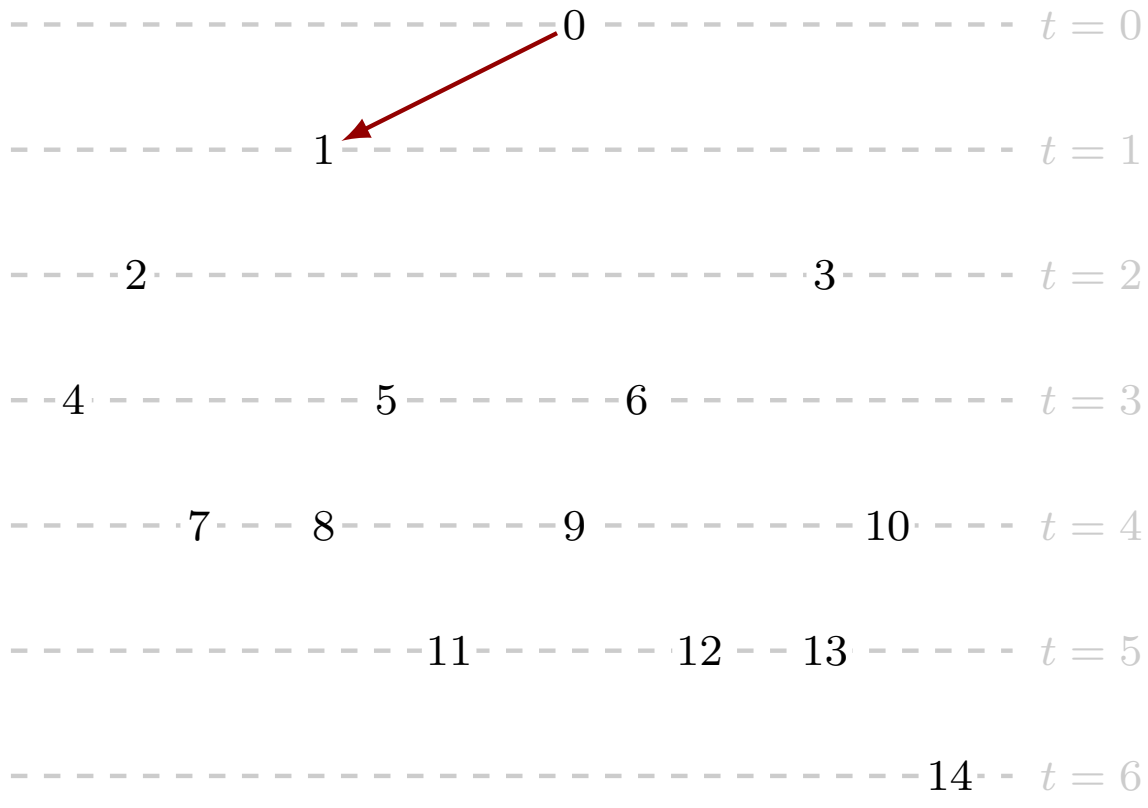
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

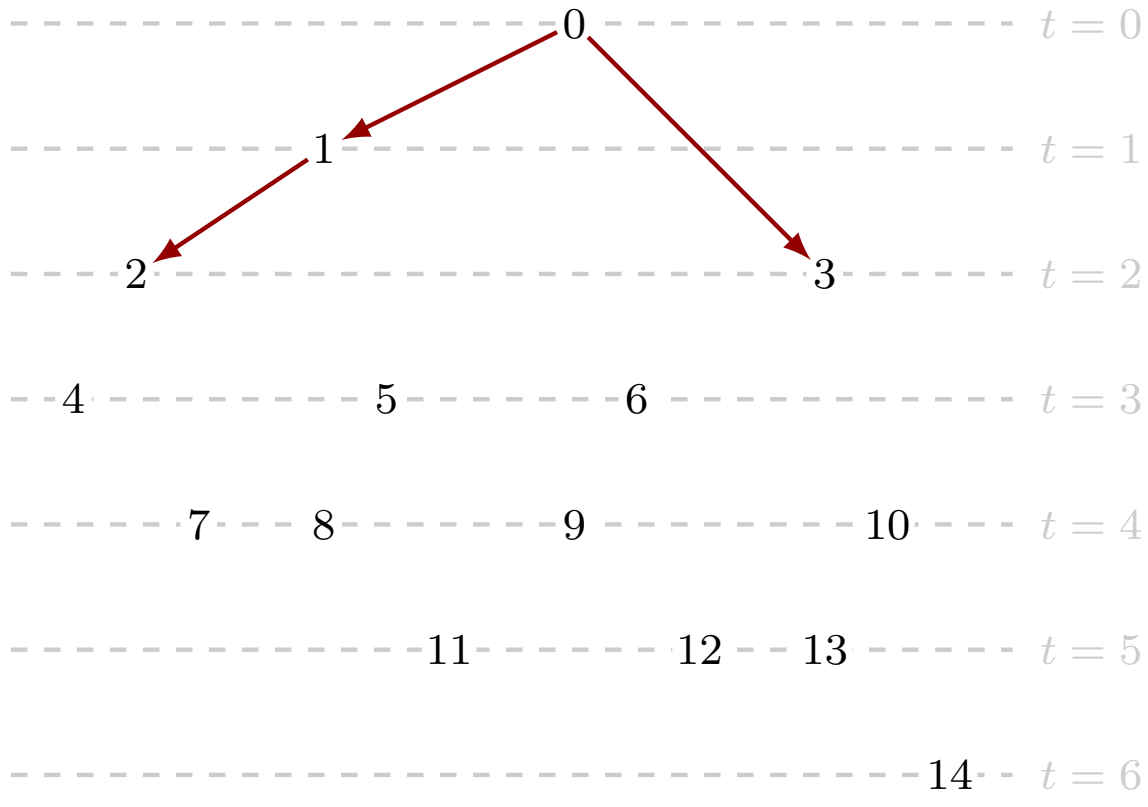
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

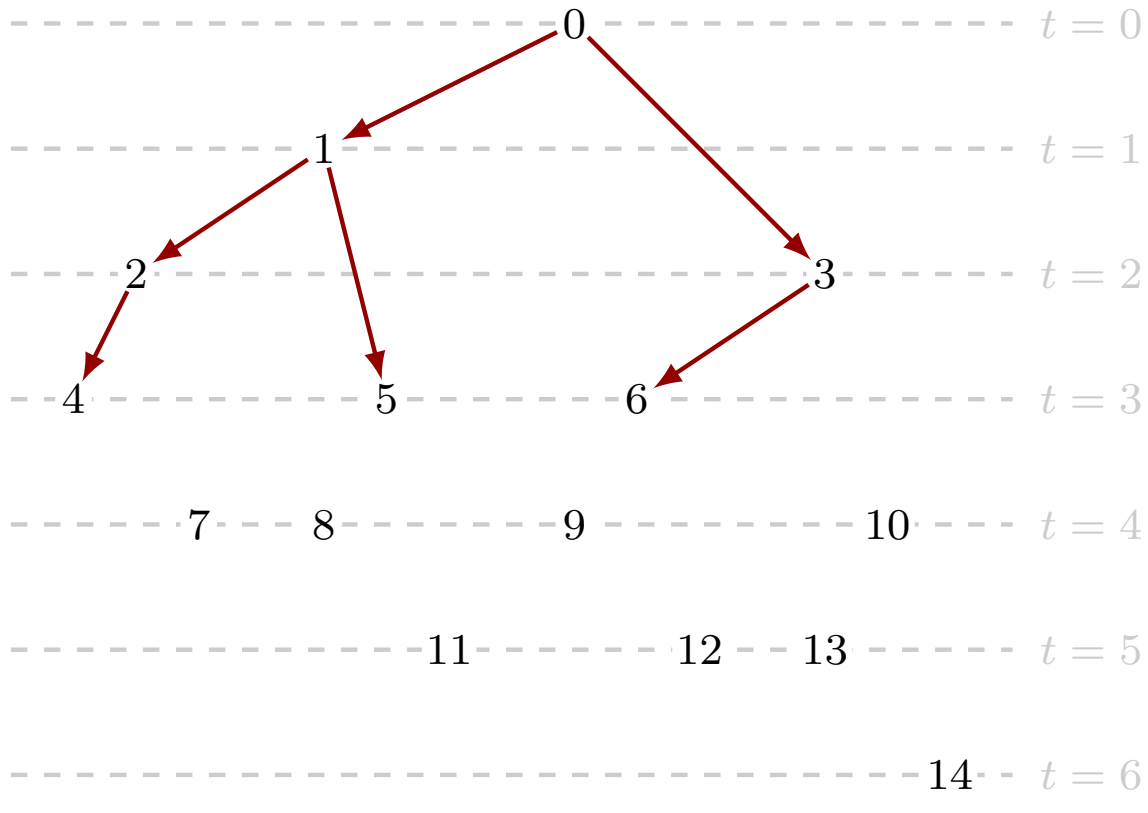
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

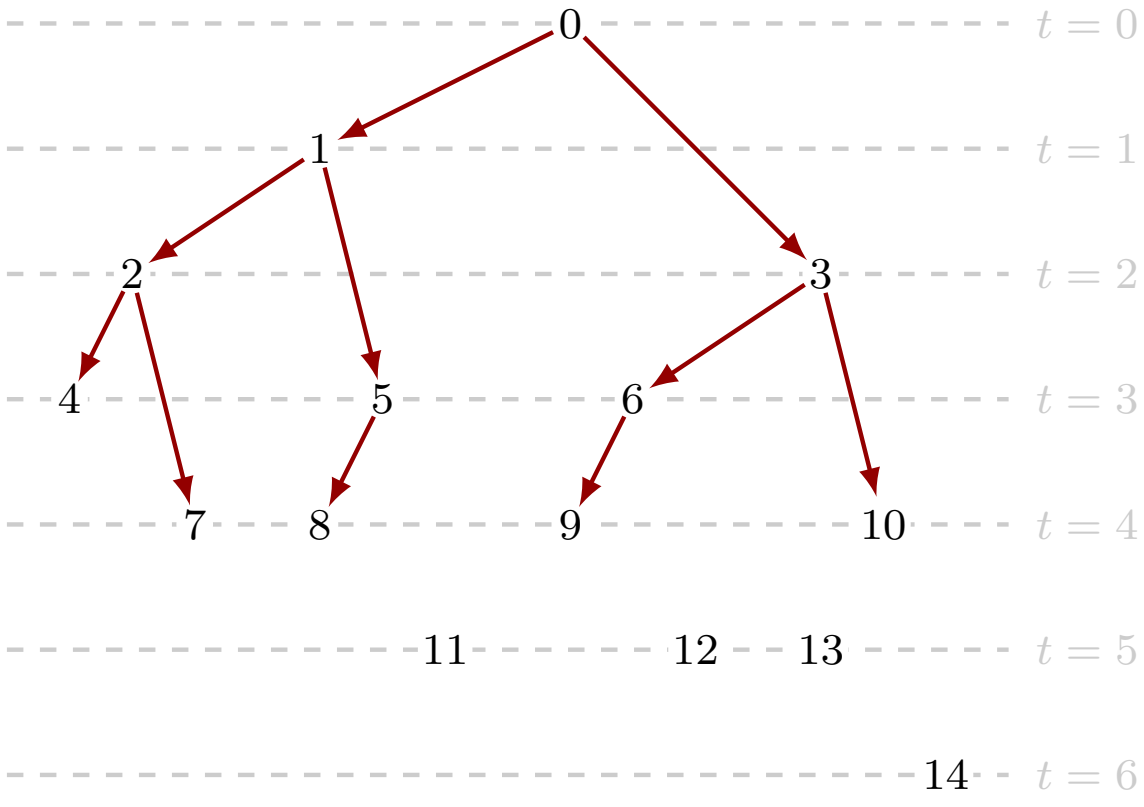
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

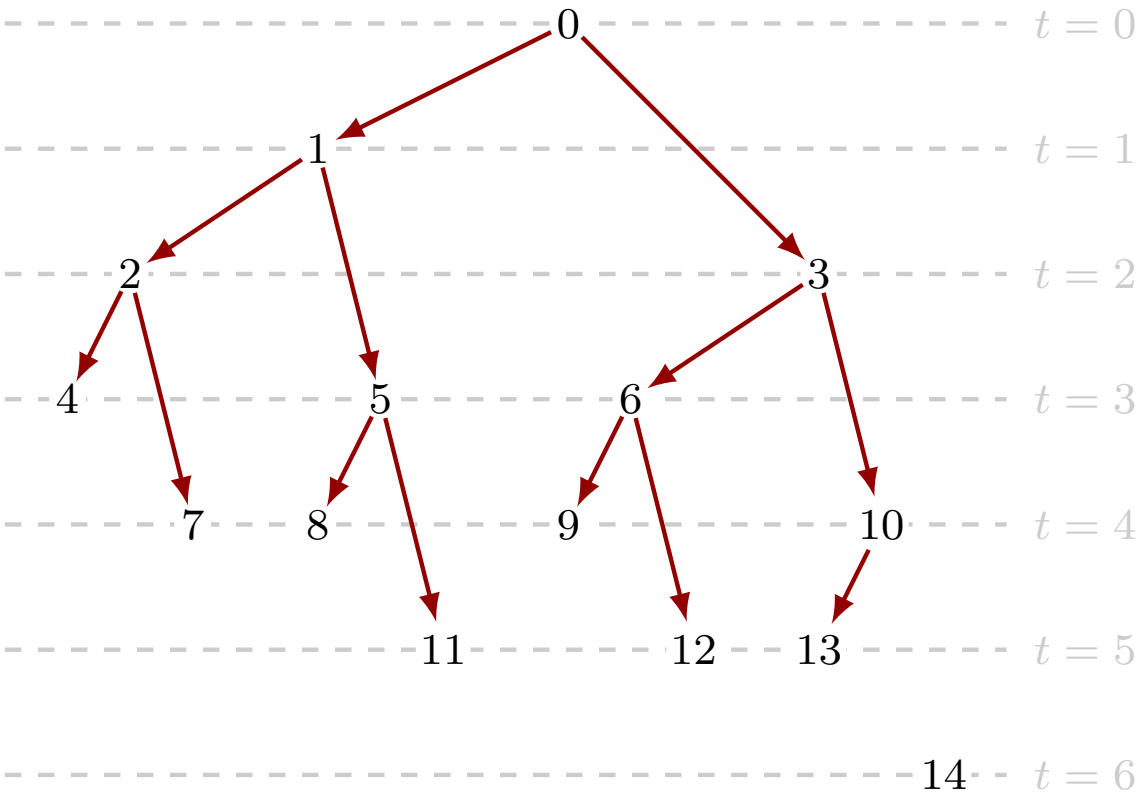
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

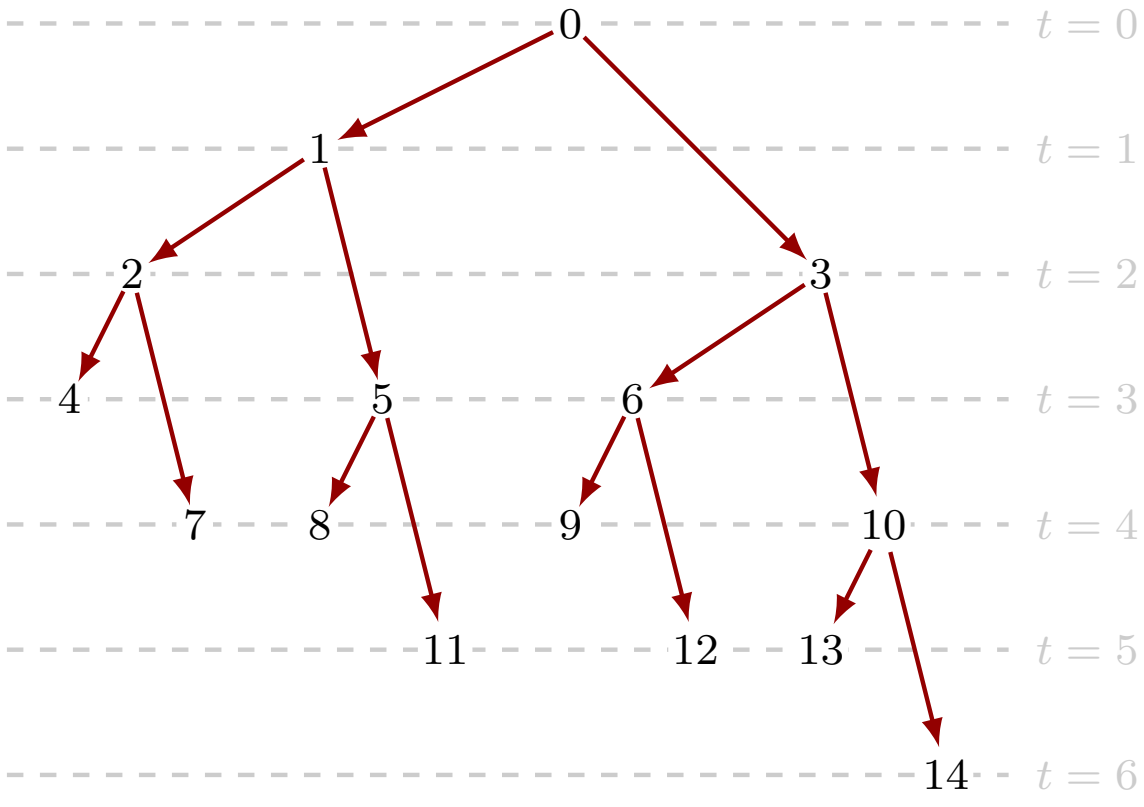
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

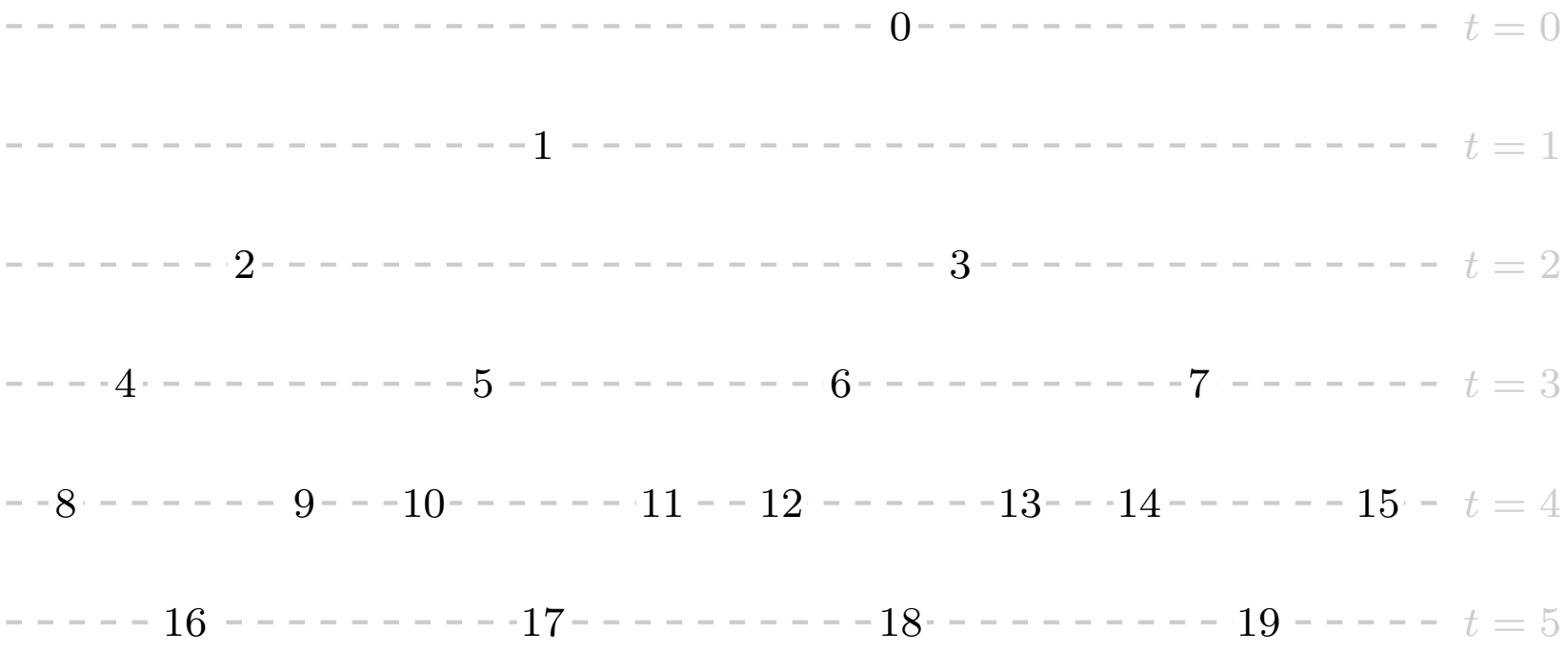
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binary tree: $T_{bcast,bnr}(m,p) = 2\log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

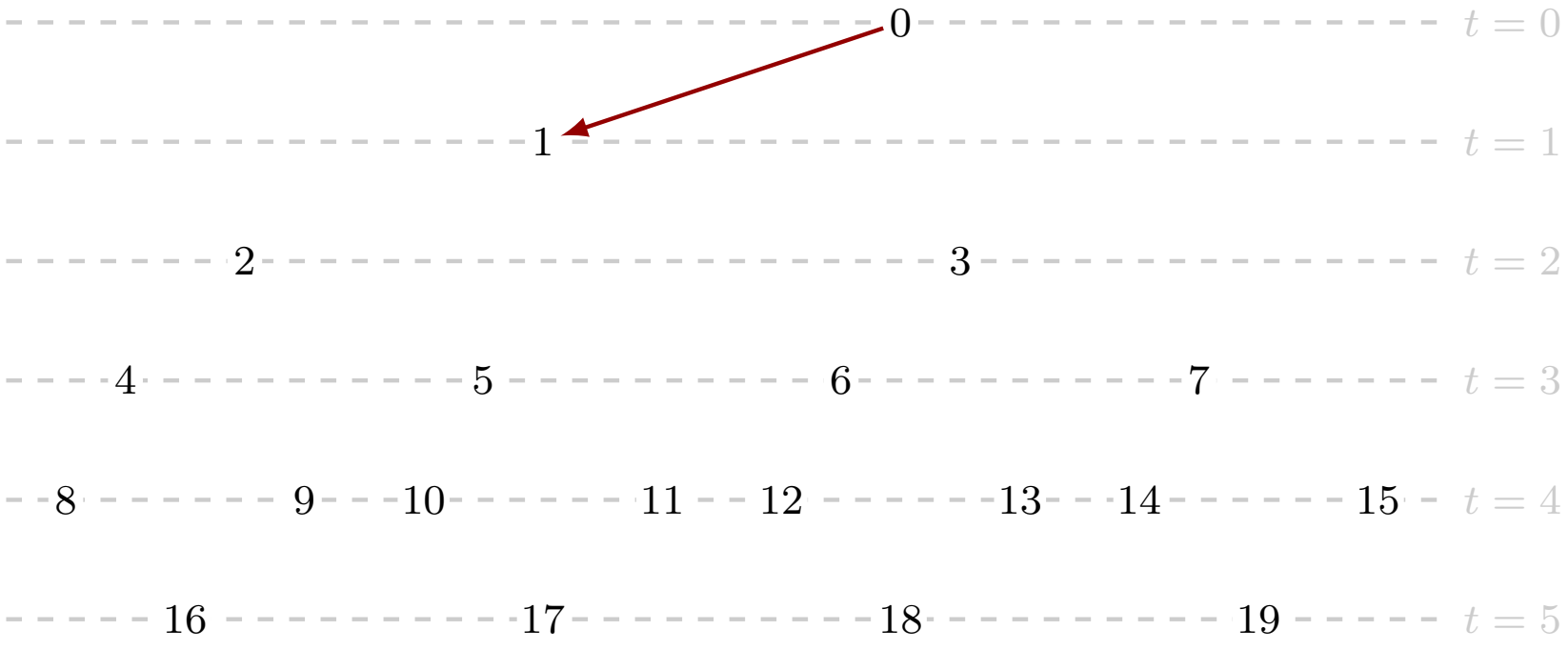
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

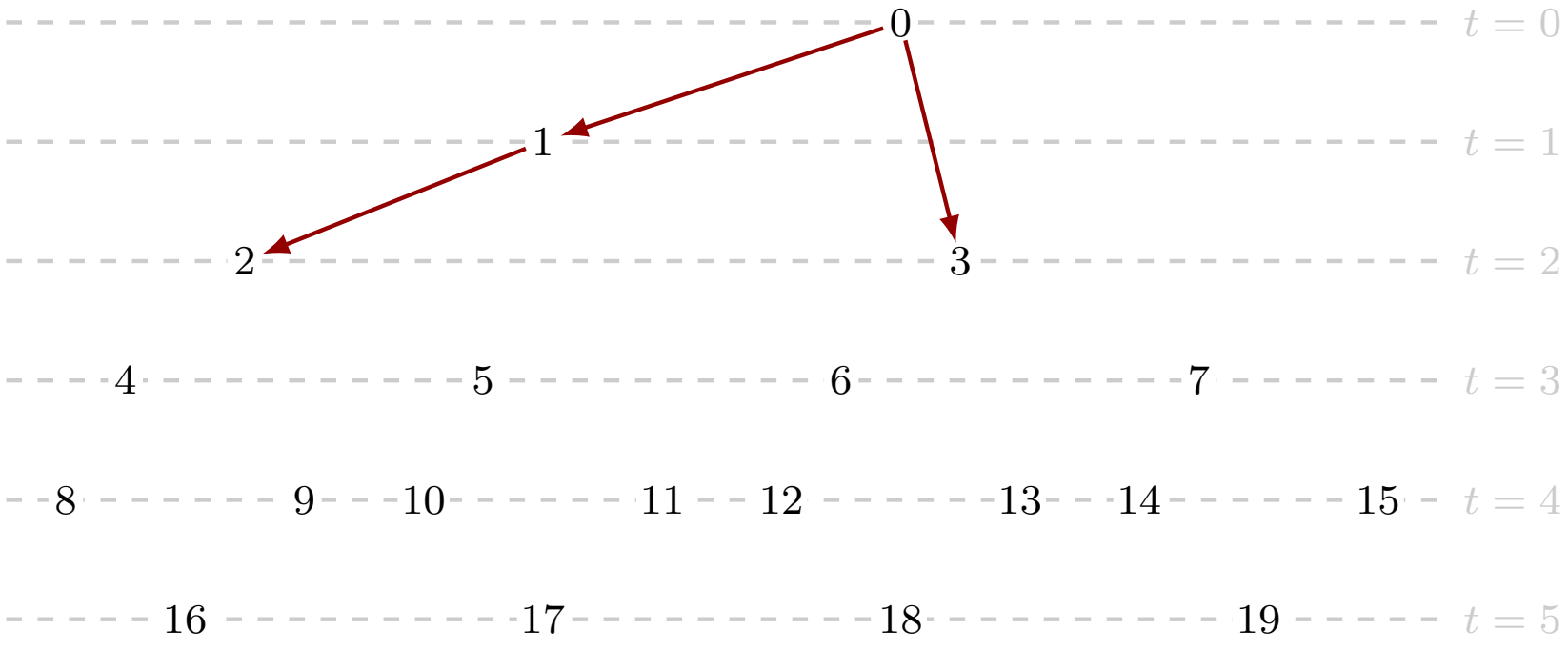
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

