Introduction to high-performance computing

Ask Hjorth Larsen

Nano-bio Spectroscopy Group and ETSF Scientific Development Centre,
Universidad del País Vasco UPV/EHU, San Sebastián, Spain

Nanoscience Summerschool at Yachay Tech NSSY2017,
August 31, 2017

The CPU

- The CPU reads instructions and inputs, then performs those instructions on the inputs
- Instruction codes and inputs are processed in workspaces on the CPU called registers
- Each cycle, the CPU can execute an instruction; different CPU architectures support different instructions

Example

- Retrieve number from address A, put it in register R
- Add numbers from registers R and R', store sum in R"
- Write number from R" to address A'
- Etc.

HPC basics

- Most time is probably spent with floating point operations
- Important: Retrieve data from memory efficiently

Some programming languages

- (Assembly language)
- Fortran (1957, 1977, 1995, 2003, ...)
- C/C++
- Python — C extensions, Numpy, Scipy, ...

Floating point numbers

Computational physics mostly boils down to multiplying floating point numbers.

IEEE 754 standard for floating point numbers

- Number is represented as $M \times 2^n$
- M is the significand or mantissa
- n is the exponent

Important types

- 32-bit single precision: 24 bit for M, 8 for n
- 64-bit double precision: 53 bit for M, 11 for n

Floating point operations are complex. Modern CPUs have one or more floating point units (FPUs) that execute floating point operations efficiently

Pipelining

- Consider a 4-step operation where different "units" can process different steps simultaneously

	u1	u2	u3	u4
Cycle 1	A^1	ø	ø	ø
Cycle 2	B^1	A^2	ø	ø
Cycle 3	C^1	B^2	A^3	ø
Cycle 4	D^1	C^2	B^3	A^4
Cycle 5	E^1	D^2	C^3	B^4
Cycle 6	F^1	E^2	D^3	C^4
Cycle 7	ø	F^2	E^3	D^4
Cycle 8	ø	ø	F^3	E^4
Cycle 9	ø	ø	ø	F^4

- Can execute up to one whole operation per cycle, but cycles may be wasted flushing/filling the pipeline
- Next input element must be readily available

Branching and pipelining

- Jumping around in memory breaks the pipeline
- Avoid branching in high-performance loops: `if` statements, function calls, `goto`, ...
- Jump can be eliminated by inlining — include the source of a function "inline" in place of calling the function, e.g. using a macro
- Also: `inline double myfunction(