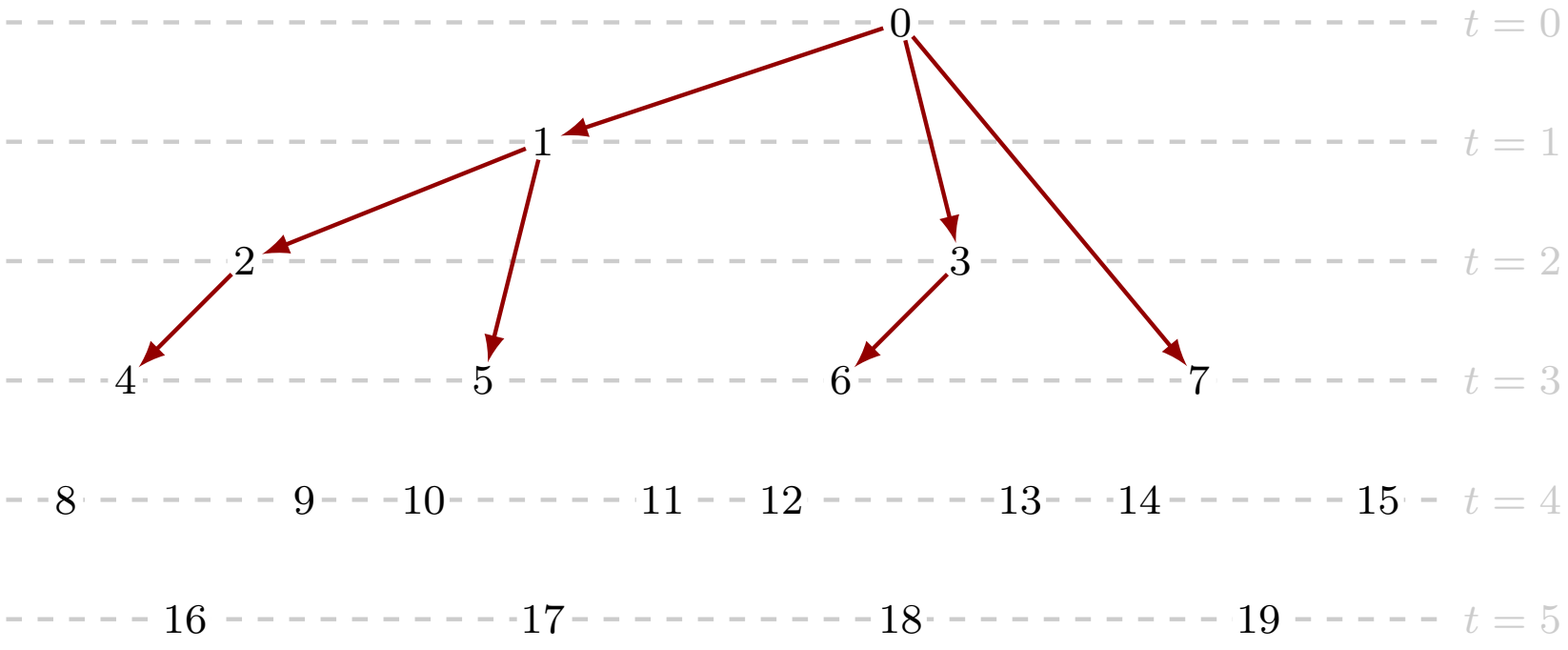
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

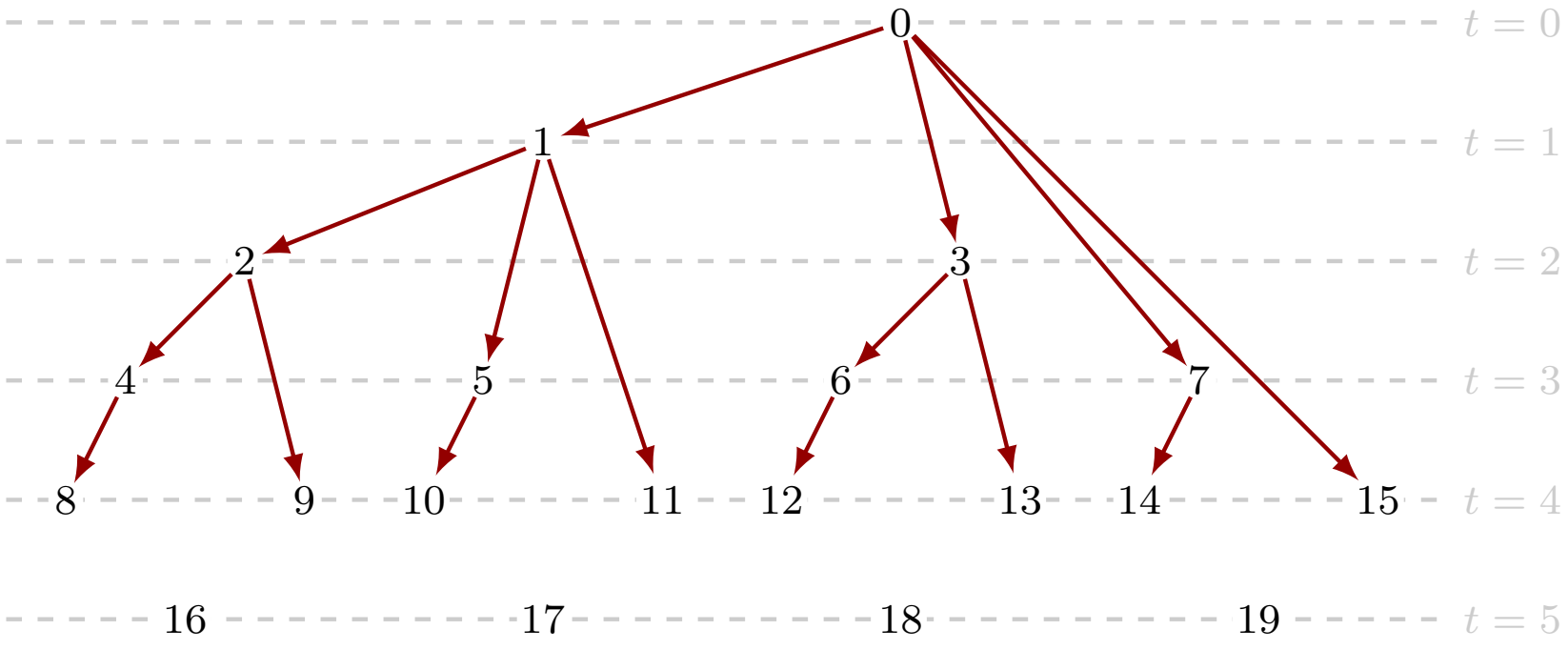
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

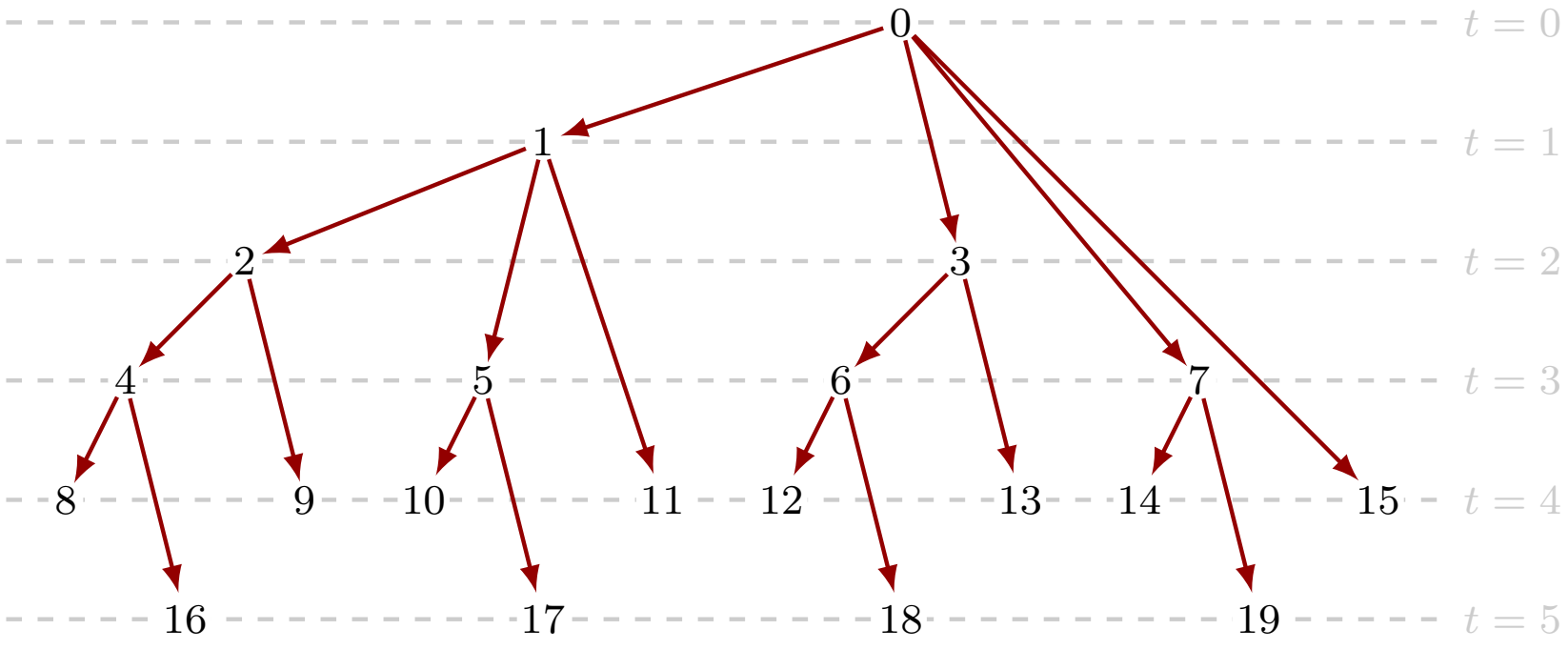
Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

Broadcast: send the same message to p nodes. This can obviously be done using p point-to-point messages but better ways exist.



Total time, binomial tree: $T_{bcast,bnm}(m,p) = \log_2(p)(\alpha + \beta m)$

One-to-all: broadcast

Should we always use the same tree or,re generally, the same approach for the broadcast and other collective communications?

Large messages:

1. The sender splits the message into p chunks and send each chunk to a different receiver

$$T_1 = p(\alpha + \beta m/p)$$

2. All receivers exchange their chunks using a ring allgather

$$T_2 = T_{allg,rng}(p, m/p) = p(\alpha + \beta m/p)$$

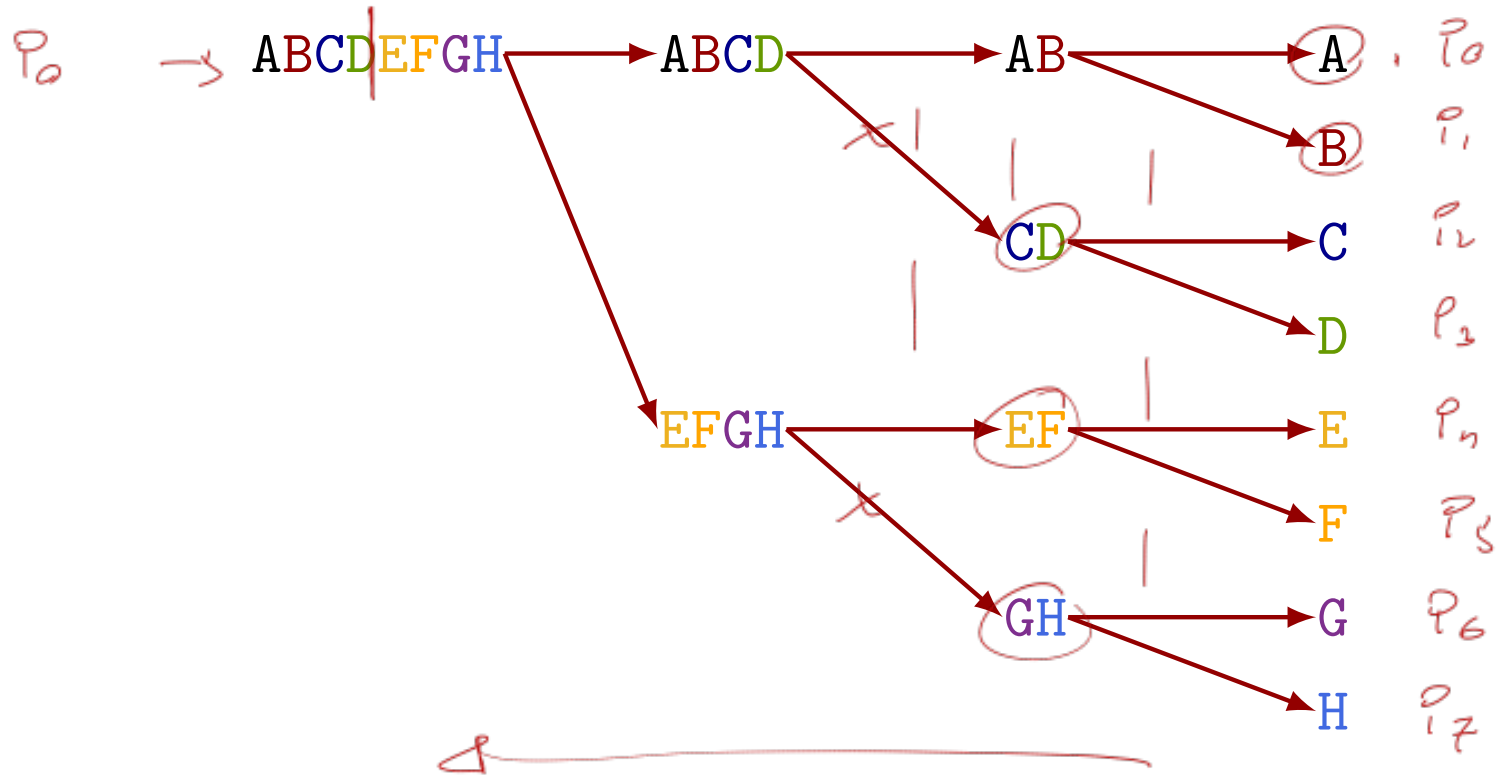
The total time using this approach is

$$T_{bcast,rng}(m, p) = 2(\alpha p + \beta m)$$

Therefore

$$\lim_{m\to\infty} \frac{T_{bcast,bnm}(m, p)}{T_{bcast,rng}(m, p)} = \log_2(p)/2$$

One-to-all, all-to-one: scatter and gather

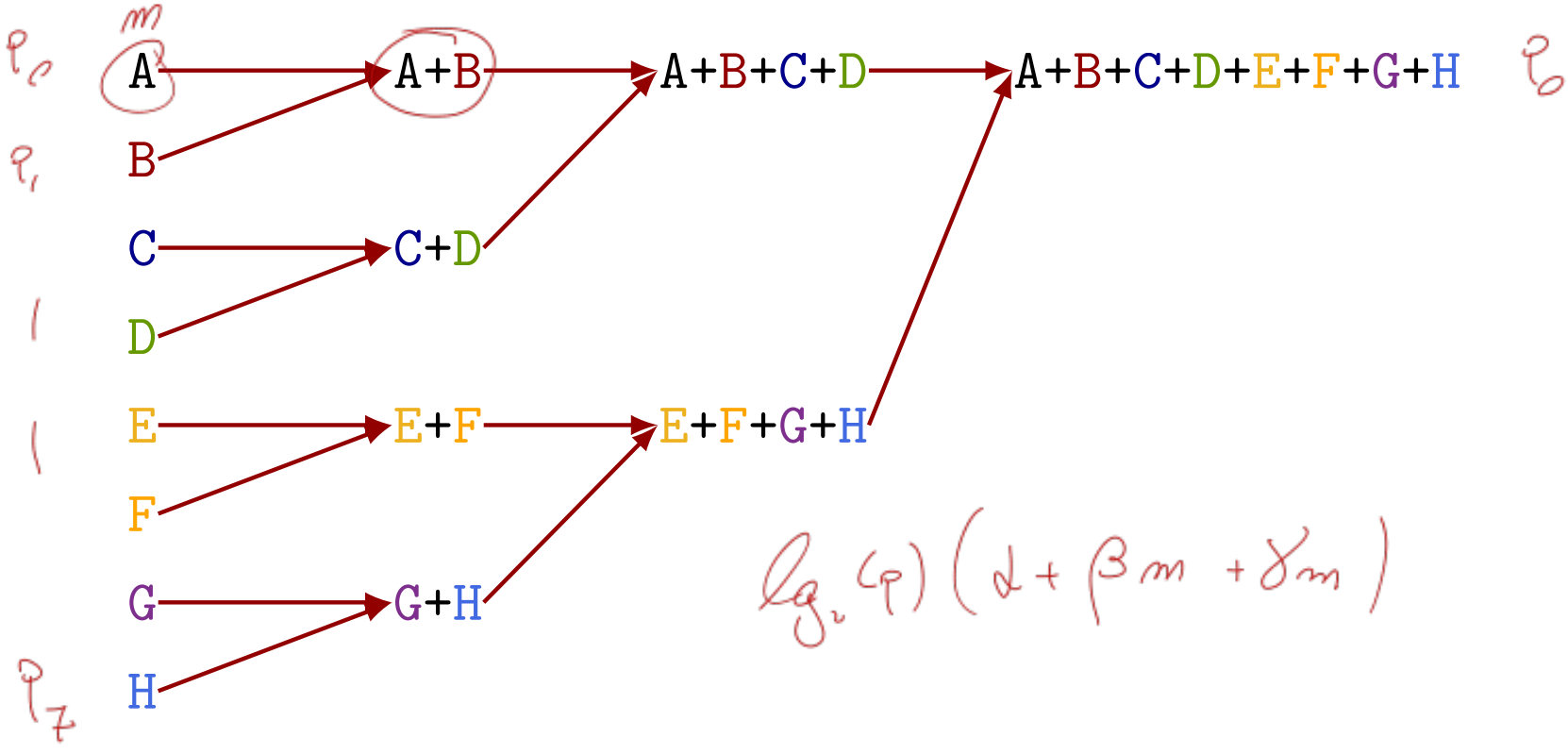


The scatter is the same as the broadcast with a reversed tree but the size of the message is divided by two at each step:

$$T_{scat} = \sum_{i=1}^{\log_2(p)} \left(\alpha + \beta \frac{m}{2^i}\right) = \log_2(p)\alpha + \beta m$$

The gather is the opposite of the scatter and has the same cost $T_{gath} = T_{scat}$

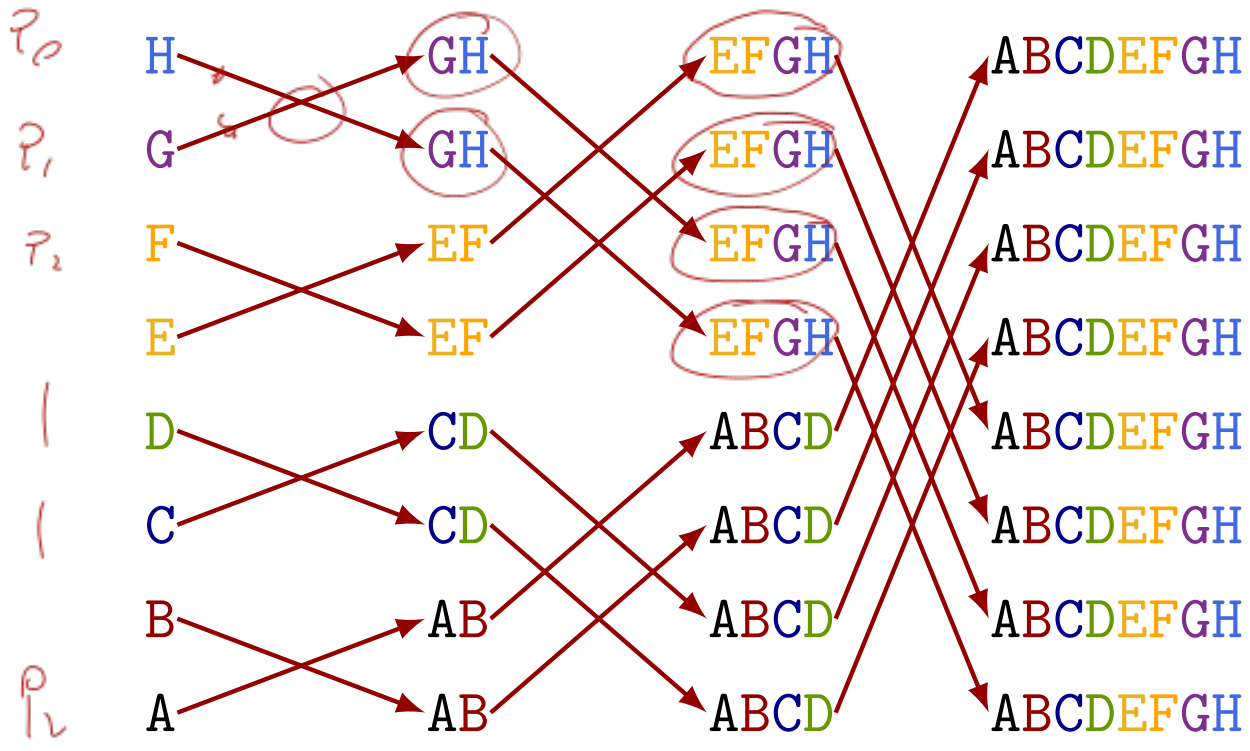
All-to-one: reduce



The reduce has the same cost as a broadcast (with an inverted tree) because the size of the message stays the same, plus the cost of the operations, which take γ time each

$$T_{red}(m,p) = \log_2(p)(\alpha + (\beta + \gamma)m)$$

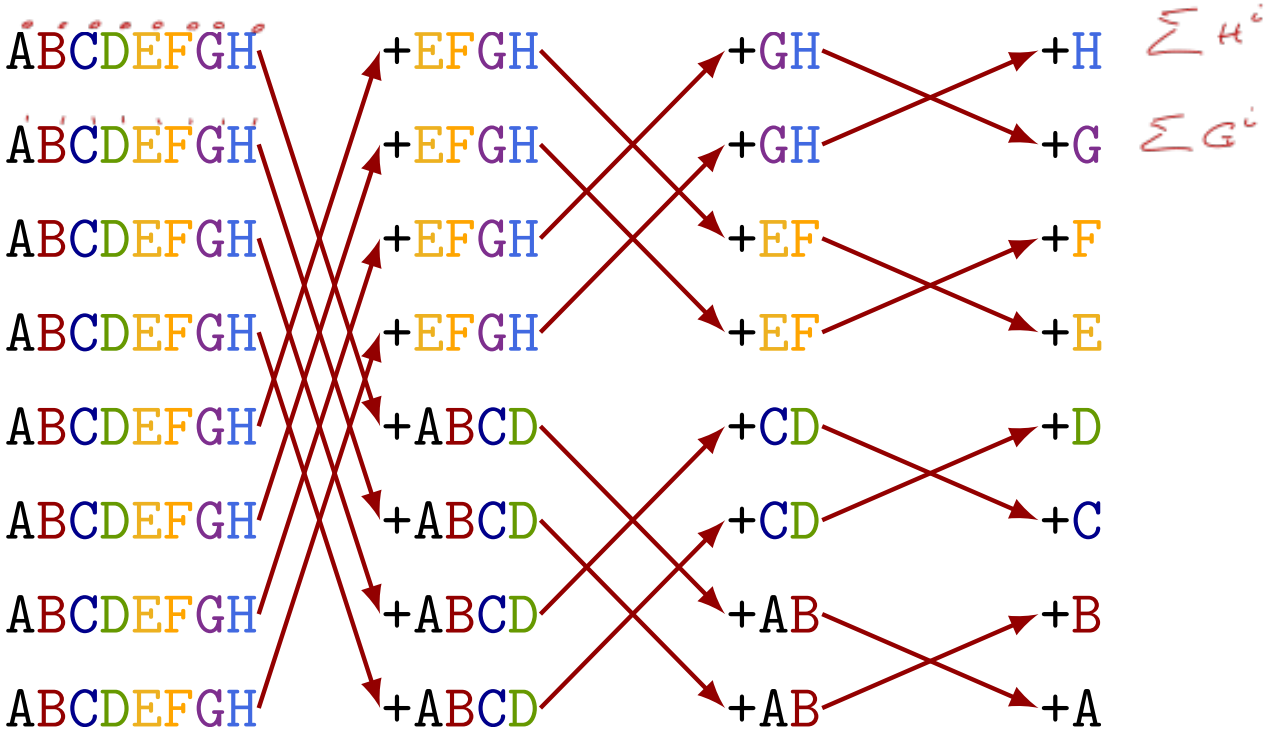
All-to-all: butterfly allgather



Considered that the size of the message doubles at every step:

$$T_{allg,bfl} = \sum_{i=1}^{\log_2(p)} \left(\alpha + \beta \frac{m}{2^i}\right) = \log_2(p)\alpha + \beta m$$

All-to-all: butterfly reduce-scatter

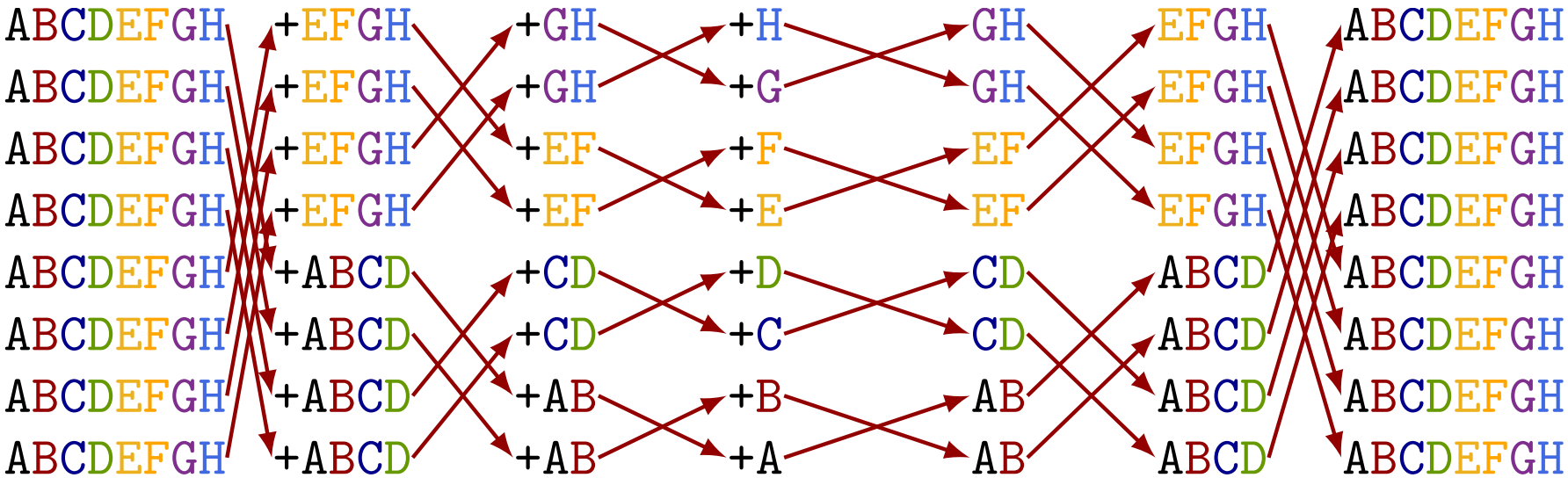


Considered that the size of the message doubles at every step:

$$T_{rsca,bfl} = \sum_{i=1}^{\log_2(p)} \left(\alpha + (\beta + \gamma)\frac{m}{2^i}\right) = \log_2(p)\alpha + (\beta + \gamma)m$$

where γ is the cost of local reduce operations

All-to-all: butterfly allreduce



The total time corresponds to the time of a reduce-scatter plus an allgather

$$T_{allr,bfl} = T_{rsca,bfl} + T_{allg,bfl}$$

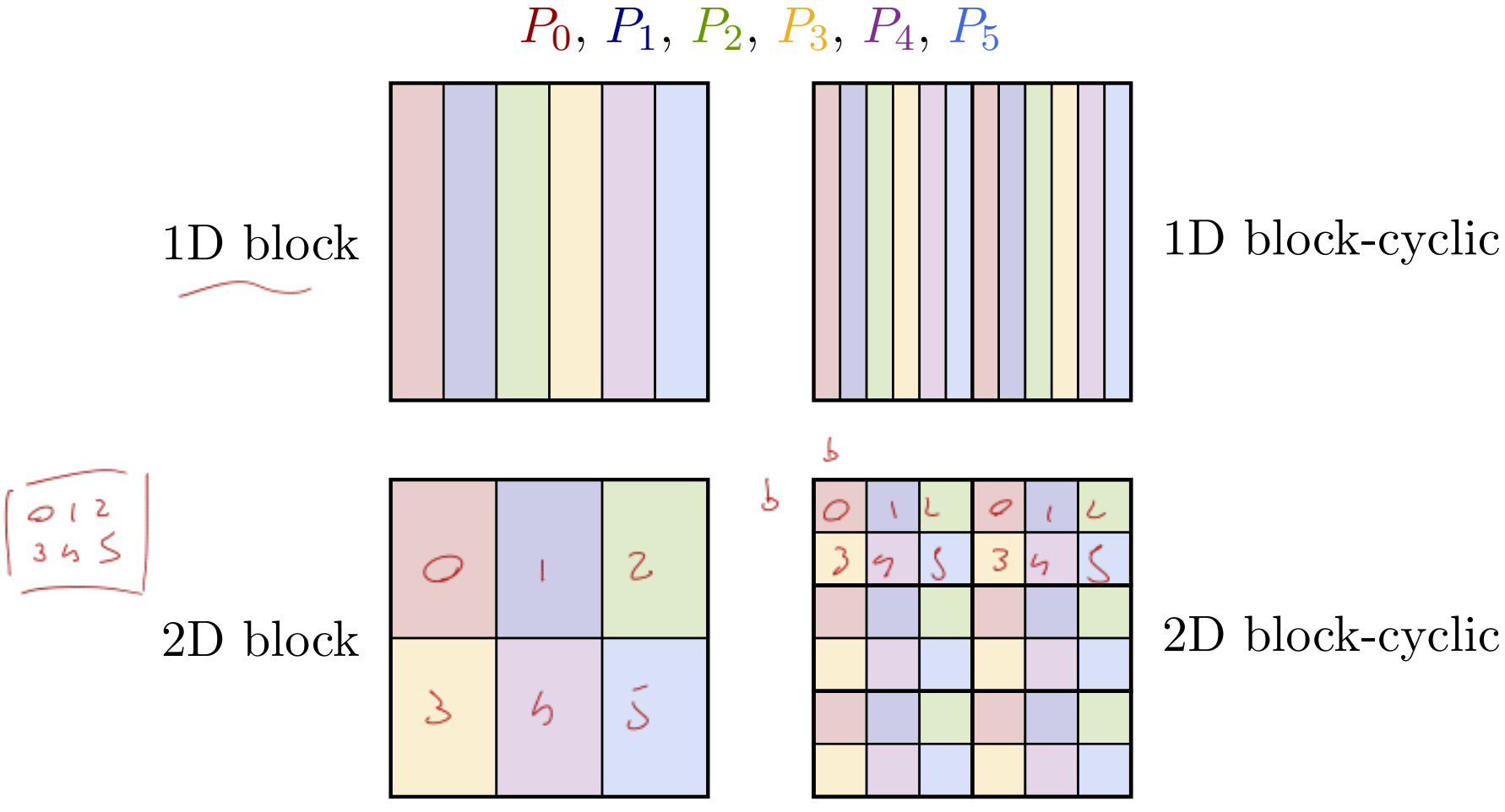
All-to-all: butterfly all-to-all

The size of the message is equal to $m/2$ in all steps

$$T_{ata,bfl} = \sum_{i=1}^{\log_2(p)} \left(\alpha + \beta \frac{m}{2}\right) = \log_2(p) \left(\alpha + \beta \frac{m}{2}\right)$$

Data distribution for dense LA

Data distribution schemes are clearly more or less effective depending on the operation to be done on the data but commonly schemes that are reasonably effective on a wide range of algorithms are chosen:



Parallel matrix-matrix multiply

Assume the matrix product $C = A * B$ where A and B are square matrices distributed in a 2D block fashion (non-cyclic) over a square grid of processors of size $r \times c = 3 \times 3$, i.e., $r = c = s = \sqrt{p}$

C_{00}	C_{01}	C_{02}
C_{10}	C_{11}	C_{12}
C_{20}	C_{21}	C_{22}

$=$

A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

$*$

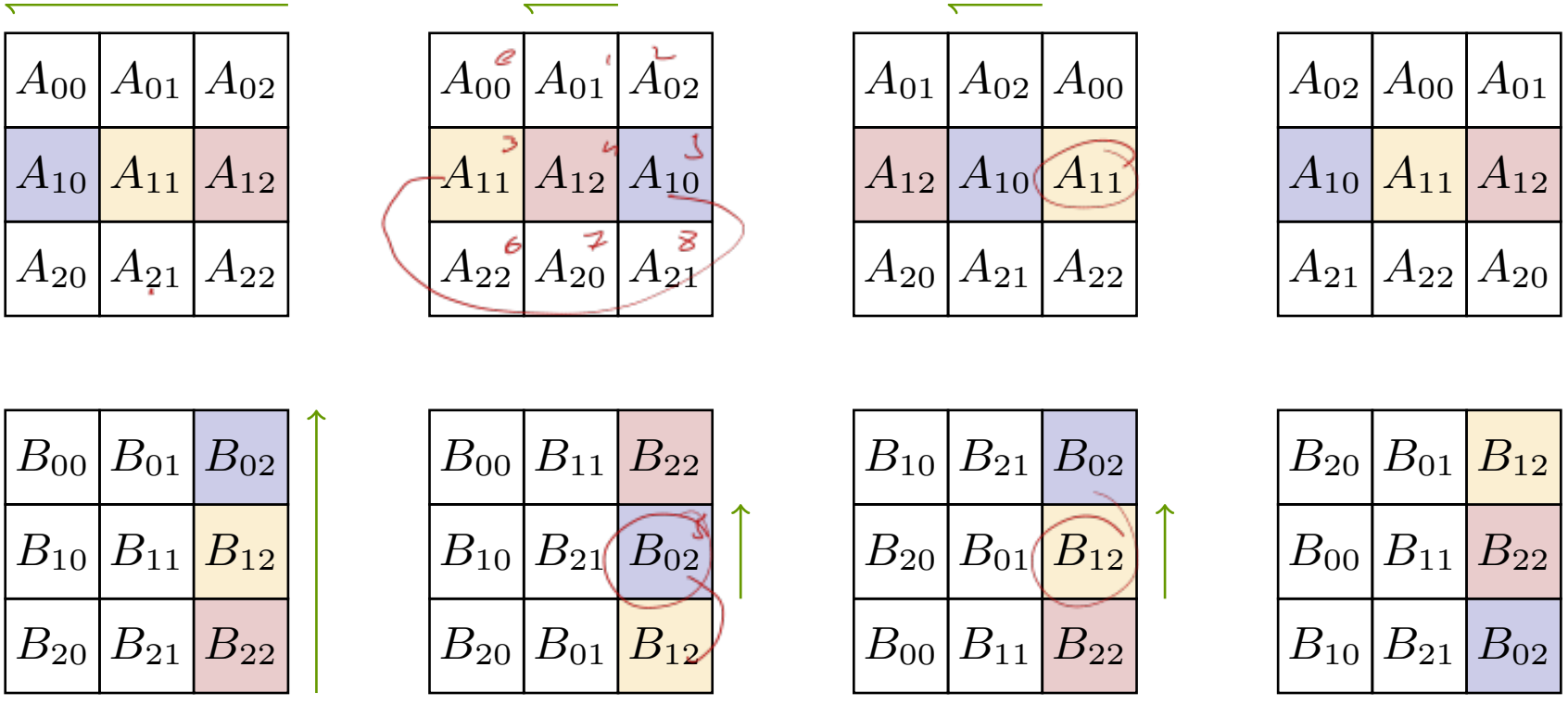
B_{00}	B_{01}	B_{02}
B_{10}	B_{11}	B_{12}
B_{20}	B_{21}	B_{22}

Note that, for example, C_{12} can be computed as

$$C_{12} = A_{10} * B_{02} + A_{11} * B_{12} + A_{12} * B_{22}$$

Parallel matrix-matrix multiply: Cannon's algorithm

How does it work:



Initial A and B | After skew | After shift 1 | After shift 2

$$C_{12} = A_{10} * B_{02} + A_{11} * B_{12} + A_{12} * B_{22}$$

All these computations are done on the node owning C_{12}

Parallel matrix-matrix multiply: Cannon's algorithm

Algorithm 5 Cannon's algorithm

1: **for all** $i = 0, \ldots, s-1$ **do**
2: left-circular-shift row i of A by i steps
3: **end for**
4: **for all** $j = 0, \ldots, s-1$ **do**
5: up-circular-shift col j of B by j steps
6: **end for**
7: **for** $k = 0, \ldots, s-1$ **do**
8: **for all** $i = 0, \ldots, s-1$ and $j = 0, \ldots, s-1$ **do**
9: $C_{ij} = C_{ij} + A_{ij} * B_{ij}$
10: left-circular-shift all rows of A by 1 step
11: up-circular-shift all cols of B by 1 step
12: **end for**
13: **end for**

Parallel matrix-matrix multiply: Cannon's algorithm

Algorithm 5 Cannon's algorithm

1: **for all** $i = 0, \ldots, s-1$ **do**
2: left-circular-shift row i of A by i steps
3: **end for**
4: **for all** $j = 0, \ldots, s-1$ **do**
5: up-circular-shift col j of B by j steps
6: **end for**
7: **for** $k = 0, \ldots, s-1$ **do**
8: **for all** $i = 0, \ldots, s-1$ and $j = 0, \ldots, s-1$ **do**
9: $C_{ij} = C_{ij} + A_{ij} * B_{ij}$
10: left-circular-shift all rows of A by 1 step
11: up-circular-shift all cols of B by 1 step
12: **end for**
13: **end for**

$s(\alpha + \beta n^2/p)$

$2\gamma(n/\sqrt{p})^3$

$(\alpha + \beta n^2/p)$

Parallel matrix-matrix multiply: Cannon's algorithm

Cost of the parallel algorithm:

$$
\begin{aligned}
T(n,p) = \quad & s(\alpha + \beta n^2/p) && \text{// skew } A \\
& s(\alpha + \beta n^2/p) && \text{// skew } B \\
& s(2\gamma(n/\sqrt{p})^3) && \text{// flops} \\
& s(\alpha + \beta n^2/p) && \text{// shift } A \\
& s(\alpha + \beta n^2/p) && \text{// shift } B
\end{aligned}
$$

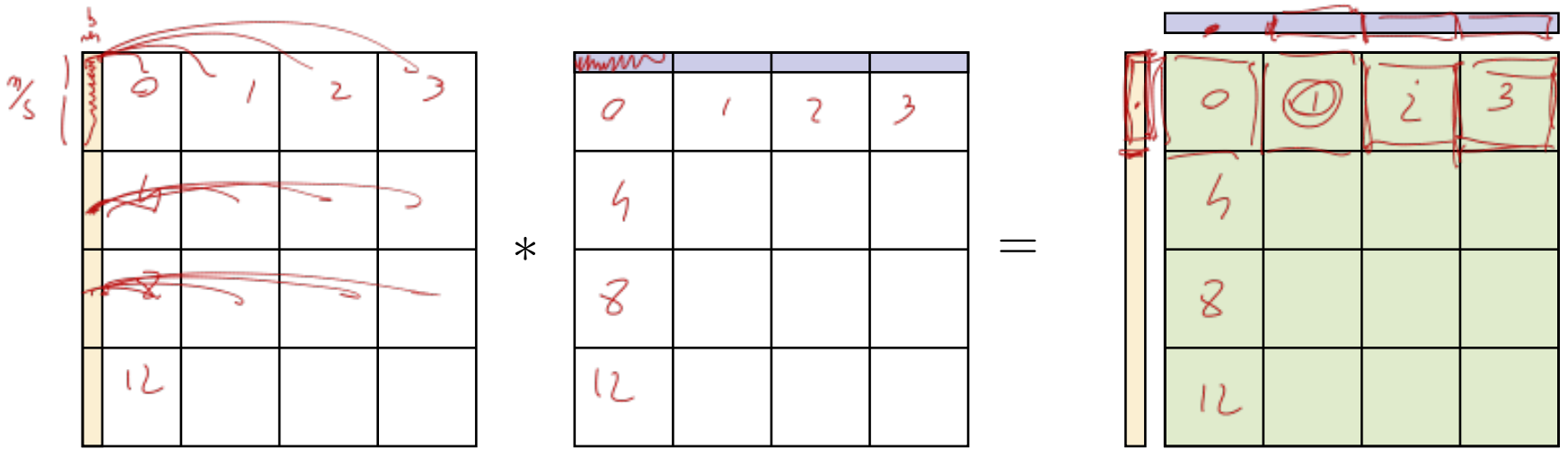
$$
T(n,p) = \frac{2n^3}{p}\gamma + 4s\alpha + \frac{4n^2}{s}\beta = T(n,1)/p + O(n,p)/p
$$

Exercise: define the iso-efficiency function for the Cannon's algorithm

- Generalizes well to the case of a 2D block cyclic distribution
- Only works with square grids of processors

The SUMMA algorithm

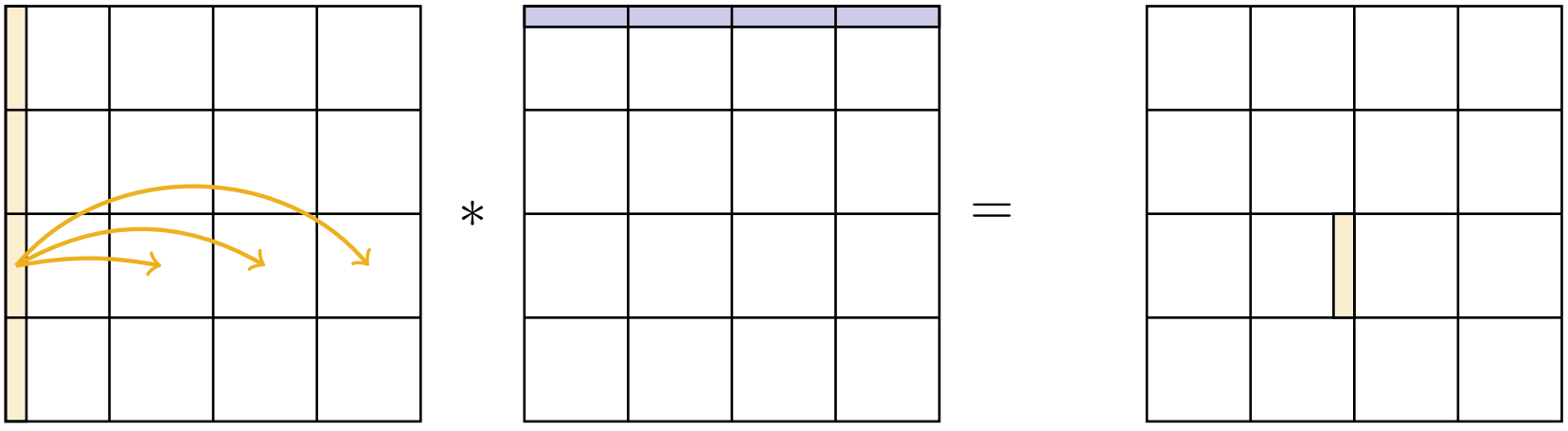
SUMMA was proposed by Agarwal et al. [1] and Geijn et al. [7]. Assume a 2D block (non-cyclic) distribution



SUMMA computes the matrix product as a sequence of $l = 1, \ldots, n/b$ *rank-b updates* of the type $C+ = A_{*l} * B_{l*}$. From the point of view of process $r \times c$:

The SUMMA algorithm

SUMMA was proposed by Agarwal et al. [1] and Geijn et al. [7]. Assume a 2D block (non-cyclic) distribution

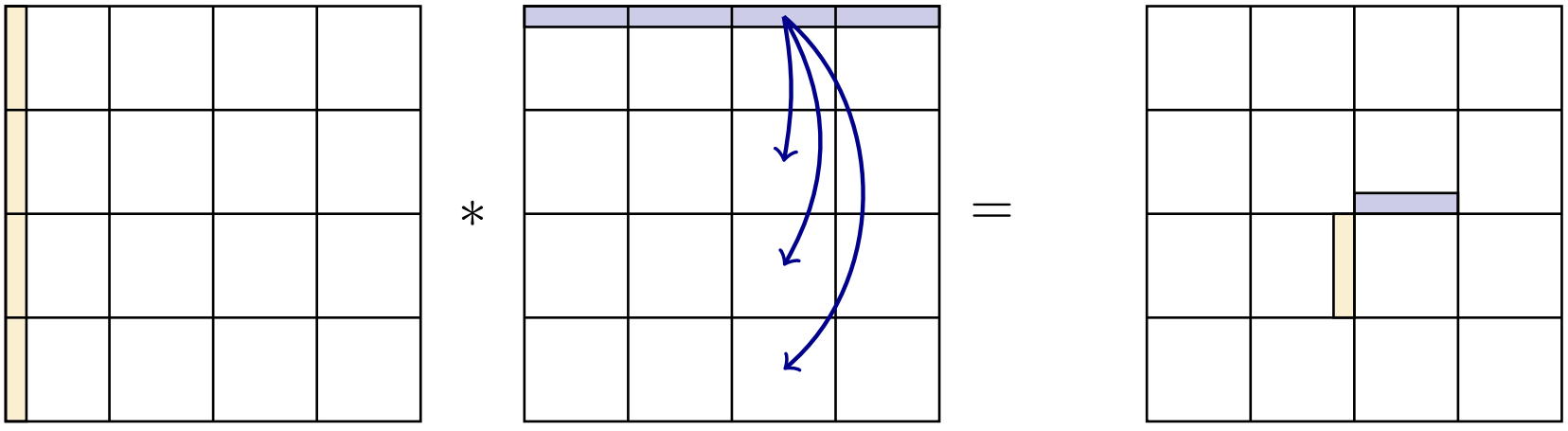


SUMMA computes the matrix product as a sequence of $l = 1, \ldots, n/b$ *rank-b updates* of the type $C+ = A_{*l} * B_{l*}$. From the point of view of process $r \times c$:

1. Broadcast A_{rl} over grid row

The SUMMA algorithm

SUMMA was proposed by Agarwal et al. [1] and Geijn et al. [7]. Assume a 2D block (non-cyclic) distribution

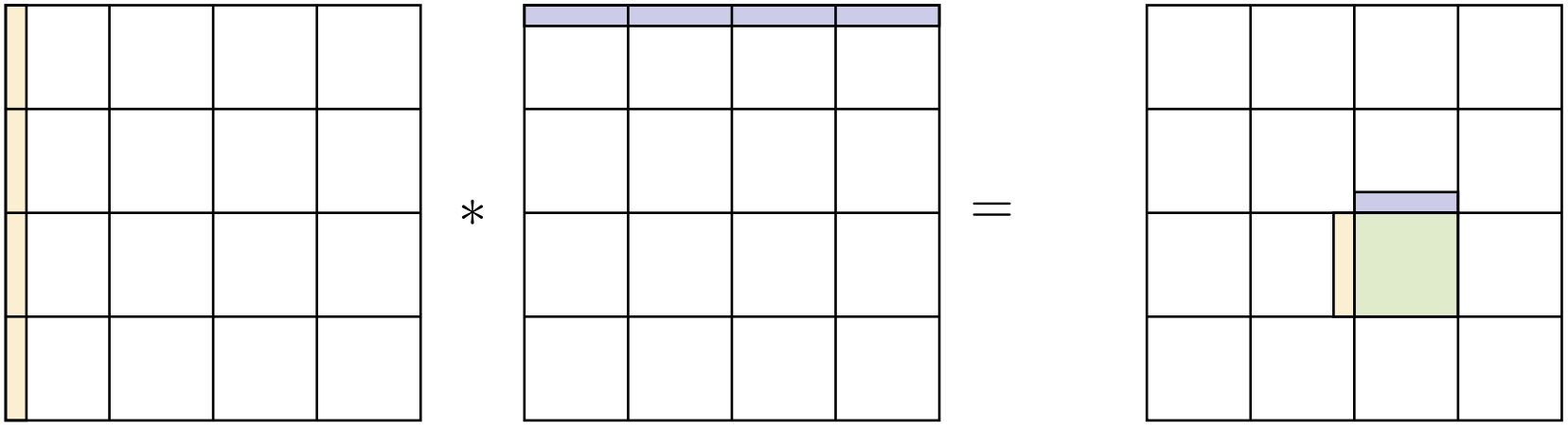


SUMMA computes the matrix product as a sequence of $l = 1, \ldots, n/b$ *rank-b updates* of the type $C+ = A_{*l} * B_{l*}$. From the point of view of process $r \times c$:

1. Broadcast A_{rl} over grid row
2. Broadcast B_{lc} over grid col

The SUMMA algorithm

SUMMA was proposed by Agarwal et al. [1] and Geijn et al. [7]. Assume a 2D block (non-cyclic) distribution



SUMMA computes the matrix product as a sequence of $l = 1, \ldots, n/b$ *rank-b updates* of the type $C+ = A_{*l} * B_{l*}$. From the point of view of process $r \times c$:

1. Broadcast A_{rl} over grid row
2. Broadcast B_{lc} over grid col
3. Compute $C_{rc}+ = A_{rl} * B_{lc}$ locally

Parallel matrix-matrix multiply: SUMMA algorithm

Assume A is of size $m \times k$, B of size $k \times n$, C of size $m \times n$.

Algorithm 6 SUMMA algorithm

1: **for** $l = 1, k/b$ **do**
2: **for all** $r, c = 0, \ldots, s-1$ **do**
3: broadcast A_{rl} over $r \times *$ (grid row)
4: broadcast B_{lc} over $* \times c$ (grid col)
5: **end for**
6: **for all** $r, c = 0, \ldots, s-1$ **do**
7: $C_{rc} = C_{rc} + A_{rl} * B_{lc}$
8: **end for**
9: **end for**

Parallel matrix-matrix multiply: SUMMA algorithm

Assume A is of size $m \times k$, B of size $k \times n$, C of size $m \times n$.

Algorithm 6 SUMMA algorithm

1: **for** $l = 1, k/b$ **do**
2: **for all** $r, c = 0, \ldots, s-1$ **do**
3: broadcast A_{rl} over $r \times *$ (grid row) ← $\log(s)(\alpha + \beta mb/s)$
4: broadcast B_{lc} over $* \times c$ (grid col) ← $\log(s)(\alpha + \beta nb/s)$
5: **end for**
6: **for all** $r, c = 0, \ldots, s-1$ **do**
7: $C_{rc} = C_{rc} + A_{rl} * B_{lc}$ ← $2\gamma bmn/p$
8: **end for**
9: **end for**

$$T_{\text{SU-C}}(m,n,k,p) = 2\frac{mnk}{p}\gamma + 2\log(s)\frac{k}{b}\alpha + \log(s)\frac{k(m+n)}{s}\beta$$
$$= T(n,1)/p + O(n,p)/p$$

Optimum for $b = k/s$ but high memory consumption

Parallel matrix-matrix multiply: SUMMA algorithm

Algorithm 7 SUMMA algorithm with lookahead

broadcast A_{r1} over $r \times *$ (grid row)
broadcast B_{1c} over $* \times c$ (grid col)
for $l = 1, k/b - 1$ **do**
  do concurrently
    for all $r, c = 0, \ldots, s-1$ **do**
      broadcast $A_{r,l+1}$ over $r \times *$ (grid row)
      broadcast $B_{l+1,c}$ over $* \times c$ (grid col)
    end for
  and
    for all $r, c = 0, \ldots, s-1$ **do**
      $C_{rc} = C_{rc} + A_{rl} * B_{lc}$
    end for
  end do concurrently
end for
$C_{rc} = C_{rc} + A_{r,k/b} * B_{k/b,c}$

Parallel matrix-matrix multiply: SUMMA algorithm

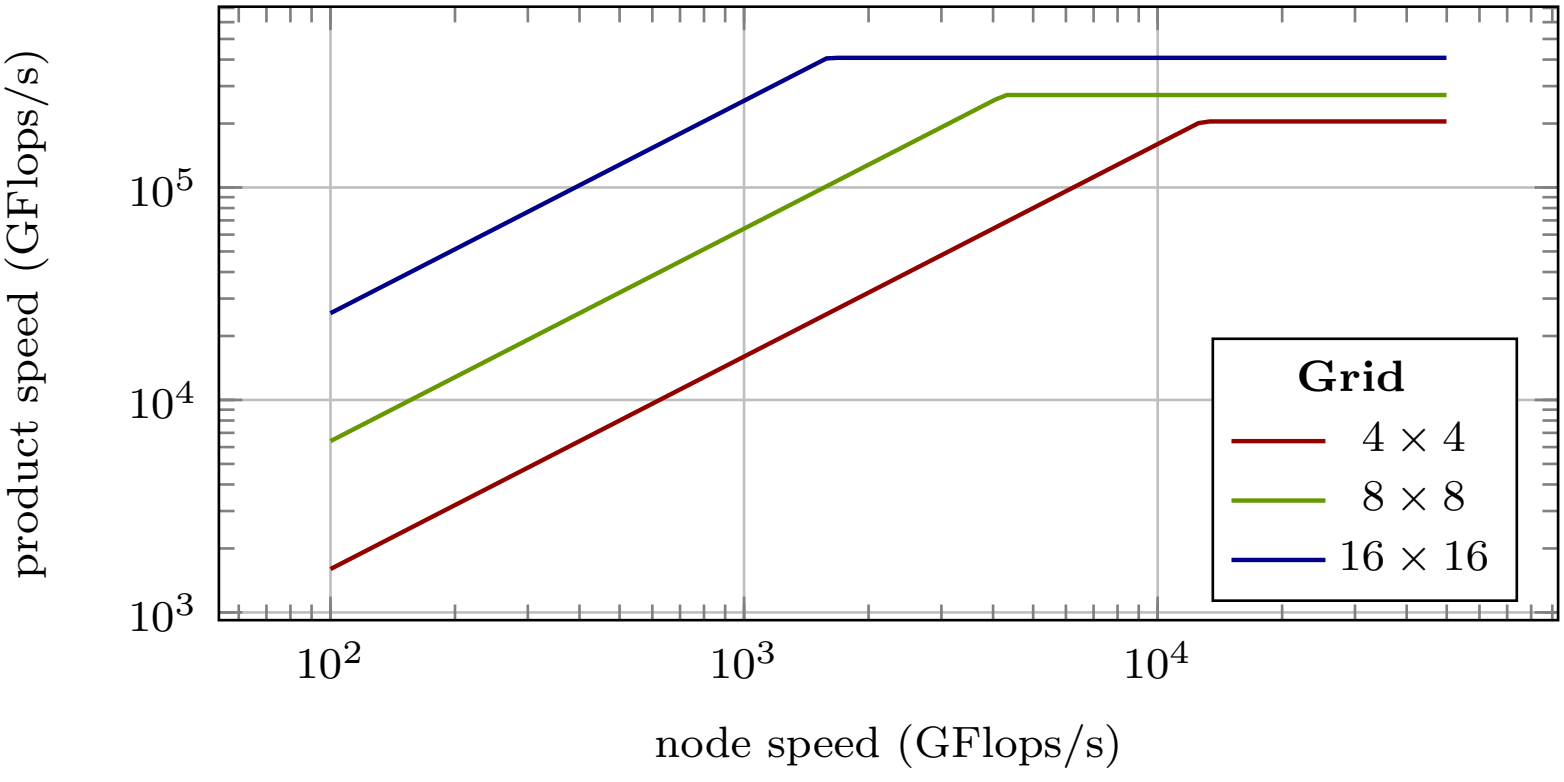
Non-blocking collective communications can be used to implement a lookahead technique where data for iteration $k+1$ is broadcast at the same time as computations of iteration k

$$
\begin{aligned}
T_{\text{SU-C}}(m,n,k,p) \quad = \quad & \log(s)(2\alpha + \beta(m+n)b/s) + \\
& \left(\tfrac{k}{b} - 1\right) \max(\\
& \quad \log(s)(2\alpha + \beta(m+n)b/s), \\
& \quad 2\gamma bmn/p) + \\
& 2\gamma bmn/p
\end{aligned}
$$

Parallel matrix-matrix multiply: SUMMA algorithm

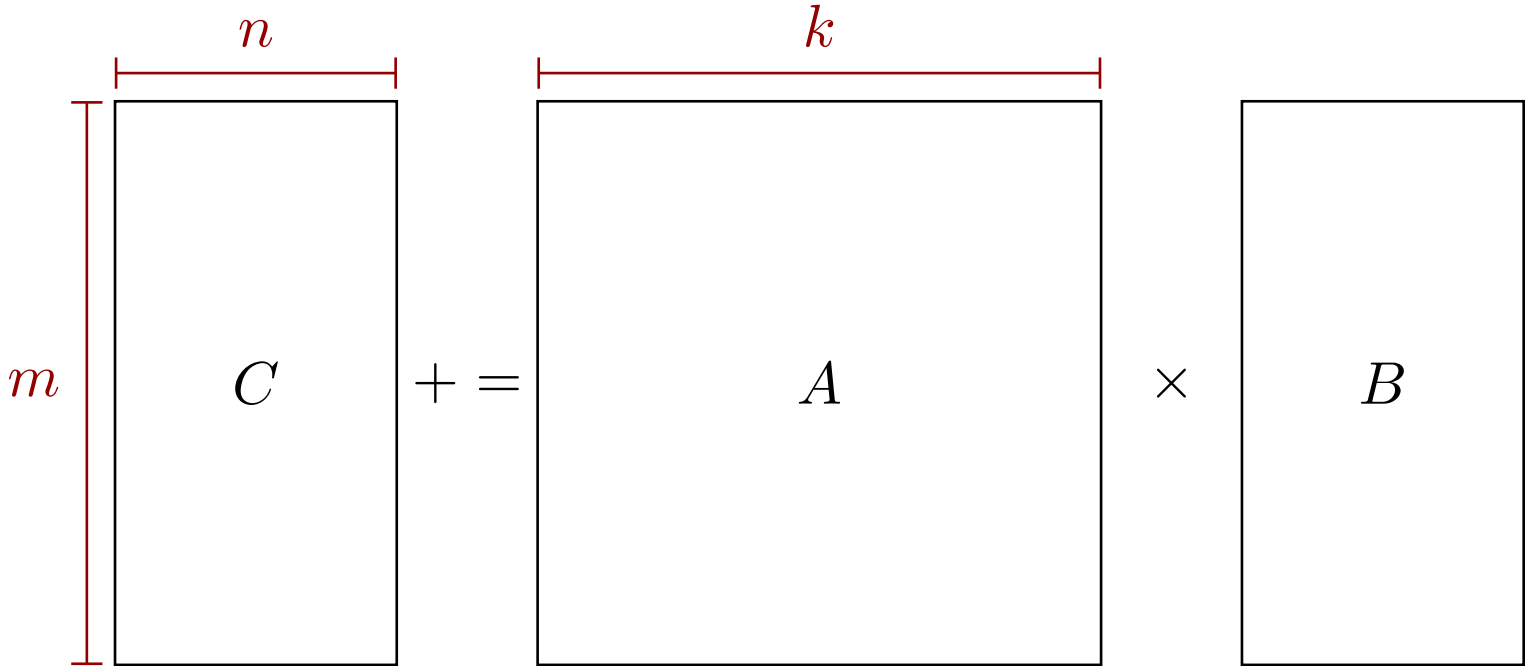
Parameters:

- n : 32768
- b : 32 (irrelevant if $\ll n$)
- α : 100 (Infiniband HDR)
- β : 64/400 (Infiniband HDR, assuming double precision)



Parallel matrix-matrix multiply: SUMMA algorithm

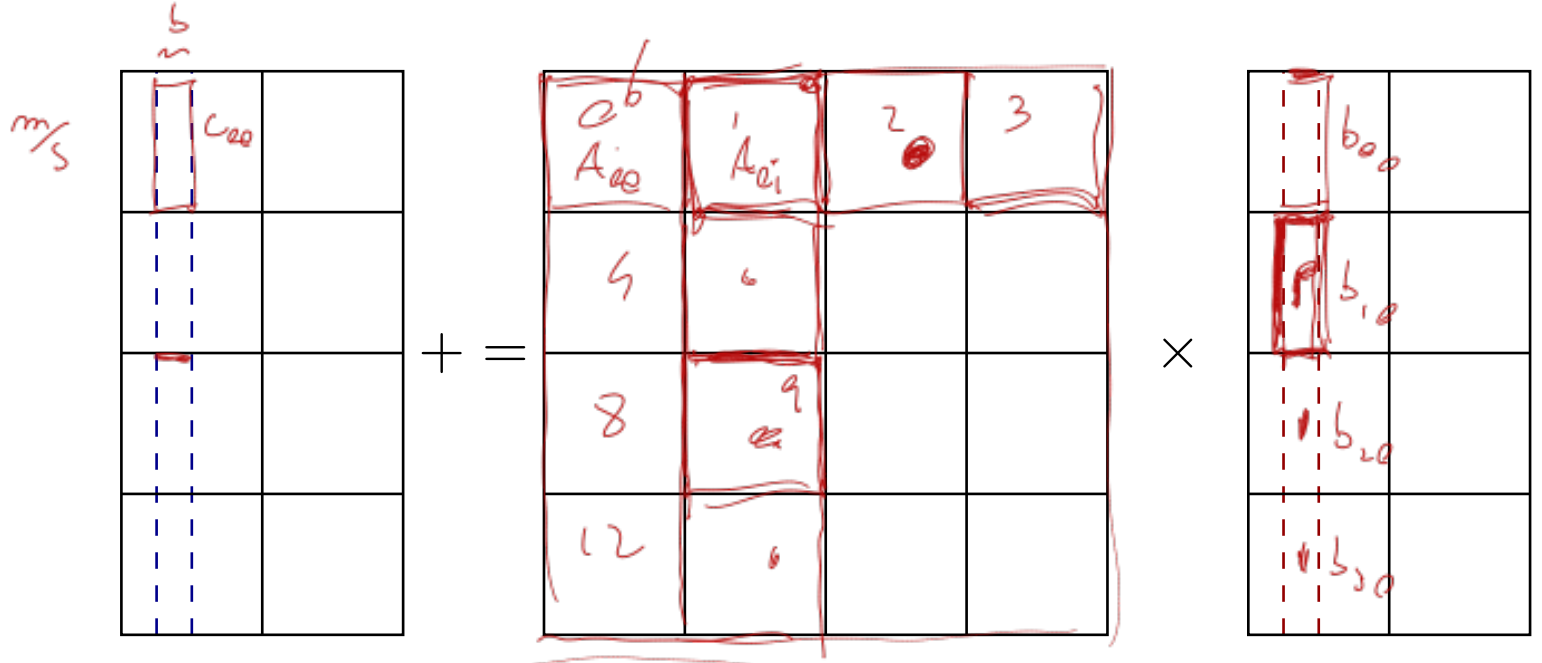
In the presented SUMMA algorithm, the C matrix stays in place and only A and B are transferred over the network. This version is called stationary-C. This is clearly not a good choice if A is of size $m \times k$, B of size $k \times n$, C of size $m \times n$ and $n \ll m, k$

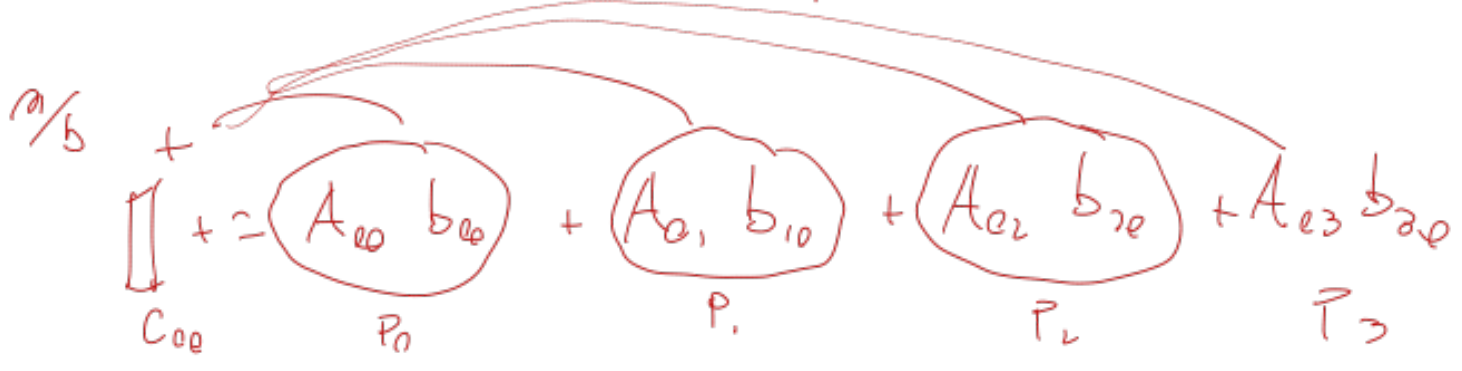


In this case it would be much more efficient to keep A in place and transfer B and C over the network

Parallel matrix-matrix multiply: SUMMA algorithm

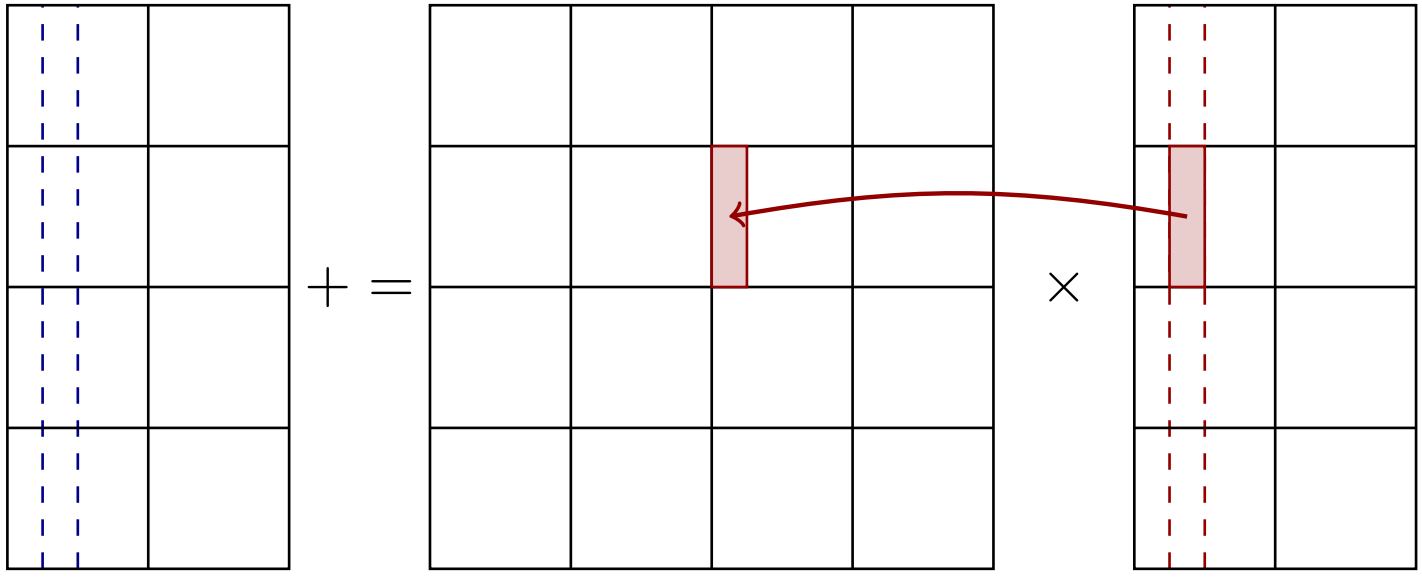
stationary$A$ SUMMA algorithm: proceeds in n/b steps l where, at each step the whole A matrix is multiplied by block-column B_{*l} yielding block-column C_{*l}





Parallel matrix-matrix multiply: SUMMA algorithm

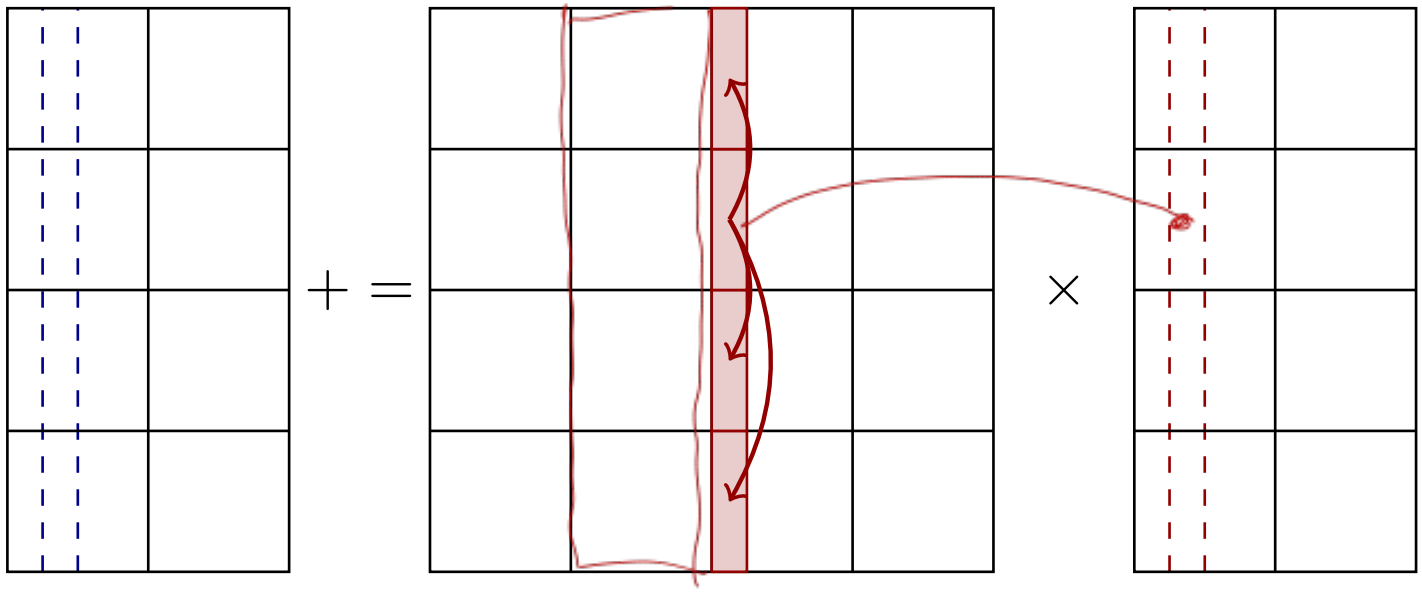
stationary$A$ SUMMA algorithm: proceeds in n/b steps l where, at each step the whole A matrix is multiplied by block-column B_{*l} yielding block-column C_{*l}



1. send block B_{cl} to node $c \times c$

Parallel matrix-matrix multiply: SUMMA algorithm

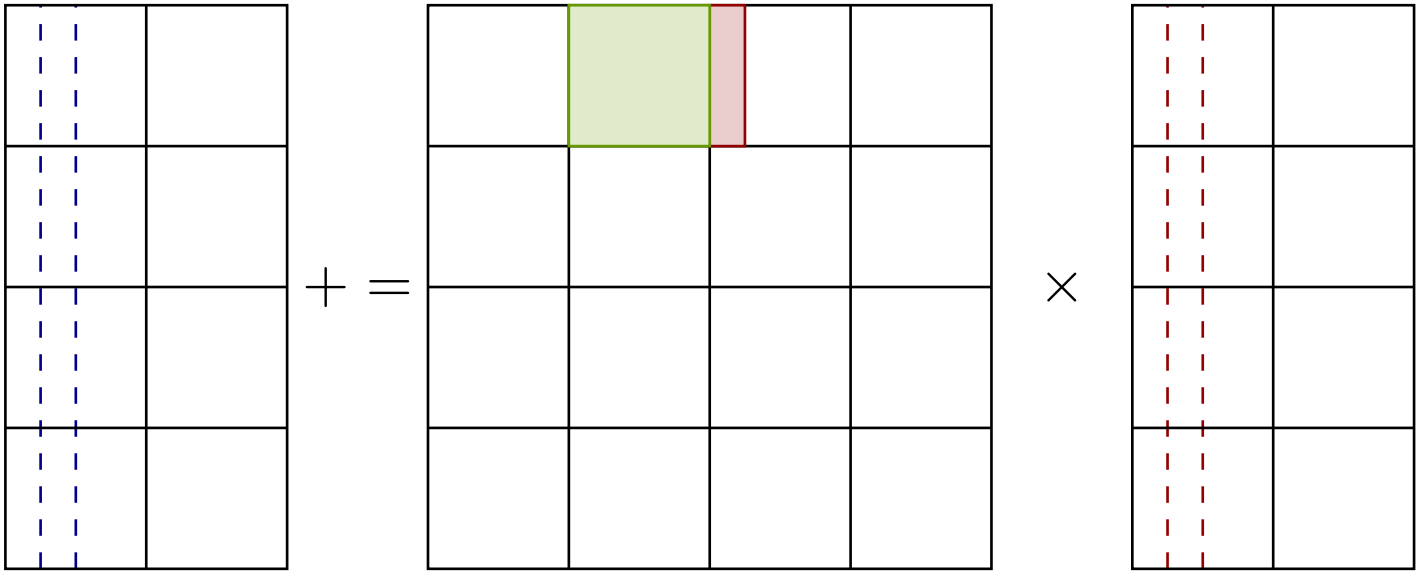
stationary$A$ SUMMA algorithm: proceeds in n/b steps l where, at each step the whole A matrix is multiplied by block-column B_{*l} yielding block-column C_{*l}



1. send block B_{cl} to node $c \times c$
2. broadcast block B_{cl} over grid column $* \times c$

Parallel matrix-matrix multiply: SUMMA algorithm

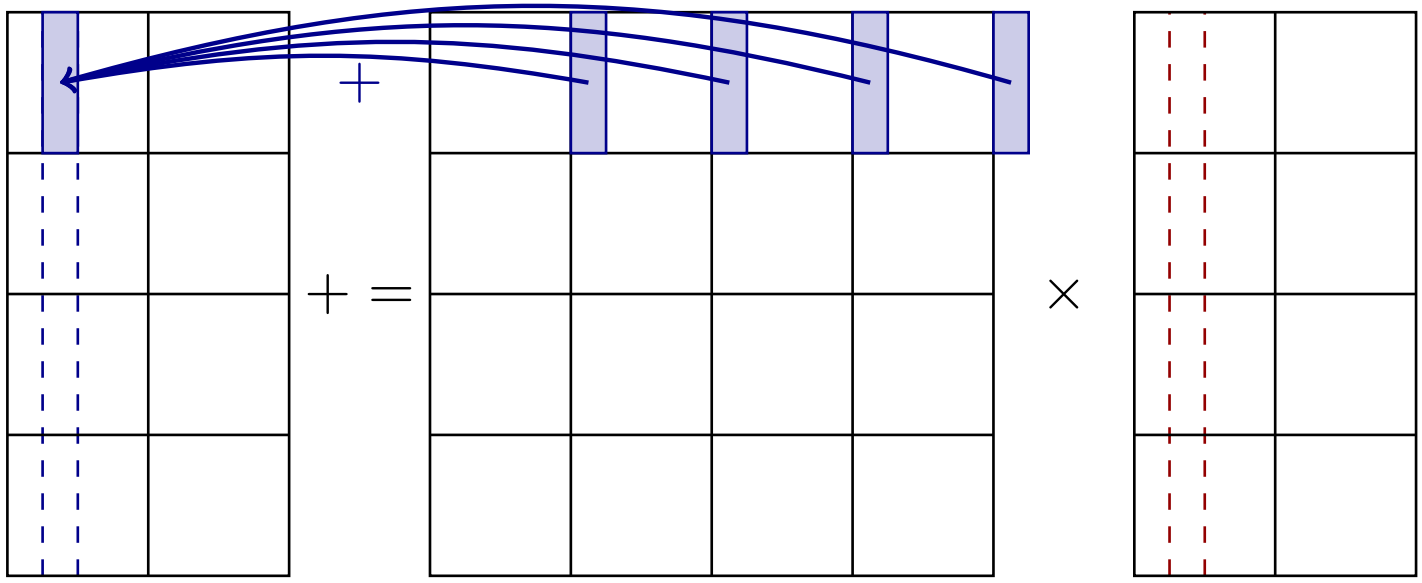
stationary$A$ SUMMA algorithm: proceeds in n/b steps l where, at each step the whole A matrix is multiplied by block-column B_{*l} yielding block-column C_{*l}



1. send block B_{cl} to node $c \times c$
2. broadcast block B_{cl} over grid column $* \times c$
3. compute $C^c_{rl} = A_{rc} * B_{cl}$ locally

Parallel matrix-matrix multiply: SUMMA algorithm

stationary$A$ SUMMA algorithm: proceeds in n/b steps l where, at each step the whole A matrix is multiplied by block-column B_{*l} yielding block-column C_{*l}



1. send block B_{cl} to node $c \times c$
2. broadcast block B_{cl} over grid column $* \times c$
3. compute $C^c_{rl} = A_{rc} * B_{cl}$ locally
4. reduce all the C^c_{rl} in row $r \times *$ onto C_{rl}

Parallel matrix-matrix multiply: SUMMA algorithm

Algorithm 8 stationary-A SUMMA

1: **for** $l = 1, n/b - 1$ **do**
2: **for all** $c = 0, \ldots, s-1$ **do**
3: send B_{cl} to node $c \times c$
4: broadcast B_{cl} over grid column $* \times c$
5: **end for**
6: **for all** $r, c = 0, \ldots, s-1$ **do**
7: $C_{rl}^{c} = A_{rc} * B_{cl}$
8: **end for**
9: **for all** $r = 0, \ldots, s-1$ **do**
10: sum-reduce C_{rl}^{c} over C_{rl}
11: **end for**
12: **end for**

Parallel matrix-matrix multiply: SUMMA algorithm

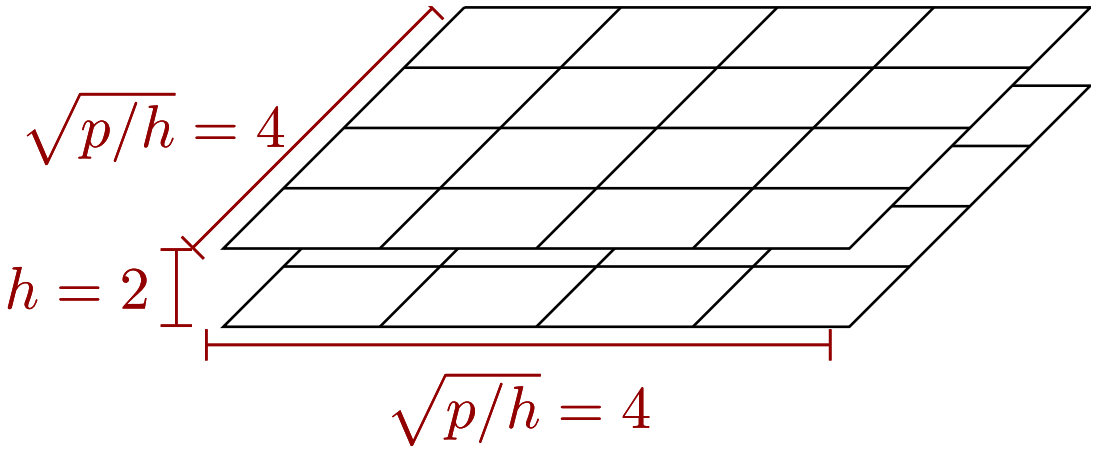
Algorithm 8 stationary-A SUMMA

1: **for** $l = 1, n/b - 1$ **do**
2: **for all** $c = 0, \ldots, s-1$ **do**
3: send B_{cl} to node $c \times c$ ← $(\alpha + \beta kb/s)$
4: broadcast B_{cl} over grid column $* \times c$ ← $log(s)(\alpha + \beta kb/s)$
5: **end for**
6: **for all** $r, c = 0, \ldots, s-1$ **do**
7: $C_{rl}^c = A_{rc} * B_{cl}$ ← $2\gamma bmk/p$
8: **end for**
9: **for all** $r = 0, \ldots, s-1$ **do**
10: sum-reduce C_{rl}^c over C_{rl} ← $log(s)(\alpha + (\beta + \gamma)mb/s)$
11: **end for**
12: **end for**

Except lower order terms, the cost is the same as stationary-C if all matrices are square

Parallel matrix-matrix multiply: 2.5D SUMMA

2.5D algorithms are designed to achieve better parallelism at the cost of an higher memory consumption.
Let's assume our p computer nodes are arranged in a three-dimentional grid of height h



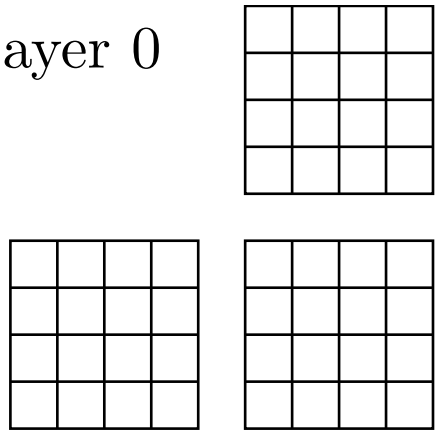
Let's also assume that matrices A, B and C are initially distributed with a 2D-block (non-cyclic) scheme on the lowest level of the grid, i.e., $r \times c \times 0$ for $r, c = 1, \ldots, \sqrt{p/h}$

Parallel matrix-matrix multiply: 2.5D SUMMA

Assume $m, k = n$ for simplicity and $p = 32$, $h = 2$

2.5D stationary-C SUMMA

Layer 0

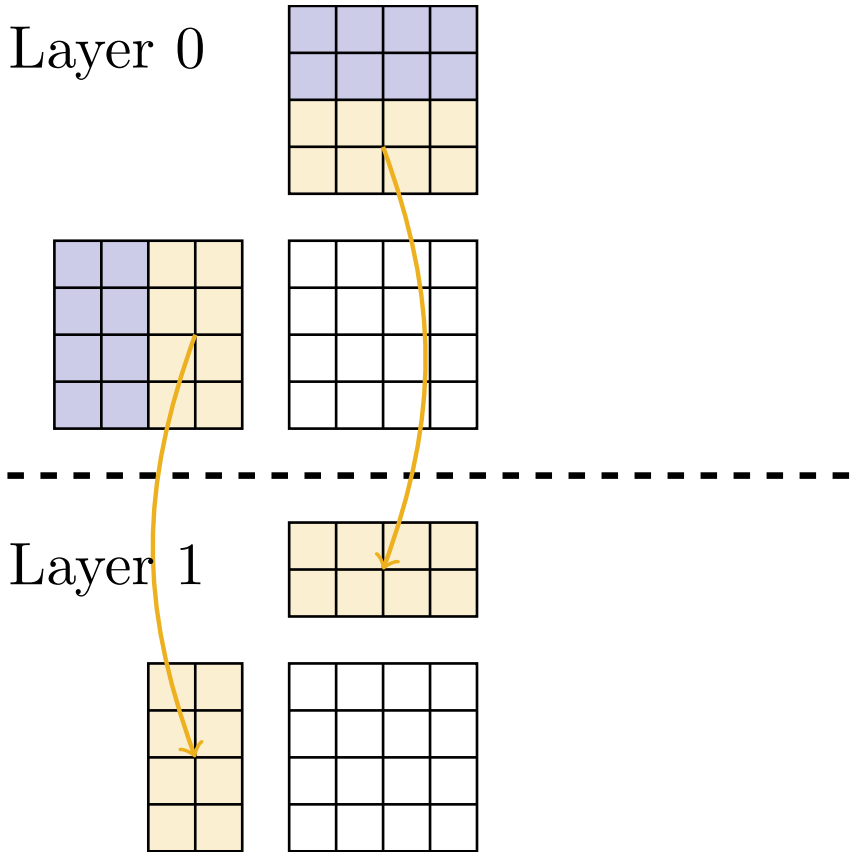


Layer 1

Parallel matrix-matrix multiply: 2.5D SUMMA

Assume $m, k = n$ for simplicity and $p = 32$, $h = 2$

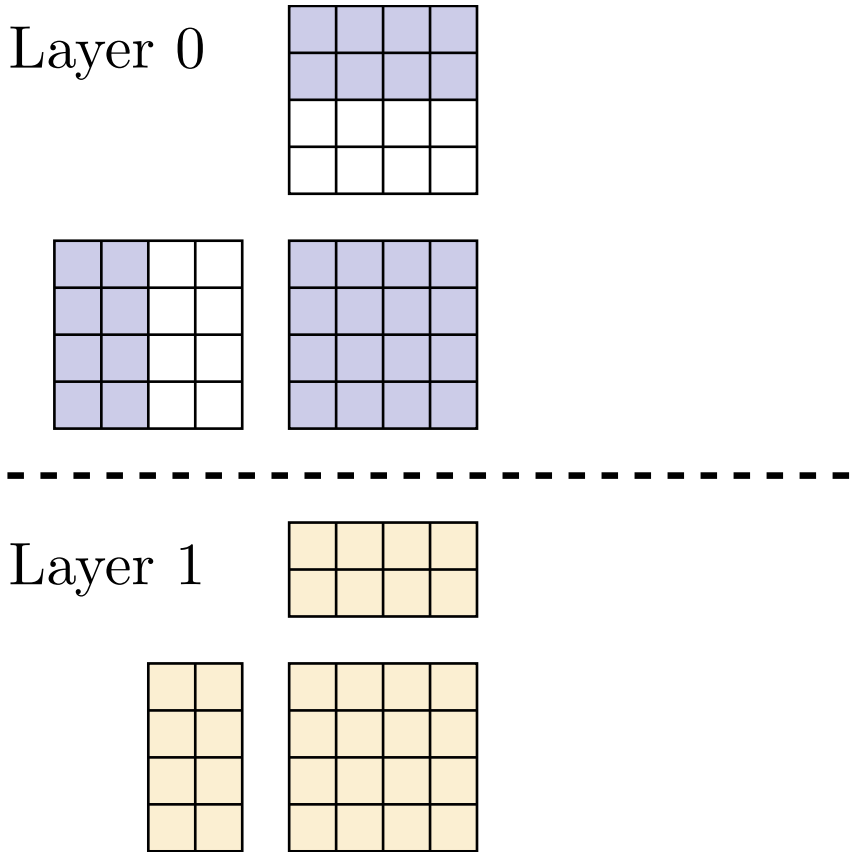
2.5D stationary-C SUMMA



1. copy A and B to upper layers

Parallel matrix-matrix multiply: 2.5D SUMMA

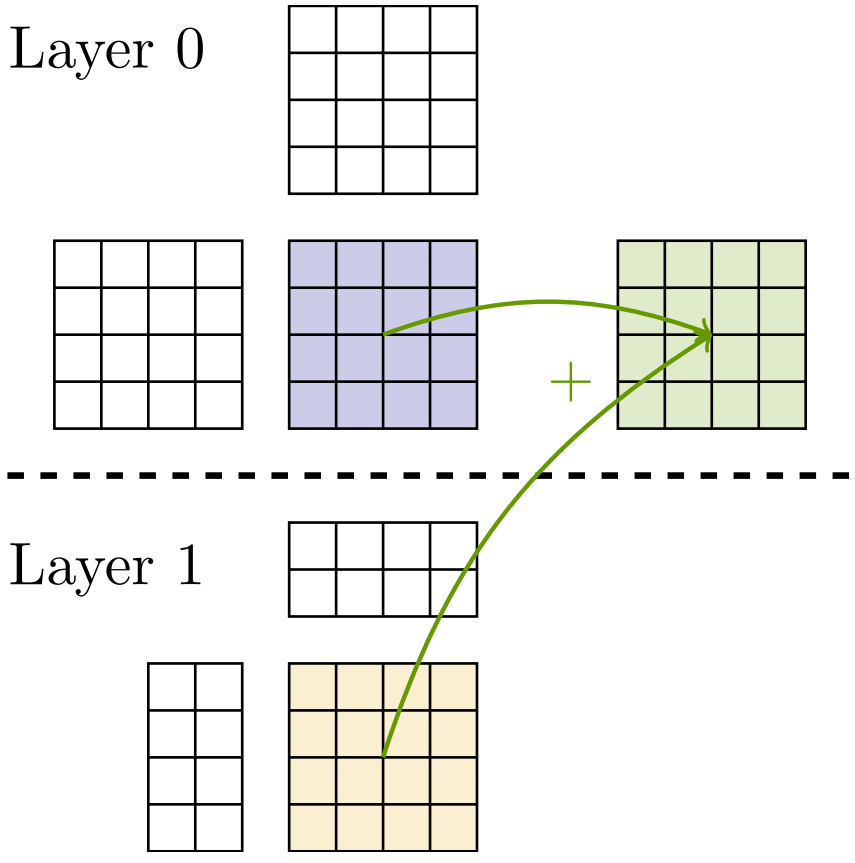
Assume $m, k = n$ for simplicity and $p = 32$, $h = 2$
2.5D stationary-C SUMMA



1. copy A and B to upper layers
2. make partial product locally on each layer using SUMMA

Parallel matrix-matrix multiply: 2.5D SUMMA

Assume $m, k = n$ for simplicity and $p = 32$, $h = 2$
2.5D stationary-C SUMMA



1. copy A and B to upper layers
2. make partial product locally on each layer using SUMMA
3. sum-reduce partial results onto layer 0

Parallel matrix-matrix multiply: 2.5D SUMMA

Assume $m, k = n$ for simplicity

Algorithm 9 2.5D stationary-C SUMMA

1: **for all** $r, c = 0, \ldots, s-1$ **do**
2: send $A_{r,c}$ from node $r \times c \times 0$ to node $r \times c \times c/h$
3: send $B_{r,c}$ from node $r \times c \times 0$ to node $r \times c \times r/h$
4: **end for**
5: **for all** $l = 0, \ldots, h-1$ **do**
6: compute $C^l += A_{*,x} * B_{x,*}$,
7: where $x = ln/h : (l+1)n/h - 1$
8: using SUMMA stationary-C
9: **end for**
10: **for all** $r, c = 0, \ldots, s-1$ **do**
11: sum-reduce $C^l_{rc},\ l = 0, \ldots, h-1$ over C_{rc}
12: **end for**

Parallel matrix-matrix multiply: 2.5D SUMMA

Assume $m, k = n$ for simplicity

Algorithm 9 2.5D stationary-C SUMMA

1: **for all** $r, c = 0, \ldots, s-1$ **do**
2: send $A_{r,c}$ from node $r \times c \times 0$ to node $r \times c \times c/h$
3: send $B_{r,c}$ from node $r \times c \times 0$ to node $r \times c \times r/h$
4: **end for**
5: **for all** $l = 0, \ldots, h-1$ **do**
6: compute $C^l += A_{*,x} * B_{x,*}$,
7: where $x = ln/h : (l+1)n/h - 1$
8: using SUMMA stationary-C
9: **end for**
10: **for all** $r, c = 0, \ldots, s-1$ **do**
11: sum-reduce C^l_{rc}, $l = 0, \ldots, h-1$ over C_{rc}
12: **end for**

$(\alpha + \beta n^2 h/p)$

$T_{\text{SU-C}}(n, n, n/h, p/h)$

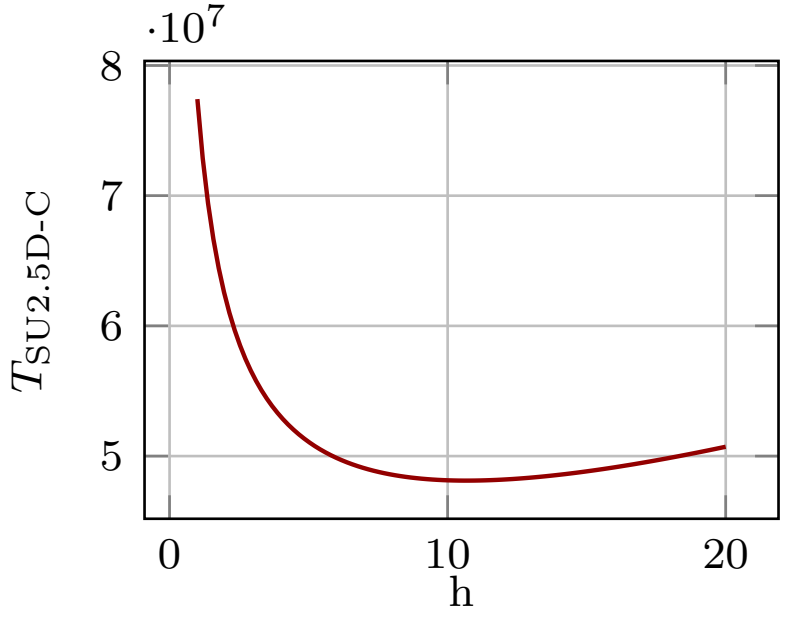
$\log(h)(\alpha + (\beta + \gamma)n^2 h/p)$

Parallel matrix-matrix multiply: 2.5D SUMMA

$$
\begin{aligned}
T_{\text{SU2.5D-C}}(n,n,n,p) &= 2(\alpha + \beta n^2 h/p) + \\
&\quad 2\gamma n^2 (n/h) h/p + \\
&\quad 2\log(s)(\alpha n/(hb) + \beta n^2/(hs)) + \\
&\quad \log(h)(\alpha + (\beta+\gamma) n^2 h/p) \\
&= T(n,1)/p + O(n,p)/p
\end{aligned}
$$

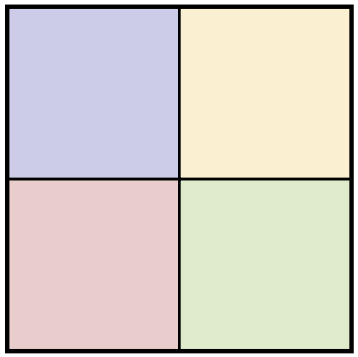
Optimum is reached for $h \simeq p^{1/3}$, i.e., a cubic grid

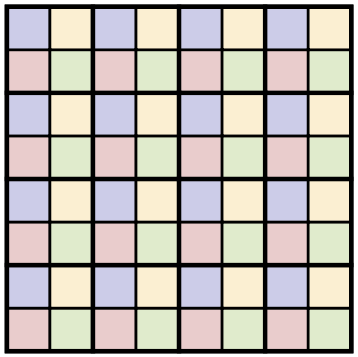
- n : 32768
- b : 32 (irrelevant if $\ll n$)
- α : 100 (Infiniband HDR)
- β : 64/400 (Infiniband HDR, assuming double precision)
- γ : 0.001 (1 TFlops/s)
- p : 2048



Parallel factorizations in distributed memory

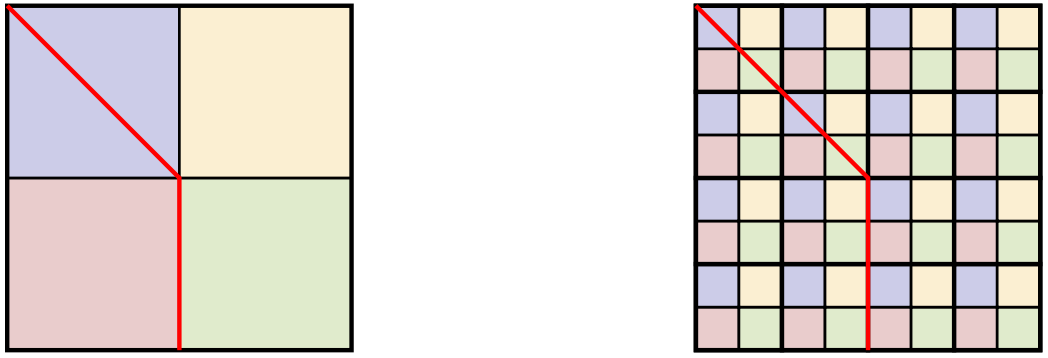
All the previous matrix-matrix product algorithms were presented using 2D block non-cyclic distribution. Although they can easily be generalized to the cyclic distribution this does not really change the efficiency because all the processes are involved in every step of the algorithm.
Unfortunately this is not the case for factorization algorithms:





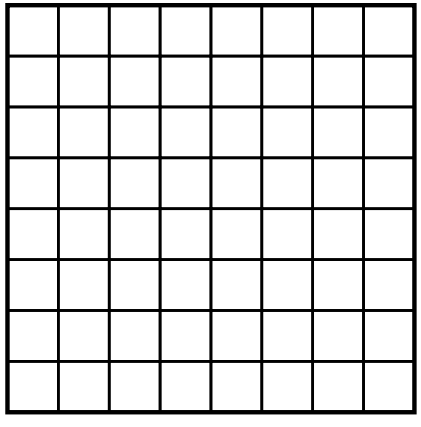
Parallel factorizations in distributed memory

All the previous matrix-matrix product algorithms were presented using 2D block non-cyclic distribution. Although they can easily be generalized to the cyclic distribution this does not really change the efficiency because all the processes are involved in every step of the algorithm.
Unfortunately this is not the case for factorization algorithms:



Because factorization algorithms proceed down along the diagonal of the matrix, every $n/(b\sqrt{p})$ steps an entire row and column of the process grid would be lost in a 2D block distribution. 2D block cyclic prevents this problem.

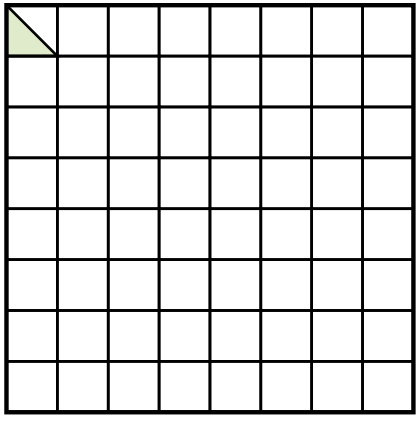
Parallel Cholesky factorization



Parallel Cholesky factorization

1. Factor diag block

$$\gamma(b^3/3)$$



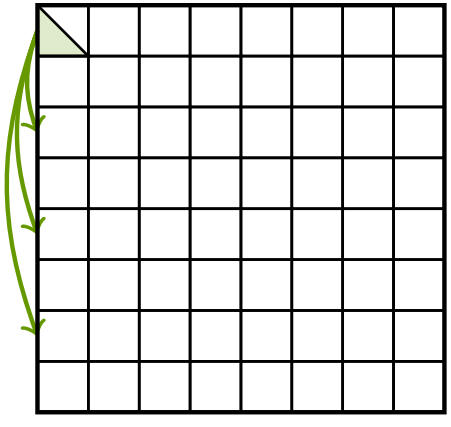
Parallel Cholesky factorization

1. Factor diag block

$$\gamma(b^3/3)$$

2. Broadcast diag block along column

$$\log(s)(\alpha + \beta b^2/2)$$



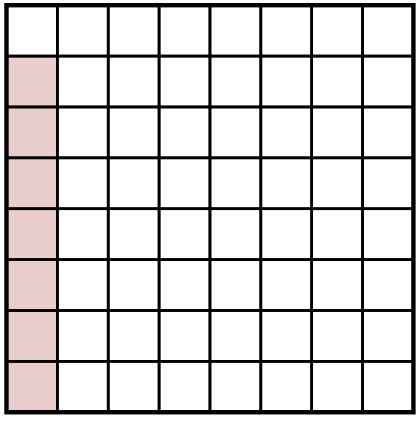
Parallel Cholesky factorization

1. Factor diag block

$$\gamma(b^3/3)$$

2. Broadcast diag block along column

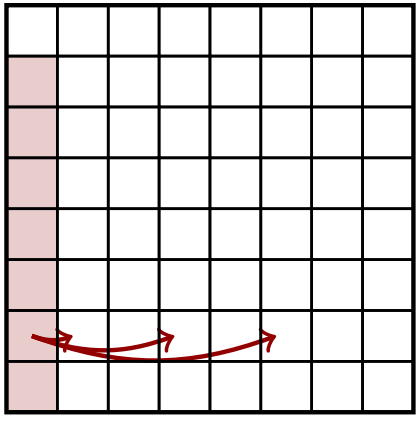
$$\log(s)(\alpha + \beta b^2/2)$$



3. Locally compute $L_{i1} = A_{i1} L_{11}^{-T}$

$$\gamma n b^2/s$$

Parallel Cholesky factorization



1. Factor diag block

$$\gamma(b^3/3)$$

2. Broadcast diag block along column

$$\log(s)(\alpha + \beta b^2/2)$$

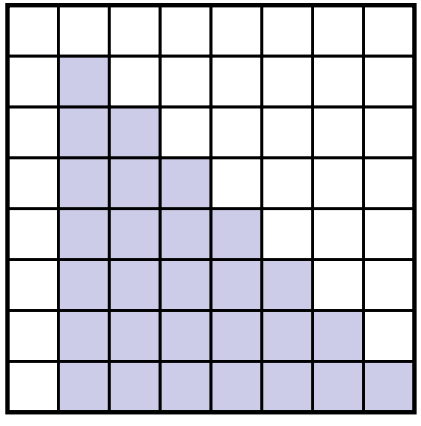
3. Locally compute $L_{i1} = A_{i1} L_{11}^{-T}$

$$\gamma n b^2/s$$

4. broadcast L_{*1} over rows and over columns

$$2\log(s)(\alpha + \beta nb/s)$$

Parallel Cholesky factorization



1. Factor diag block

$$\gamma(b^3/3)$$

2. Broadcast diag block along column

$$\log(s)(\alpha + \beta b^2/2)$$

3. Locally compute $L_{i1} = A_{i1} L_{11}^{-T}$

$$\gamma n b^2/s$$

4. broadcast L_{*1} over rows and over columns

$$2\log(s)(\alpha + \beta nb/s)$$

5. Compute $A_{ij} - = L_{i1} L_{j1}^T$ locally

$$\gamma n^2 b/p$$

Parallel Cholesky factorization

Algorithm 10 Parallel cholesky

1: **for** $k = 1, \ldots, n/b$ **do**
2: factorize $A_{kk} = L_{kk} L_{kk}^T$
3: broadcast L_{kk} over grid column $k\%s$
4: **for all** $i = k+1, \ldots, n/b$ **do**
5: compute $L_{ik} = A_{ik} L_{kk}^{-T}$
6: **end for**
7: **for all** $i = k+1, \ldots, n/b$ **do**
8: broacast L_{ik} over row $i\%s$
9: broacast L_{ik} over col $i\%s$
10: **end for**
11: **for all** $i = k+1, \ldots, n/b, j = k+1, \ldots, i$ **do**
12: compute $A_{ij} -= L_{ik} L_{jk}^T$
13: **end for**
14: **end for**

Parallel Cholesky factorization

Algorithm 10 Parallel cholesky

1: **for** $k = 1, \ldots, n/b$ **do**
2: factorize $A_{kk} = L_{kk} L_{kk}^T$ — $\gamma b^3/3$
3: broadcast L_{kk} over grid column $k\%s$ — $\log(s)(\alpha + \beta b^2/2)$
4: **for all** $i = k+1, \ldots, n/b$ **do**
5: compute $L_{ik} = A_{ik} L_{kk}^{-T}$ — $\gamma(n - kb)b^2/s$
6: **end for**
7: **for all** $i = k+1, \ldots, n/b$ **do**
8: broacast L_{ik} over row $i\%s$
9: broacast L_{ik} over col $i\%s$ — $\log(s)(\alpha + \beta(n - kb)b/s)$
10: **end for**
11: **for all** $i = k+1, \ldots, n/b, j = k+1, \ldots, i$ **do**
12: compute $A_{ij} -= L_{ik} L_{jk}^T$ — $\gamma(n - kb)^2 b/p$
13: **end for**
14: **end for**

Parallel Cholesky factorization

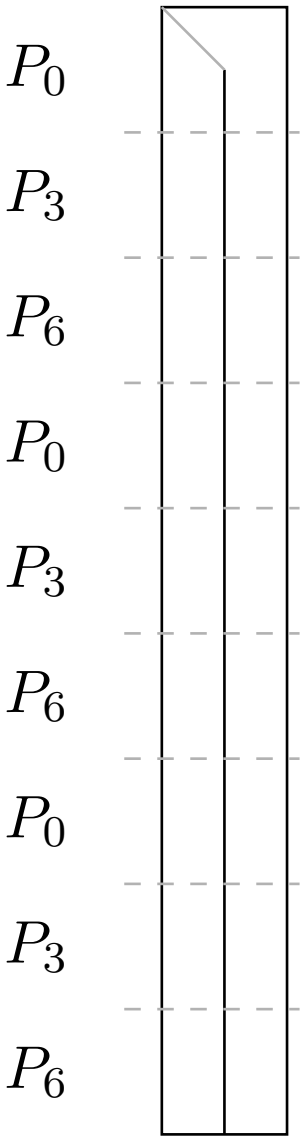
The cost of the distributed memory parallel Cholesky is

$$
\begin{aligned}
T_{chol}(n,p) \quad = \quad \sum_{k=1}^{n/b} \quad & \gamma b^3/3 + \\
& \log(s)(\alpha + \beta b^2/2) + \\
& \gamma(n-kb)b^2/s + \\
& \log(s)(\alpha + \beta(n-kb)b/s) + \\
& \gamma(n-kb)^2 b/p
\end{aligned}
$$

$$
T_{chol}(n,p) \simeq \frac{n^3}{3p}\gamma + O\left(\frac{n}{b}\log(s)\alpha\right) + O\left(n^2\frac{\log(s)}{s}\beta\right) = \frac{T(n,1)}{p} + \frac{O(n,p)}{p}
$$

Parallel LU factorization

What happens at step i in the panel reduction assuming the panel is of size $n \times b$ (e.g., first panel):

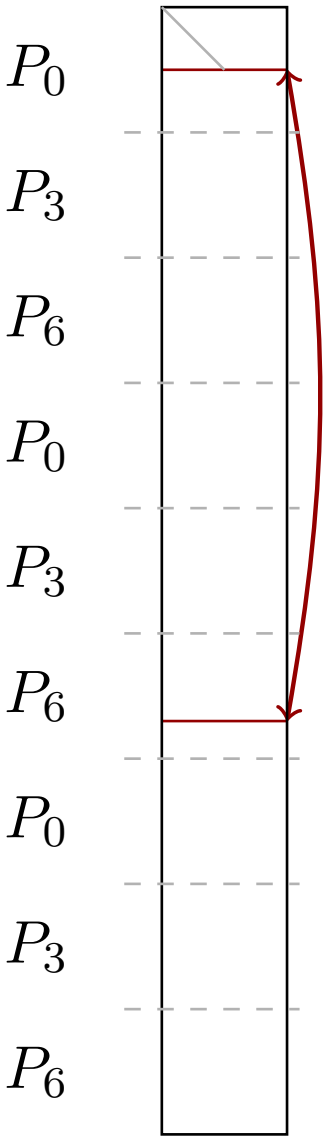


1. allreduce to find cmax pivot in column i

$$2\log(s)(\alpha + \beta)$$

Parallel LU factorization

What happens at step i in the panel reduction assuming the panel is of size $n \times b$ (e.g., first panel):



1. allreduce to find cmax pivot in column i

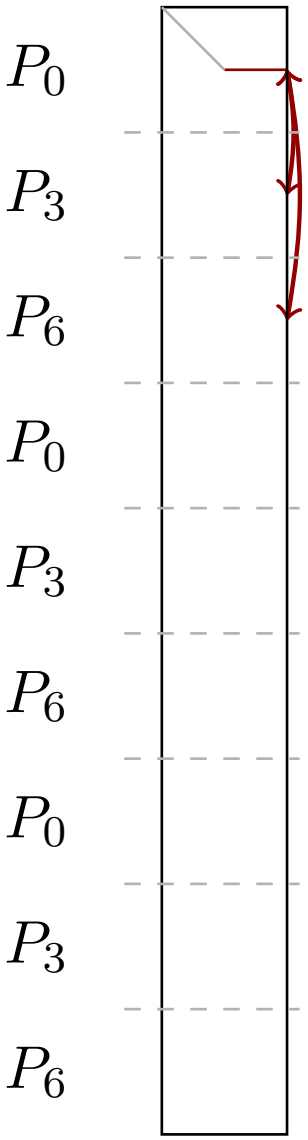
$$2\log(s)(\alpha + \beta)$$

2. permute row i and row of found pivot

$$2(\alpha + \beta b)$$

Parallel LU factorization

What happens at step i in the panel reduction assuming the panel is of size $n \times b$ (e.g., first panel):



1. allreduce to find cmax pivot in column i

$$2\log(s)(\alpha + \beta)$$

2. permute row i and row of found pivot

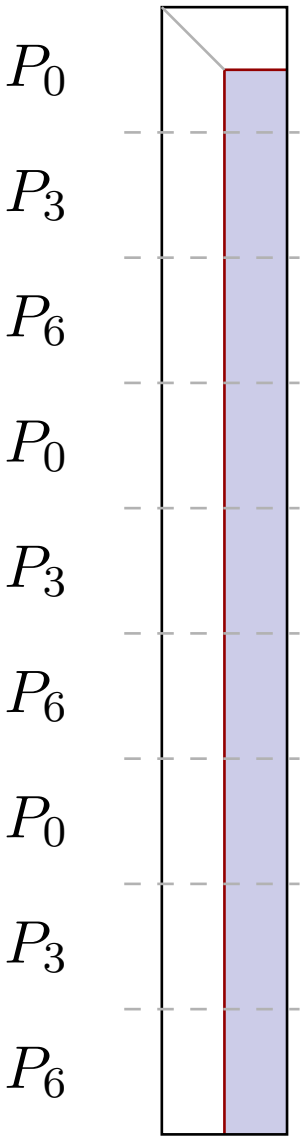
$$2(\alpha + \beta b)$$

3. broadcast row i along grid column

$$\log(s)(\alpha + (b - i)\beta)$$

Parallel LU factorization

What happens at step i in the panel reduction assuming the panel is of size $n \times b$ (e.g., first panel):



1. allreduce to find cmax pivot in column i

$$2\log(s)(\alpha + \beta)$$

2. permute row i and row of found pivot

$$2(\alpha + \beta b)$$

3. broadcast row i along grid column

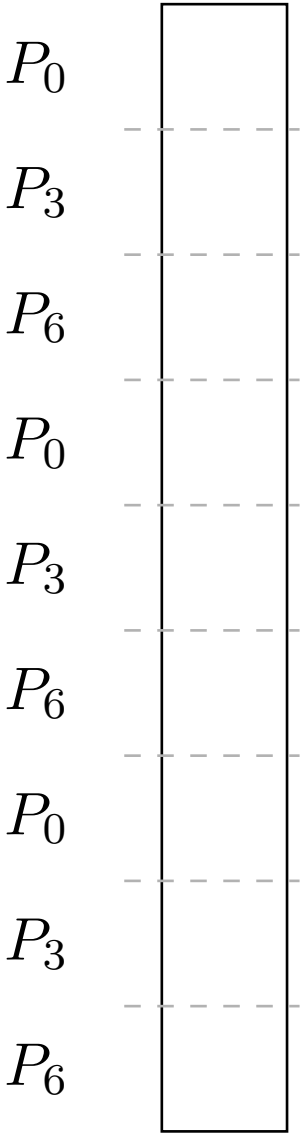
$$\log(s)(\alpha + (b - i)\beta)$$

4. scale column i with pivot and local trailing subpanel update

$$\gamma \frac{n - i}{s} (1 + 2(b - i))$$

Parallel LU factorization

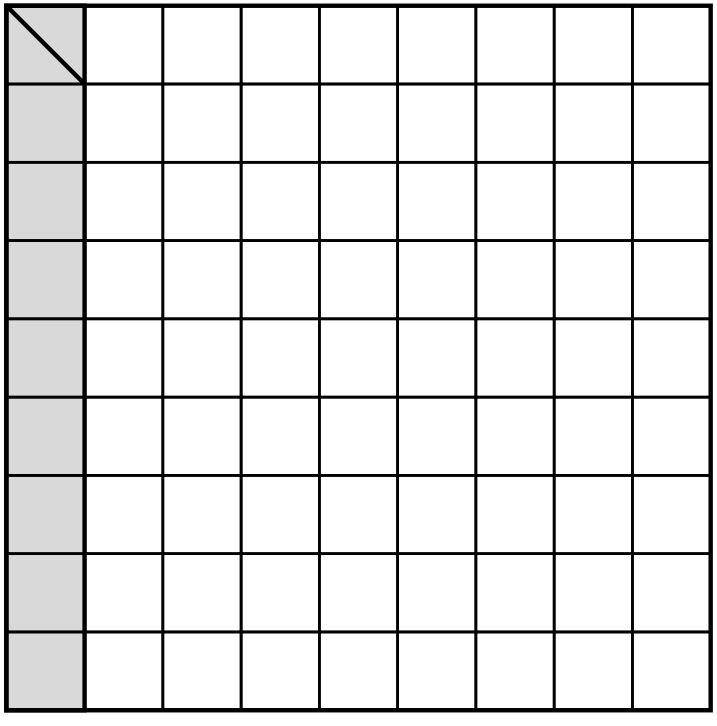
What happens at step i in the panel reduction assuming the panel is of size $n \times b$ (e.g., first panel):



$$
\begin{aligned}
T_{pnl}(n, b, s) &= \sum_{i=1}^{b} (\dots) \\
&\simeq O\left(\frac{m}{s} b^2 \gamma\right) + \\
&\quad O\left(b \log(s) \alpha\right) \\
&\quad O\left(\frac{b^2}{2} \log(s) \beta\right)
\end{aligned}
$$

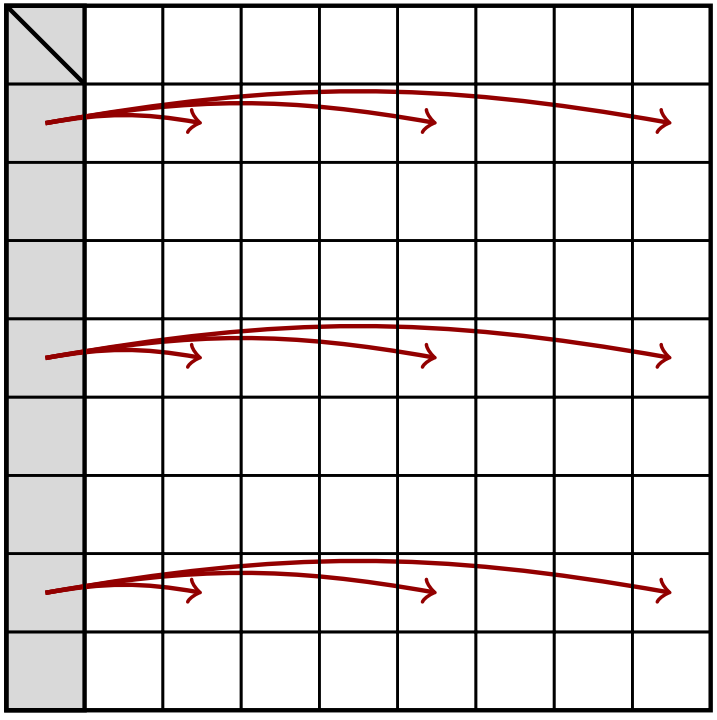
Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$



Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$

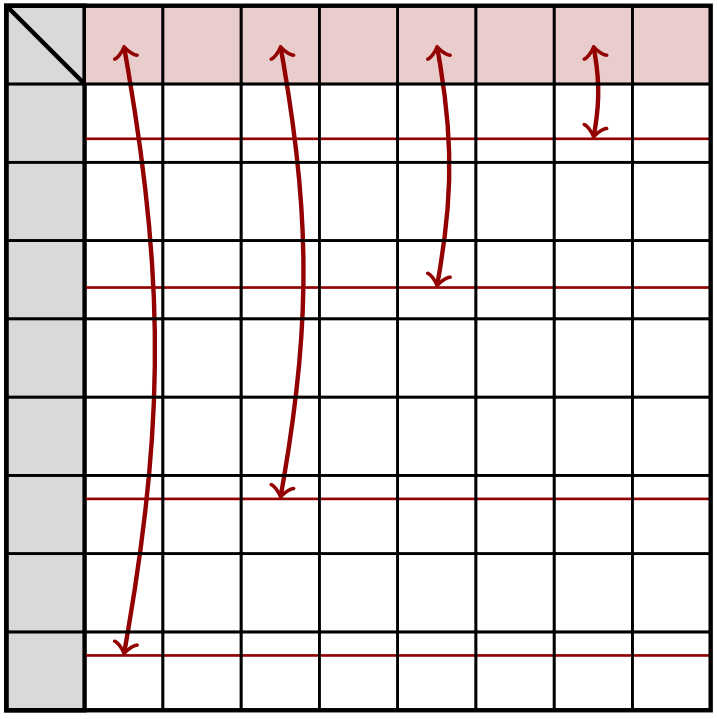


1. Bcast permutation along grid rows and permute rows

$$\log s(\alpha + \beta b) + b(\alpha + \beta n/s)$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$

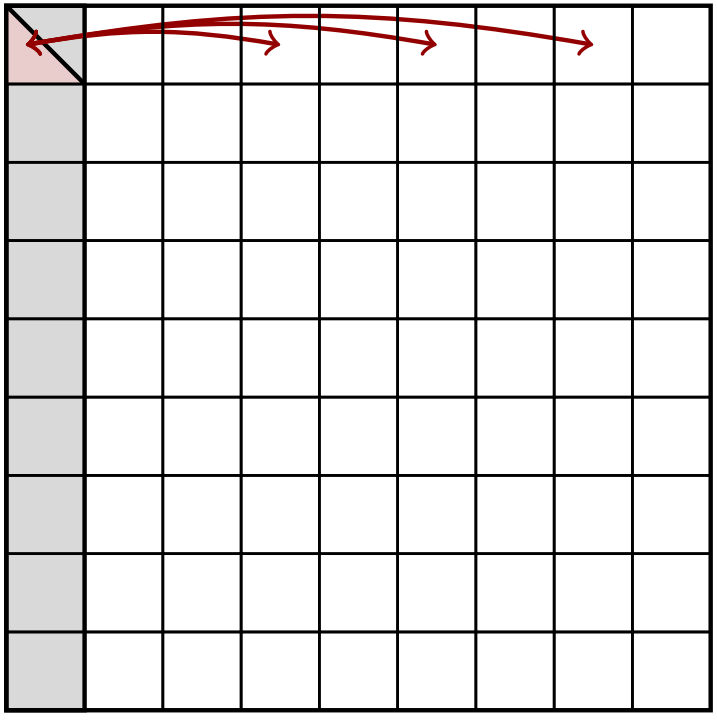


1. Bcast permutation along grid rows and permute rows

$$\log s(\alpha + \beta b) + b(\alpha + \beta n/s)$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$



1. Bcast permutation along grid rows and permute rows

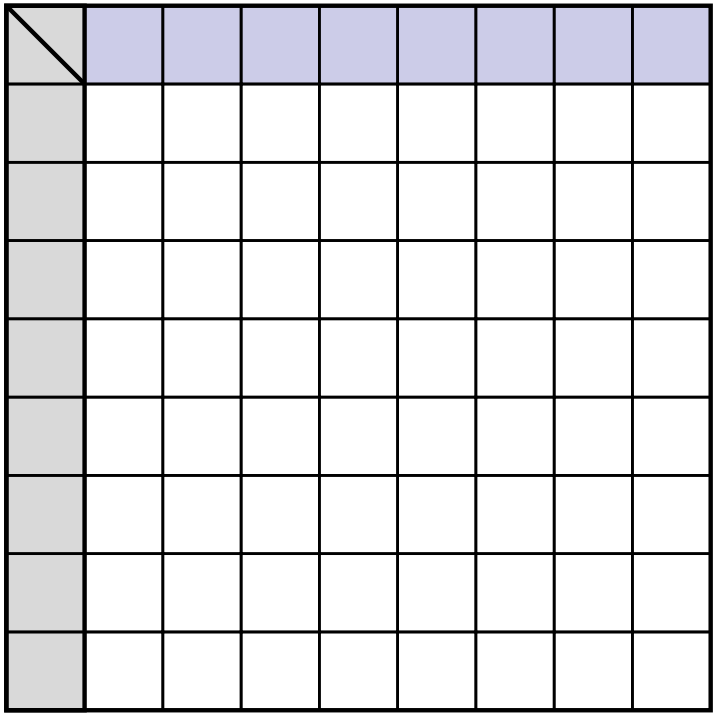
$$\log s(\alpha + \beta b) + b(\alpha + \beta n/s)$$

2. Bcast L_{kk} along grid row and compute $U_{kj} = L_{kk}^{-1} A_{kj}$

$$\log s(\alpha + \beta b^2/2) + \gamma b^2(n - b)/s$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$



1. Bcast permutation along grid rows and permute rows

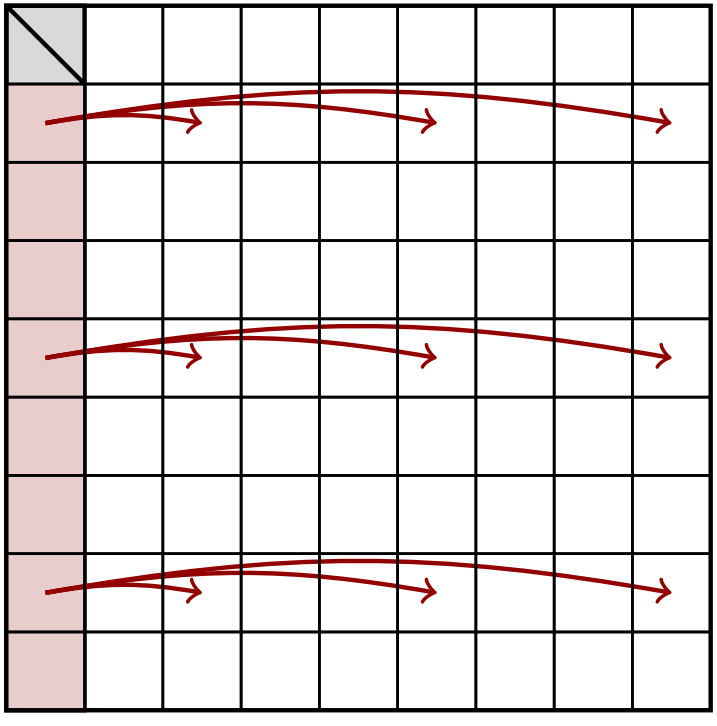
$$\log s(\alpha + \beta b) + b(\alpha + \beta n/s)$$

2. Bcast L_{kk} along grid row and compute $U_{kj} = L_{kk}^{-1} A_{kj}$

$$\log s(\alpha + \beta b^2/2) + \gamma b^2(n - b)/s$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$

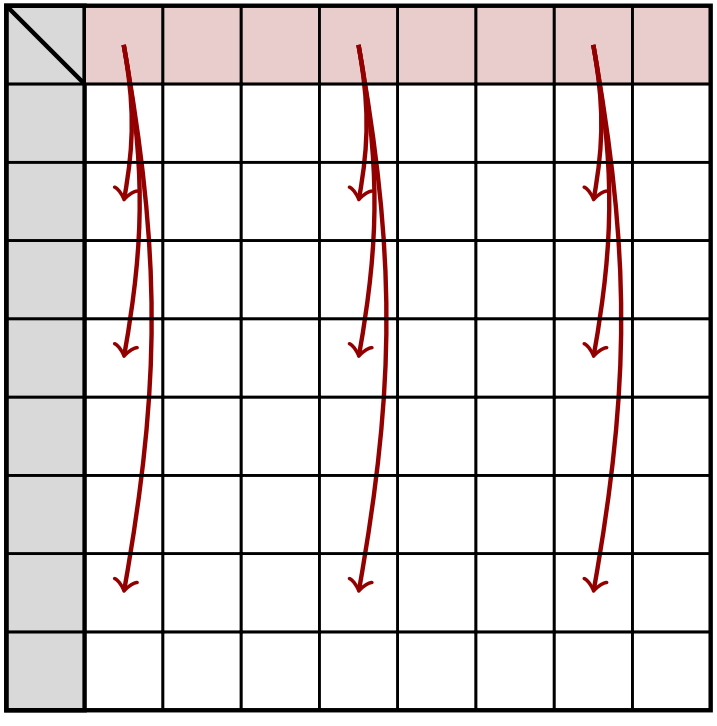


3. Bcast L_{ik} along grid rows and U_{kj} along grid cols

$$2 \log s(\alpha + \beta b(n - b)/s)$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$

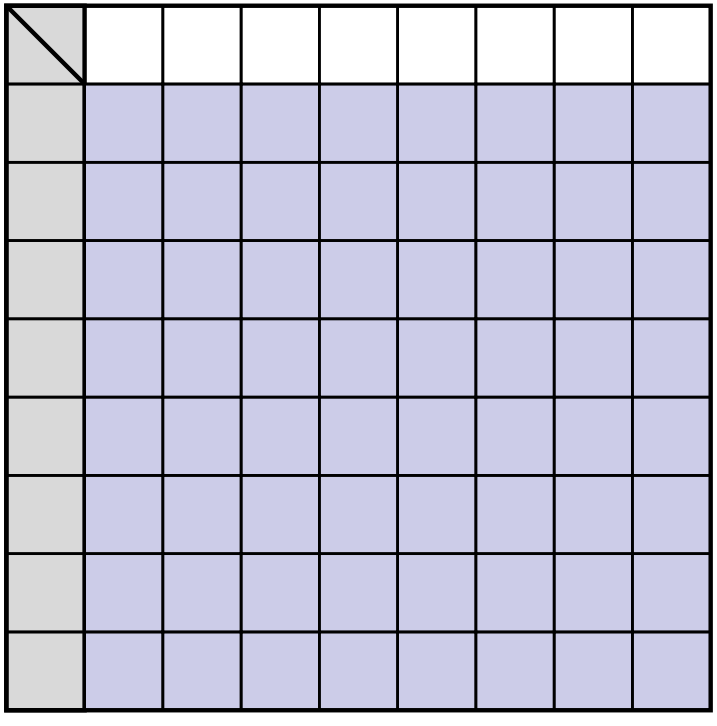


3. Bcast L_{ik} along grid rows and U_{kj} along grid cols

$$2 \log s(\alpha + \beta b(n - b)/s)$$

Parallel LU factorization

What happens at step i in the trailing submatrix update assuming it is of size $n \times (n - b)$



3. Bcast L_{ik} along grid rows and U_{kj} along grid cols

$$2 \log s(\alpha + \beta b(n - b)/s)$$

4. compute $A_{ij} - = L_{ik} U_{kj}$

$$2\gamma b((n - b)/s)^2$$

Parallel LU factorization

Algorithm 11 Parallel LU

1: **for** $k = 1, \ldots, n/b$ **do**
2: Panel $P_k A_{k,k:} = L_{k,k:} U_{kk}$
3: **for all** $j = k+1, \ldots, n/b$ **do**
4: Bcast P_k over grid rows $j\%s$
5: Apply P_k to col j
6: **end for**
7: Bcast L_k along grid row $k\%s$
8: **for all** $j = k+1, \ldots, n/b$ **do**
9: compute $U_{ik} = L_{kk}^{-1} A_{ik}$
10: **end for**
11: **for all** $i, j = k+1, \ldots, n/b$ **do**
12: broacast L_{ik} over row $i\%s$ and U_{kj} over col $j\%s$
13: **end for**
14: **for all** $i, j = k+1, \ldots, n/b$ **do**
15: compute $A_{ij} -= L_{ik} U_{kj}$
16: **end for**
17: **end for**

Parallel LU factorization

Algorithm 11 Parallel LU

1: **for** $k = 1, \ldots, n/b$ **do**
2: Panel $P_k A_{k,k:} = L_{k,k:} U_{kk}$ — $T_{pnl}(n - kb, b, s)$
3: **for all** $j = k + 1, \ldots, n/b$ **do**
4: Bcast P_k over grid rows $j\%s$ — $\log(s)(\alpha + \beta b)$
5: Apply P_k to col j — $b(\alpha + \beta n/s)$
6: **end for**
7: Bcast L_k along grid row $k\%s$ — $\log(s)(\alpha + \beta b^2/2)$
8: **for all** $j = k + 1, \ldots, n/b$ **do**
9: compute $U_{ik} = L_{kk}^{-1} A_{ik}$ — $\gamma b^2 (n - kb)/s$
10: **end for**
11: **for all** $i, j = k + 1, \ldots, n/b$ **do**
12: broacast L_{ik} over row $i\%s$ and U_{kj} over col $j\%s$ — $2 \log s(\alpha + \beta b(n - kb)/s)$
13: **end for**
14: **for all** $i, j = k + 1, \ldots, n/b$ **do**
15: compute $A_{ij} -= L_{ik} U_{kj}$ — $2\gamma b((n - kb)/s))^2$
16: **end for**
17: **end for**

Parallel LU factorization

The cost of the distributed memory parallel LU is

$$
\begin{aligned}
T_{LU}(n,p) \quad = \quad \sum_{k=1}^{n/b} \quad & T_{pnl}(n-kb,b,s)+ \\
& \log(s)(\alpha+\beta b)+ \\
& b(\alpha+\beta n/s)+ \\
& \log(s)(\alpha+\beta b^2/2)+ \\
& \gamma b^2(n-kb)/s+ \\
& 2\log s(\alpha+\beta b(n-kb)/s)+ \\
& 2\gamma b((n-kb)/s))^2
\end{aligned}
$$

$$
T_{LU}(n,p) \simeq \frac{2n^3}{3p}\gamma + O\left(n\log(s)\alpha\right) + O\left(n^2\frac{\log(s)}{s}\beta\right) = \frac{T(n,1)}{p} + \frac{O(n,p)}{p}
$$

The LU factorization involves $O(b)$ times more messages because of partial pivoting.

Parallel LU factorization

The LU factorization involves $O(b)$ times more messages because of partial pivoting. Alternative but not as stable approaches are

- static pivoting: replace small pivots by a small value, e.g., $\tau\|A\|$ with τ of the order of the unit roundoff. Same communications as in Cholesky but often unstable/inaccurate in practice.

- block-pairwise pivoting: works the same as the tiled QR factorization with partial pivoting within pairs of blocks. Not as practically stable as partial pivoting.

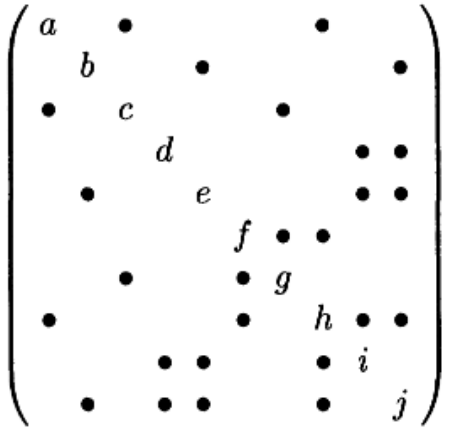
- tournament pivoting: search for pivots in the panel in a recursive fashion. Fewer number of messages than partial pivoting but more operations and theoretically not as stable.

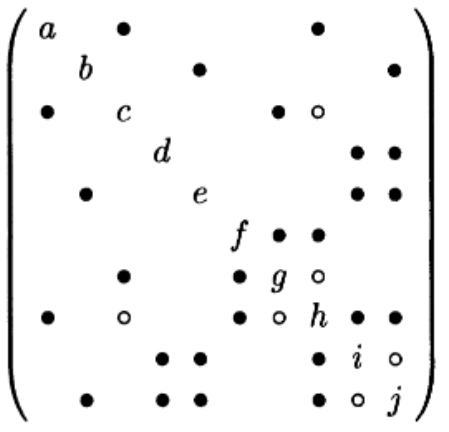
Factorization of sparse matrices: fill-in

The Cholesky update1 step (assuming $a_{ij}^0 = a_{ij}$ and $l_{ij} = a_{ij}^{(j)}$):

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)} a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}}$$

Even if $a_{ij}^{(k-1)}$ is null, $a_{ij}^{(k)}$ can be a nonzero

$A =$ 

$F =$ 

- the factorization is more expensive than $\mathcal{O}(nz)$
- higher amount of memory required than $\mathcal{O}(nz)$
- more complicated algorithms to achieve the factorization

Modeling fill-in: adjacency graphs

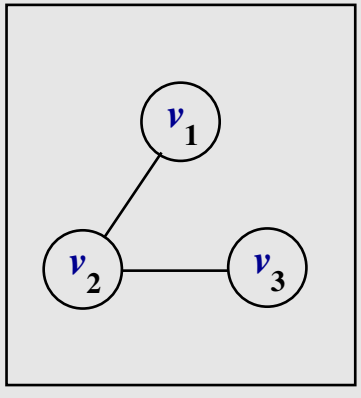
Symmetric pattern matrix: undirected graph

A graph $G = (V, E)$ consists of a finite set V, called the vertex set and a finite, binary relation E on V, called the edge set.
In an Undirected graph the edges are unordered pair of vertices, i.e., $\{u, v\} \in E$ for some $u, v \in V$.

The rows/columns and nonzeros of a given sparse matrix correspond (with natural labelling) to the vertices and edges, respectively, of a graph.

Square symmetric pattern matrices

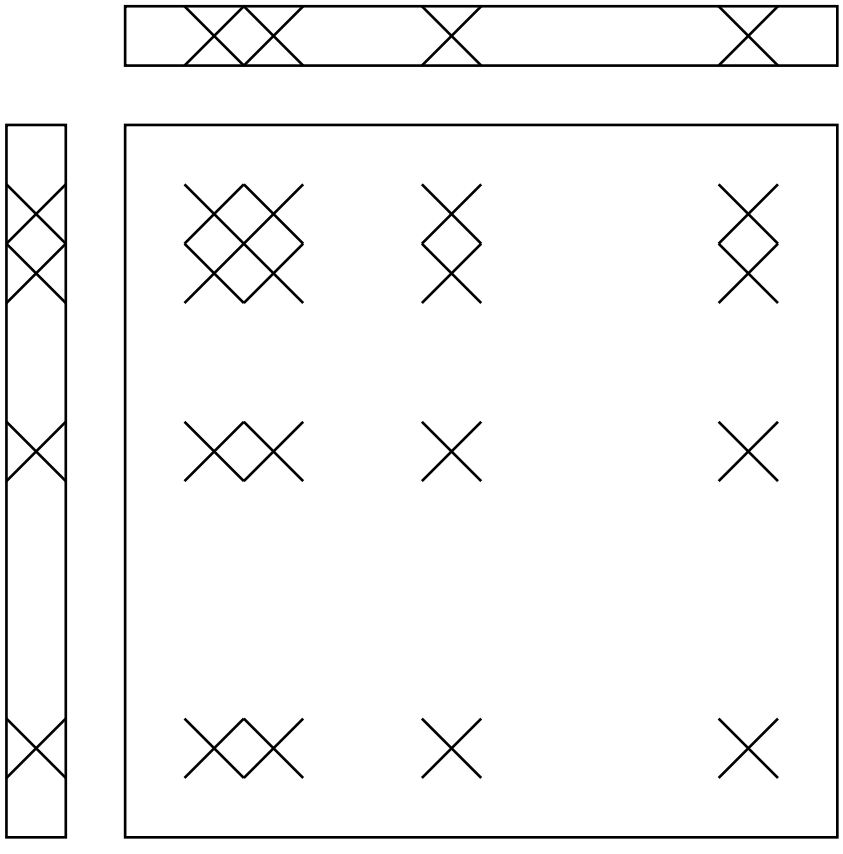
$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{pmatrix} & \times & \\ \times & \times & \times \\ & \times & \times \end{pmatrix} \end{array}$$



Modeling fill-in

Remember the trailing submatrix update $A_1 = \overline{A_1} - \frac{a_{:1}a_{:1}^T}{a_{11}}$.

What is $a_{:1}a_{:1}^T$ in terms of structure?



$a_{:1}$ is the first column of A, thus it contains the neighbors of 1 in the adjacency graph of A.

$a_{:1}a_{:1}^T$ results in a dense sub-block in A_1, i.e., the elimination of a node results in the creation of a clique that connects all the neighbors of the eliminated node.

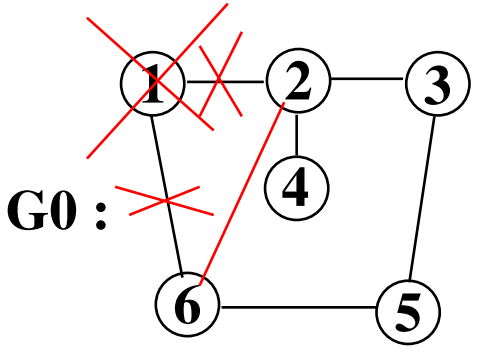
If any of the nonzeros in dense submatrix are not in A, then we have fill-ins.

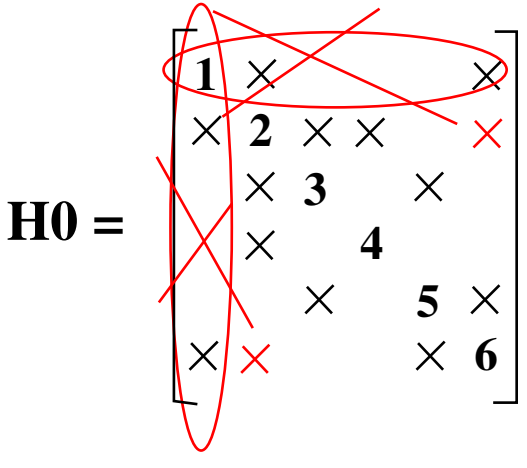
The elimination process in the graphs

$G_U(V, E) \leftarrow$ undirected graph of A
for $k = 1 : n - 1$ **do**
    $V \leftarrow V - \{k\}$ ▷ remove vertex k

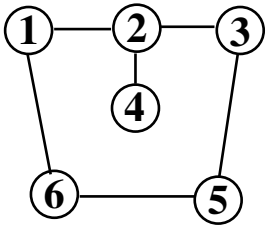
$E \leftarrow E - \{(k, \ell) : \ell \in \text{adj}(k)\} \cup \{(x, y) : x \in \text{adj}(k) \text{ and } y \in \text{adj}(k)\}$
    $G_k \leftarrow (V, E)$ ▷ for definition
end for

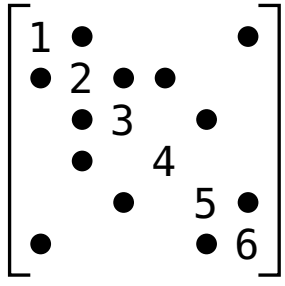
G_k are the so-called elimination graphs (Parter [0]).

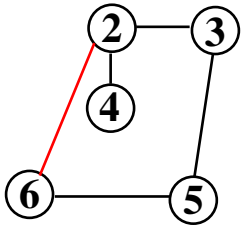


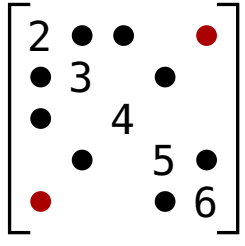


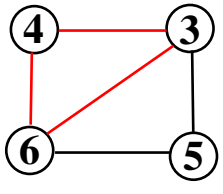
A sequence of elimination graphs

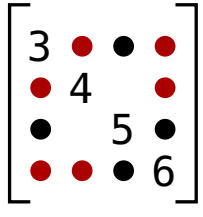
G_0: 

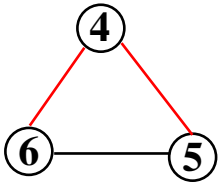
$A_0 =$ 

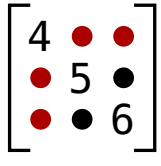
G_1: 

$A_1 =$ 

G_2: 

$A_2 =$ 

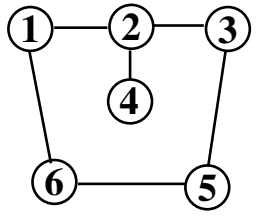
G_3: 

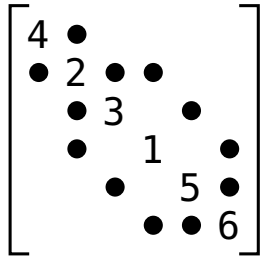
$A_3 =$ 

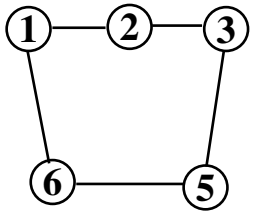
$$\begin{bmatrix} l_{11} & & & & & \\ l_{21} & l_{22} & & & & \\ & l_{32} & l_{33} & & & \\ & l_{42} & l_{43} & l_{44} & & \\ & & l_{53} & l_{54} & l_{55} & \\ l_{61} & l_{62} & l_{63} & l_{64} & l_{65} & l_{66} \end{bmatrix}$$

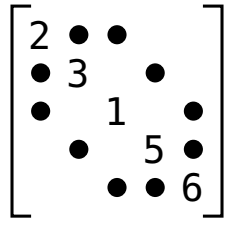
Fill-in and matrix permutations

The nodes don't have to be necessarily eliminated in the natural order:

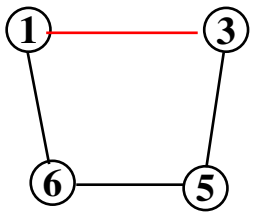
G_0: 

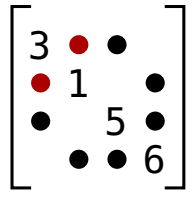
$A_0 =$ 

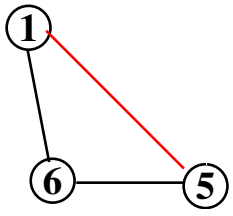
G_4: 

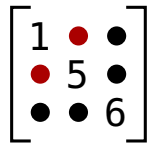
$A_4 =$ 

$$\begin{bmatrix} l_{44} & & & & & \\ l_{24} & l_{22} & & & & \\ & l_{32} & l_{33} & & & \\ & l_{12} & l_{13} & l_{11} & & \\ & & l_{53} & l_{51} & l_{55} & \\ & & & l_{61} & l_{65} & l_{66} \end{bmatrix}$$

G_2: 

$A_2 =$ 

G_3: 

$A_3 =$ 

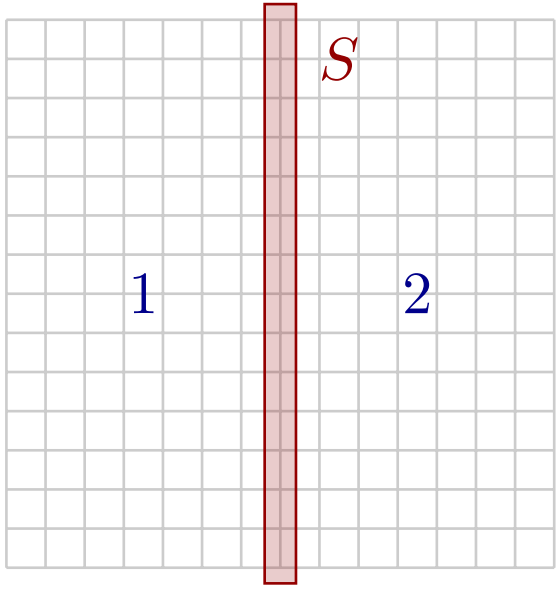
Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.

$[A]$

Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.



$$\begin{bmatrix} A_{11} & & A_{1S} \\ & A_{22} & A_{2S} \\ A_{S1} & A_{S2} & A_{SS} \end{bmatrix}$$

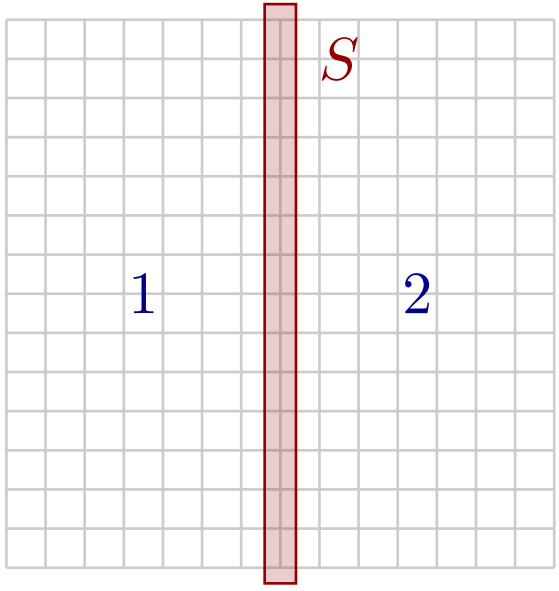
$$A_{11} \rightarrow L_{11}L_{11}^T$$
$$L_{21} = A_{21}L_{11}^{-T}$$
$$L_{S1} = A_{S1}L_{11}^{-T}$$
$$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T$$
$$\tilde{A}_{S2} = A_{S2} - L_{S1}L_{21}^T$$
$$\tilde{A}_{SS} = A_{SS} - L_{S1}L_{S1}^T$$

$$\tilde{A}_{22} \rightarrow L_{22}L_{22}^T$$
$$L_{S2} = \tilde{A}_{S2}L_{22}^{-T}$$
$$\hat{A}_{SS} = \tilde{A}_{SS} - L_{S2}L_{S2}^T$$

$$\hat{A}_{SS} \rightarrow L_{SS}L_{SS}^T$$

Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.



$$\begin{bmatrix} A_{11} & & A_{1S} \\ & A_{22} & A_{2S} \\ A_{S1} & A_{S2} & A_{SS} \end{bmatrix}$$

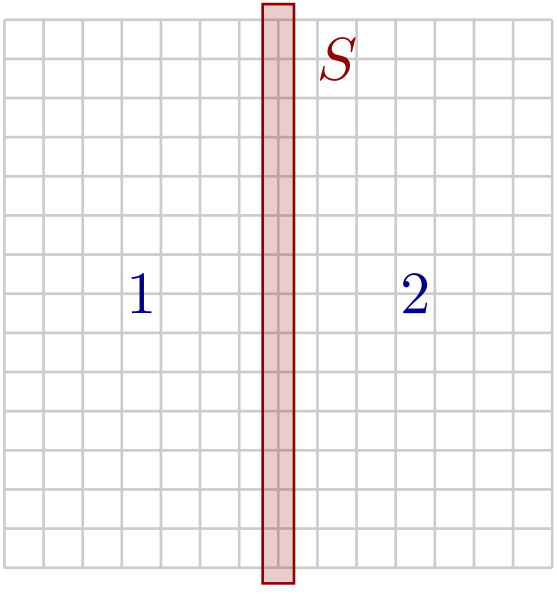
$A_{11} \rightarrow L_{11}L_{11}^T$
$L_{21} = A_{21}L_{11}^{-T}$
$L_{S1} = A_{S1}L_{11}^{-T}$
$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T$
$\tilde{A}_{S2} = A_{S2} - L_{S1}L_{21}^T$
$\tilde{A}_{SS} = A_{SS} - L_{S1}L_{S1}^T$

$A_{22} \rightarrow L_{22}L_{22}^T$
$L_{S2} = A_{S2}L_{22}^{-T}$
$\hat{A}_{SS} = \tilde{A}_{SS} - L_{S2}L_{S2}^T$

$\hat{A}_{SS} \rightarrow L_{SS}L_{SS}^T$

Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.



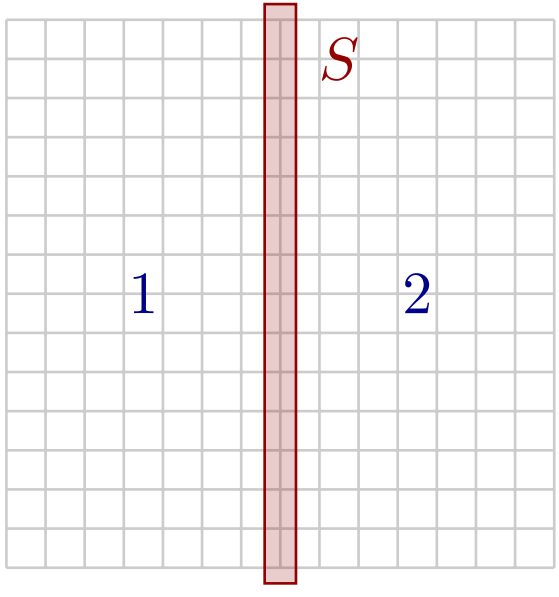
$$\begin{bmatrix} A_{11} & & A_{1S} \\ & A_{22} & A_{2S} \\ A_{S1} & A_{S2} & A_{SS} \end{bmatrix}$$

$$A_{11} \rightarrow L_{11}L_{11}^T \qquad A_{22} \rightarrow L_{22}L_{22}^T$$
$$L_{S1} = A_{S1}L_{11}^{-T} \qquad L_{S2} = A_{S2}L_{22}^{-T}$$
$$\tilde{A}_{SS} = A_{SS} - L_{S1}L_{S1}^T \qquad \hat{A}_{SS} = \tilde{A}_{SS} - L_{S2}L_{S2}^T$$

$$\hat{A}_{SS} \rightarrow L_{SS}L_{SS}^T$$

Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.



$$\begin{bmatrix} A_{11} & & A_{1S} \\ & A_{22} & A_{2S} \\ A_{S1} & A_{S2} & A_{SS} \end{bmatrix}$$

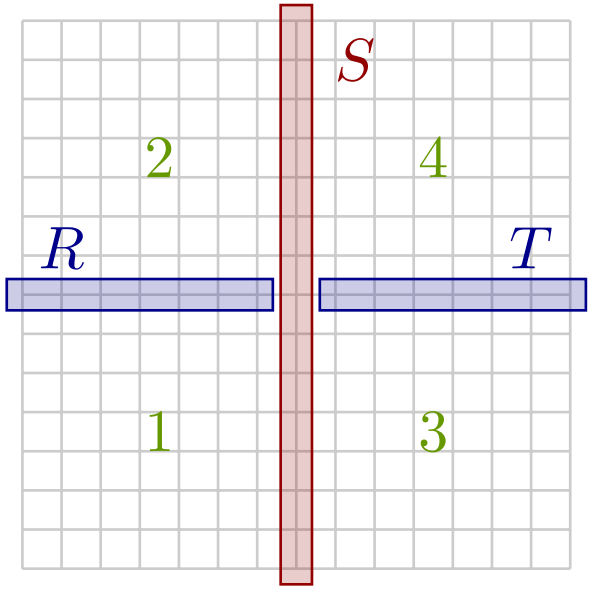
$$A_{11} \rightarrow L_{11}L_{11}^T$$
$$L_{S1} = A_{S1}L_{11}^{-T}$$
$$B_1 = L_{S1}L_{S1}^T$$

$$A_{22} \rightarrow L_{22}L_{22}^T$$
$$L_{S2} = A_{S2}L_{22}^{-T}$$
$$B_2 = L_{S2}L_{S2}^T$$

$$\hat{A}_{SS} = A_{SS} - B_1 - B_2$$
$$\hat{A}_{SS} \rightarrow L_{SS}L_{SS}^T$$

Nested dissection

The nested dissection method was proposed by George [8].
Assume a square adjacency graph with $n = N \times N$ nodes.

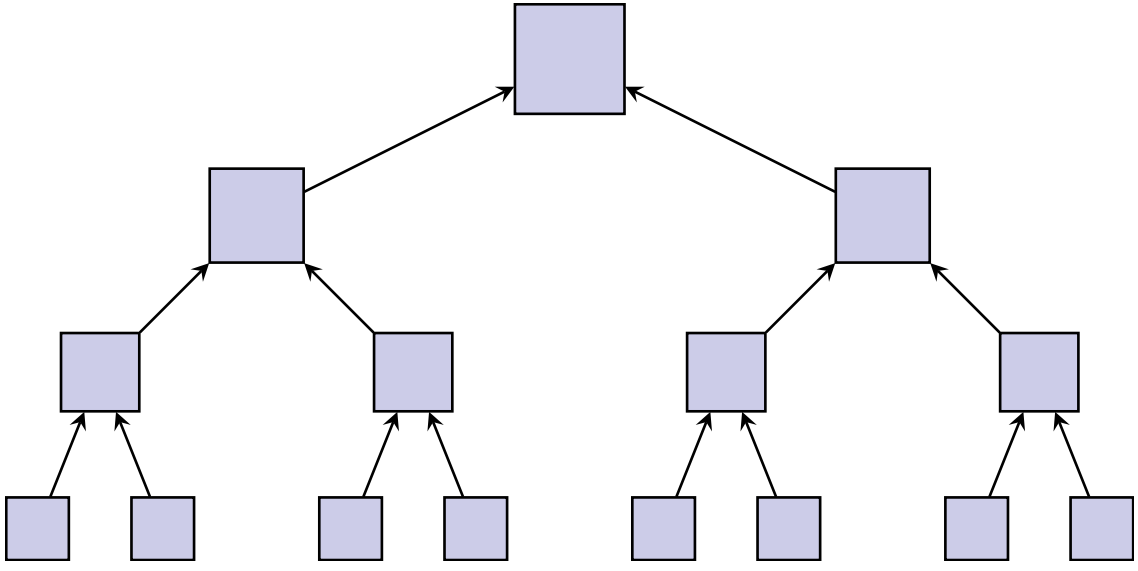


$$\begin{bmatrix}
A_{11} & & A_{1R} & & & & A_{1S} \\
& A_{22} & A_{2R} & & & & A_{2S} \\
A_{R1} & A_{R2} & A_{RR} & & & & A_{RS} \\
& & & A_{33} & & A_{3T} & A_{1S} \\
& & & & A_{44} & A_{4T} & A_{2S} \\
& & & A_{T3} & A_{T4} & A_{TT} & A_{TS} \\
A_{S1} & A_{S2} & & A_{S3} & A_{S4} & A_{ST} & A_{SS}
\end{bmatrix}$$

This procedure can be iterated recursively until small enough subdomains are found

Nested dissection

The sparse factorization with nested dissection can be described through a dependency graph called elimination tree



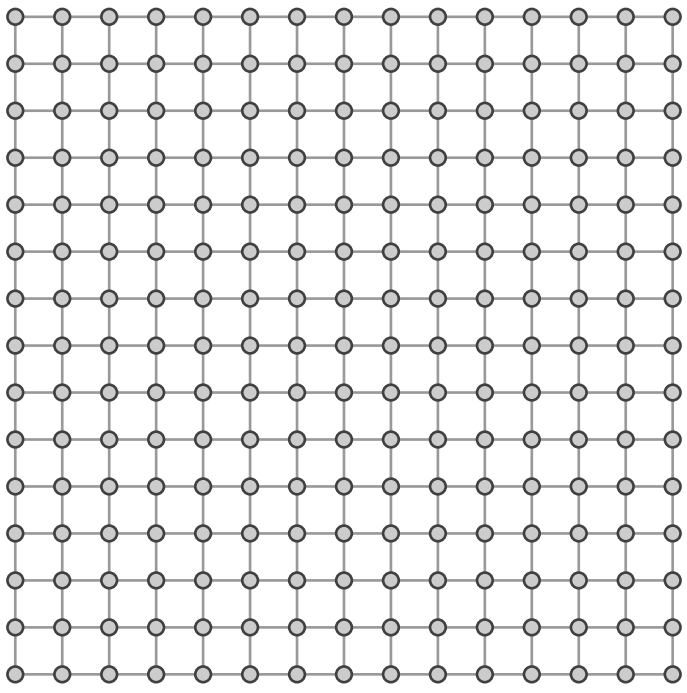
This tree (not necessarily binary because it depends on the spahe of the graph/separators) is traversed in a bottom-up fashion. At every node we work on dense submatrices of size $O(m)$ where m is the size of the corresponding separator and, therefore,

- we use $\mathcal{M}(m) = O(m^2)$ memory
- perform $\mathcal{F}(m) = O(m^3)$ operations

Complexity of the factorization with ND

Definition (2D ND assumptions (George [8]))

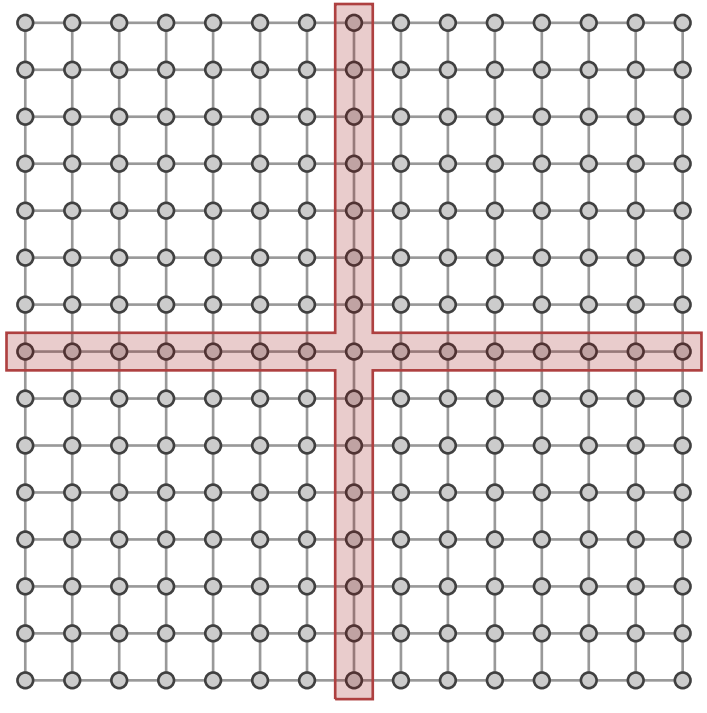
- 2D, square grid of size $N \times N$ and cross-shaped separators.
- The size of the separators/fronts is divided by 2 at every level starting at $2N$
- The number of nodes is multiplied by 4 at every level

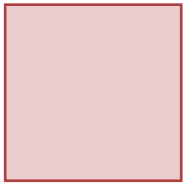


Complexity of the factorization with ND

Definition (2D ND assumptions (George [8]))

- 2D, square grid of size $N \times N$ and cross-shaped separators.
- The size of the separators/fronts is divided by 2 at every level starting at $2N$
- The number of nodes is multiplied by 4 at every level

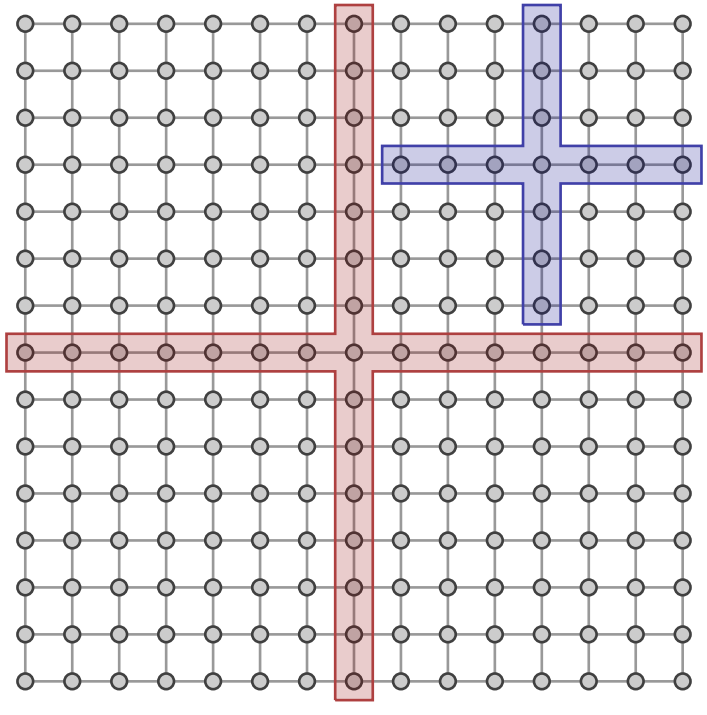


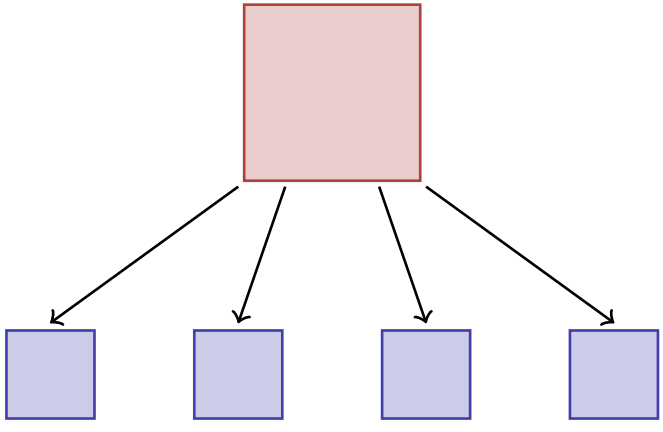


Complexity of the factorization with ND

Definition (2D ND assumptions (George [8]))

- 2D, square grid of size $N \times N$ and cross-shaped separators.
- The size of the separators/fronts is divided by 2 at every level starting at $2N$
- The number of nodes is multiplied by 4 at every level

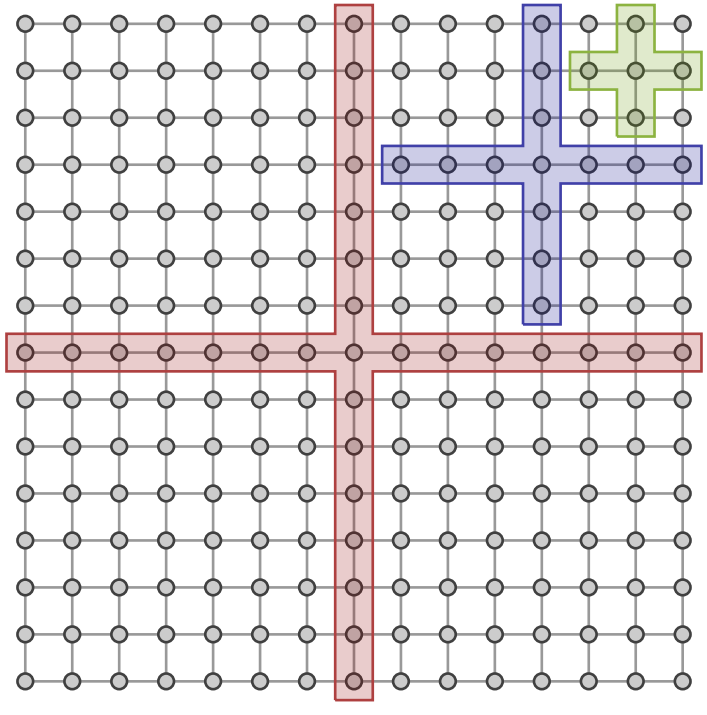


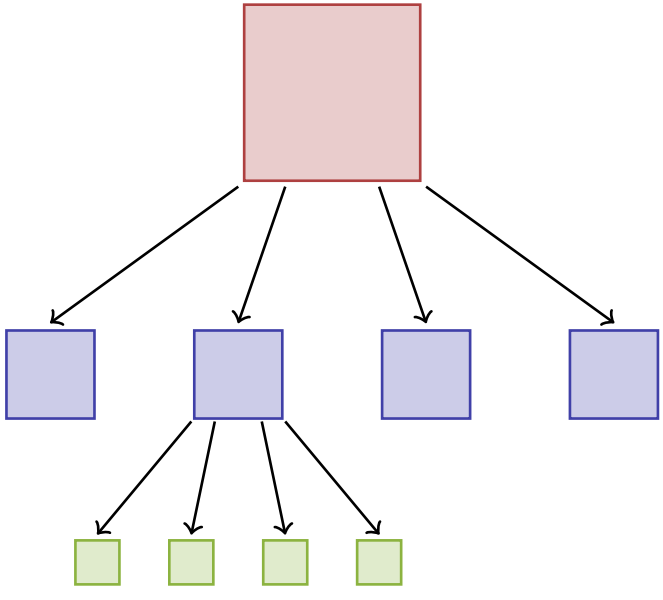


Complexity of the factorization with ND

Definition (2D ND assumptions (George [8]))

- 2D, square grid of size $N \times N$ and cross-shaped separators.
- The size of the separators/fronts is divided by 2 at every level starting at $2N$
- The number of nodes is multiplied by 4 at every level





Complexity of the factorization with ND

Flops

The factorization cost for a front of order m is $\mathcal{F}(m) = O(m^3)$

$$\mathcal{F}_{sp} = \sum_{l=0}^{log_2 N} 4^l \mathcal{F}\left(\frac{2N}{2^l}\right) = O\left(\sum_{l=0}^{log_2 N} 4^l \left(\frac{N}{2^l}\right)^3\right) = O(N^3)$$

Factors size

The size of factors at a front of order m is $\mathcal{M}(m) = O(m^2)$

$$\mathcal{M}_{sp} = \sum_{l=0}^{log_2 N} 4^l \mathcal{M}\left(\frac{2N}{2^l}\right) = O\left(\sum_{l=0}^{log_2 N} 4^l \left(\frac{N}{2^l}\right)^2\right) = O(N^2 log_2 N)$$

Complexity of the factorization with ND

Generic formula for a d-dimentional domain with $d = 2, 3$ and $\mathcal{C} = \mathcal{F},\ \mathcal{M}$

$$\mathcal{C}_{sp} = \sum_{l=0}^{log_2 N} 2^{ld} \mathcal{C}\left(\left(\frac{N}{2^l}\right)^{d-1}\right)$$

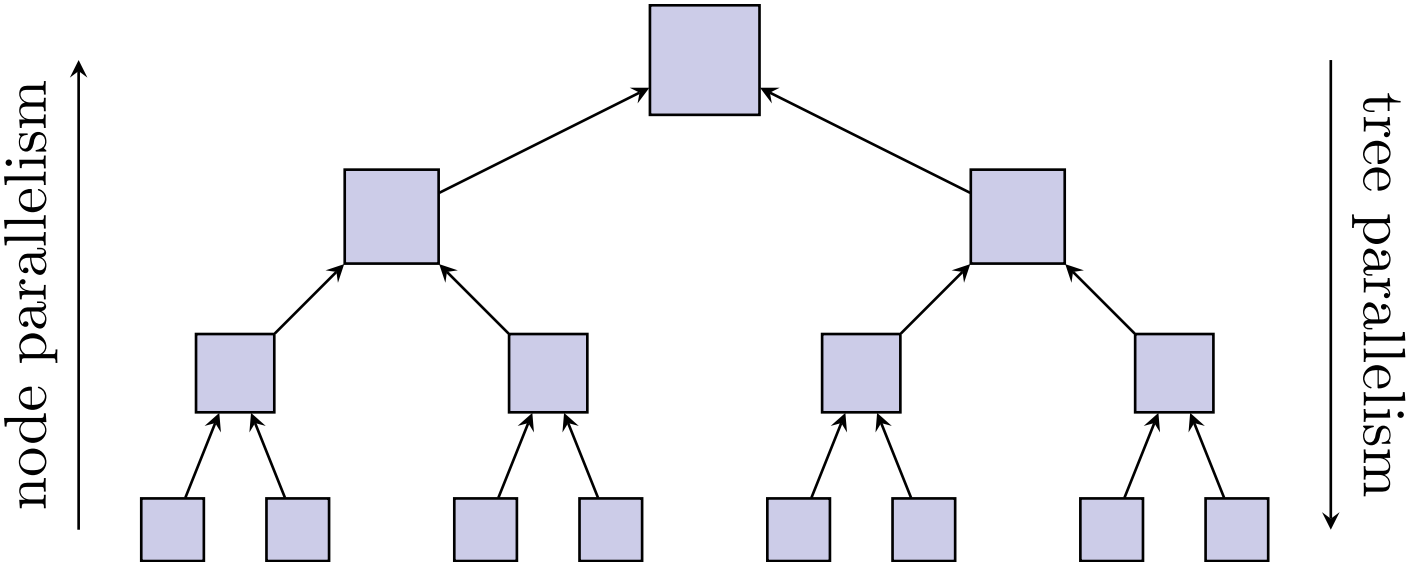
Regular problems (nested dissection)	2D $N \times N$ grid	3D $N \times N \times N$ grid
Nonzeros in original matrix	$O(N^2)$	$O(N^3)$
Nonzeros in factors	$O(N^2 logN)$	$O(N^4)$
Floating-point ops	$O(N^3)$	$O(N^6)$

The cost of the multifrontal factorization is dominated by the cost of the topmost front factorization.
In 3D this is also the case for the factors size.

Parallelization: two sources of parallelism

tree parallelism : nodes in separate subtrees of the elimination tree can be eliminated at the same time

node parallelism : within each node, parallel dense factorization



Using both sources of parallelism is crucial because they are complementary:

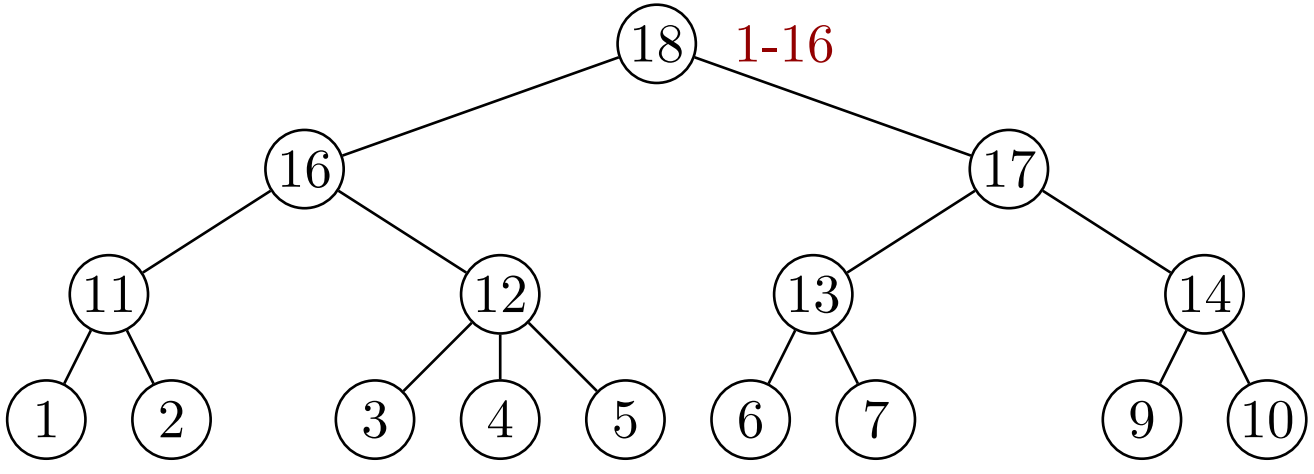
- Tree parallelism decreases going up because the tree gets more and more narrow
- Node parallelism grows going up because nodes become bigger and bigger

Proportional mapping

The Proportional Mapping method was proposed by **ps:93** and aims at computing an efficient processes-to-nodes mapping for sparse, direct solvers for distributed memory parallelism.

Proportional mapping

- initially assigns all processes to root node.
- performs a top-down traversal of the tree where the processes assigned to a node are subdivided among its children in a way that is proportional to their relative weight

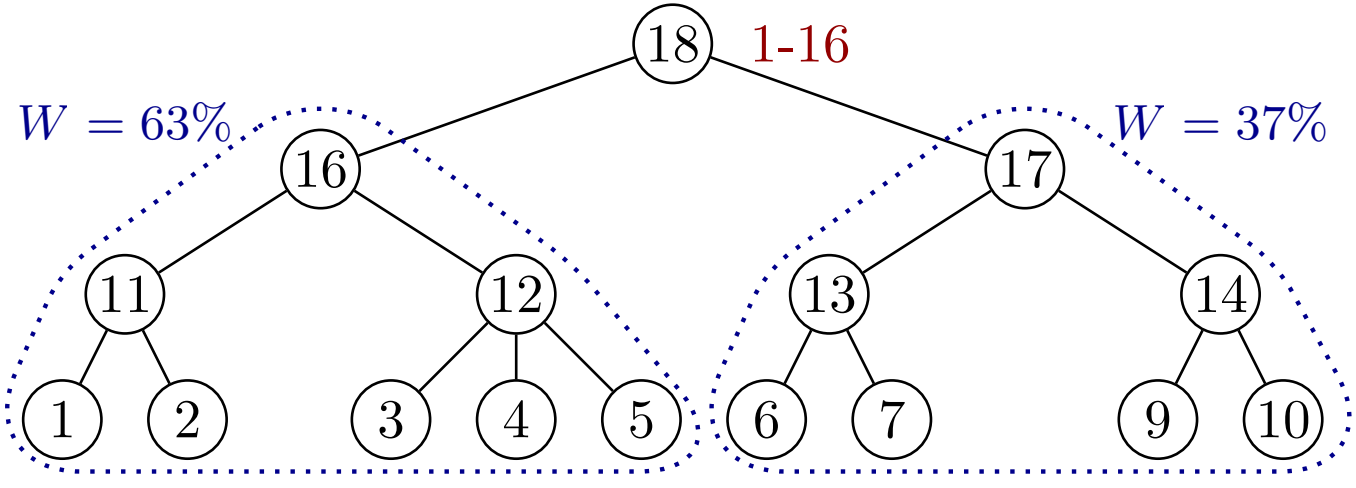


Proportional mapping

The Proportional Mapping method was proposed by **ps:93** and aims at computing an efficient processes-to-nodes mapping for sparse, direct solvers for distributed memory parallelism.

Proportional mapping

- initially assigns all processes to root node.
- performs a top-down traversal of the tree where the processes assigned to a node are subdivided among its children in a way that is proportional to their relative weight

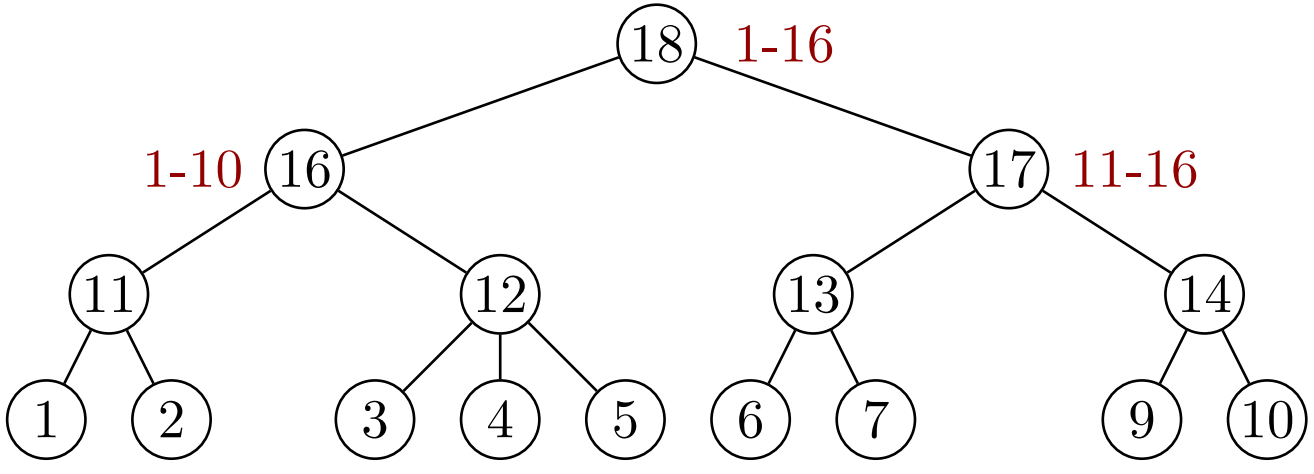


Proportional mapping

The Proportional Mapping method was proposed by **ps:93** and aims at computing an efficient processes-to-nodes mapping for sparse, direct solvers for distributed memory parallelism.

Proportional mapping

- initially assigns all processes to root node.
- performs a top-down traversal of the tree where the processes assigned to a node are subdivided among its children in a way that is proportional to their relative weight

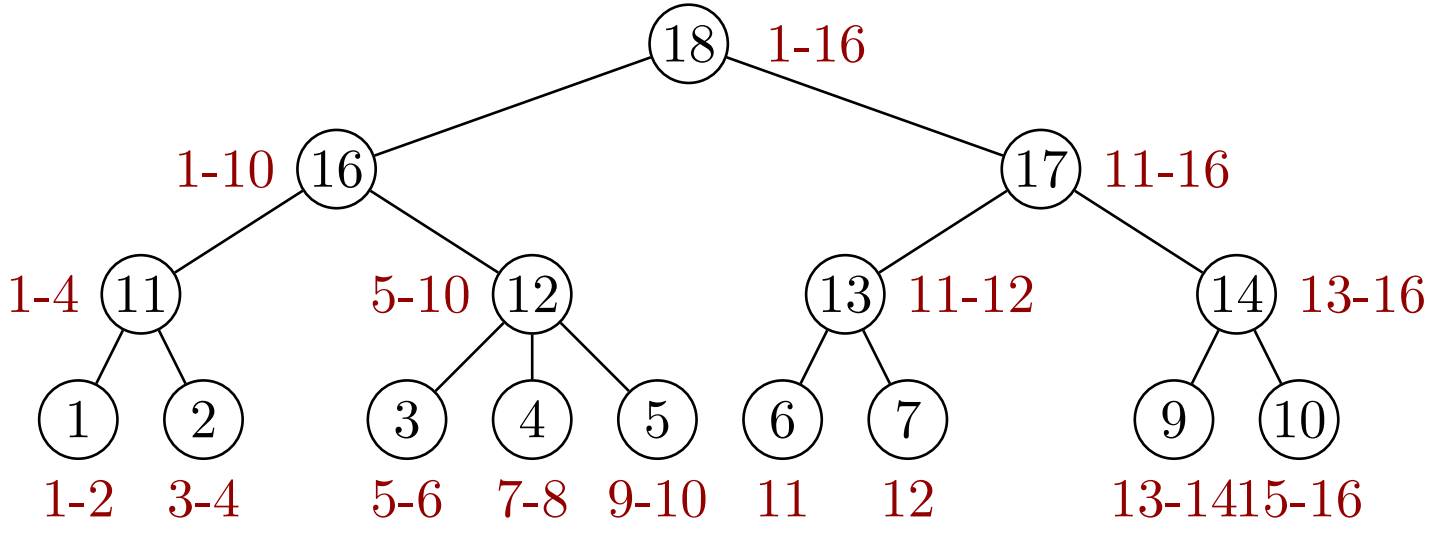


Proportional mapping

The Proportional Mapping method was proposed by **ps:93** and aims at computing an efficient processes-to-nodes mapping for sparse, direct solvers for distributed memory parallelism.

Proportional mapping

- initially assigns all processes to root node.
- performs a top-down traversal of the tree where the processes assigned to a node are subdivided among its children in a way that is proportional to their relative weight



Proportional mapping

Properties of proportional mapping:

- Compatible with node parallelism: more processes are assigned to topmost nodes that are larger
- Reduces communications in tree parallelism:
  - no branch-to-branch communications because the assigned processes are in disjoint subsets
  - processes assigned to a node are also assigned to its parent which reduces communications in assembly operations
- Aims at achieving a good balance between the branches of the tree because of the proportional distribution
- The weight of subtrees can be computed using different metrics (e.g., flops or memory or a combination of the two) which allows for balancing different properties

Sparse factorization: parallel execution time

Only tree parallelism

In this case all the branches are visited concurrently by one process. Therefore we will the execution time corresponds to the time along a single branch (because they are all equal)

$$T_{sp}(N,d,p) = \sum_{l=0}^{log_2 N} 2^{ld} T_{chol}\left(\left(\frac{N}{2^l}\right)^{d-1}, 1\right) =$$

$$(2D) \qquad O\left(N^3\right)\gamma$$

$$(3D) \qquad O\left(N^6\right)\gamma$$

- No communications (nice!)
- The time is still dominated by the time spent on the root node which is treated by a single process like all the other nodes. Therefore the execution time is not reduced wrt a sequential execution

Sparse factorization: parallel execution time

Only node parallelism

In this case all the tree nodes are visited sequentially by all p processes

$$T_{sp}(N,d,p) = \sum_{l=0}^{log_2 N} 2^{ld} T_{chol}\left(\left(\frac{N}{2^l}\right)^{d-1}, p\right) =$$

$$(2D) \qquad O\left(\frac{N^3}{p}\right)\gamma + O\left(\frac{N^2}{b}\log(s)\right)\alpha + O\left(N^2\log(N)\frac{\log(s)}{s}\right)\beta$$

$$(3D) \qquad O\left(\frac{N^6}{p}\right)\gamma + O\left(\frac{N^3}{b}\log(s)\right)\alpha + O\left(N^4\frac{\log(s)}{s}\right)\beta$$

- The terms depending on the floating point operations and the message volume scale well
- The term depending on the number of messages scales bad and is huge

Sparse factorization: parallel execution time

Node and tree parallelism

In this case all the branches are visited concurrently by multiple processes. Therefore we will the execution time corresponds to the time along a single branch (because they are all equal) and the number of processes is divided by 2^d at every level

$$T_{sp}(N,d,p) = \sum_{l=0}^{log_2 N} T_{chol}\left(\left(\frac{N}{2^l}\right)^{d-1}, \frac{p}{2^{ld}}\right) =$$

$$(2D) \qquad O\left(\frac{N^3}{p}\right)\gamma + O\left(\frac{N}{b}\log(s)\right)\alpha + O\left(N^2\frac{\log(s)}{s}\right)\beta$$

$$(3D) \qquad O\left(\frac{N^6}{p}\right)\gamma + O\left(\frac{N^2}{b}\log(s)\right)\alpha + O\left(N^4\frac{\log(s)}{s}\right)\beta$$

- The terms depending on the floating point operations and the message volume scale well
- The term depending on the number of messages scales bad and is smaller

References I

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. "A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication". In: IBM J. Res. Dev. 38.6 (Nov. 1994), pp. 673–681. ISSN: 0018-8646. DOI: 10.1147/rd.386.0673. URL: https://doi.org/10.1147/rd.386.0673.

[2] E. Anderson et al. LAPACK Users' Guide. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[3] S. Bauer, M. Huber, S. Ghelichkhan, M. Mohr, U. Rüde, and B. Wohlmuth. "Large-scale simulation of mantle convection based on a new matrix-free approach". In: Journal of Computational Science 31 (2019), pp. 60–76. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2018.12.006. URL: https://www.sciencedirect.com/science/article/pii/S1877750318309840.

[4] C. Bischof and C. van Loan. "The WY representation for products of householder matrices". In: SIAM J. Sci. Stat. Comput. 8.1 (Jan. 1987), pp. 2–13. ISSN: 0196-5204. DOI: 10.1137/0908009. URL: http://dx.doi.org/10.1137/0908009.

[5] J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. J. Hanson. "An extented set of Fortran Basic Linear Algebra Subprograms". In: 14 (1988), 17 and 18–32.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. "A set of level 3 basic linear algebra subprograms". In: ACM Trans. Math. Softw. 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500. DOI: 10.1145/77626.79170. URL: http://doi.acm.org/10.1145/77626.79170.

References II

[7] R. van de Geijn and J. Watts. "SUMMA: scalable universal matrix multiplication algorithm". In: CONCURRENCY: PRACTICE AND EXPERIENCE 9.4 (1997), pp. 255–274. URL: http://www.netlib.org/lapack/lawnspdf/lawn96.pdf.

[8] A. J. George. "Nested dissection of a regular finite-element mesh". In: SIAM J. Numer. Anal. 10.2 (1973), pp. 345–363. DOI: 10.1137/0710032. eprint: https://doi.org/10.1137/0710032. URL: https://doi.org/10.1137/0710032.

[9] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002. FLAME Working Note 9. 2002. URL: citeseer.ist.psu.edu/goto02reducing.html.

[10] K. Goto and R. A. v. d. Geijn. "Anatomy of high-performance matrix multiplication". In: ACM Trans. Math. Softw. 34.3 (May 2008), 12:1–12:25. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: http://doi.acm.org/10.1145/1356052.1356053.

[11] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. "Tile QR factorization with parallel panel processing for multicore architectures". In: IPDPS. IEEE, 2010, pp. 1–10. URL: http://dx.doi.org/10.1109/IPDPS.2010.5470443.

[12] N. J. Higham. Accuracy and Stabilty of Numerical Algorithms. 2. Philadelphia: SIAM, 2002.

References III

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". In: 5 (1979), pp. 308–323.

[0] S. V. Parter. "The Use of Linear Graphs in Gauss Elimination". In: SIAM Review 3 (1961), pp. 119–130.

[14] R. C. Whaley, A. Petitet, and J. J. Dongarra. "Automated Empirical Optimization of Software and the ATLAS Project". In: Parallel Computing 27.1–2 (2001). Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps), pp. 3–35.

[15] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: Communications of the ACM 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.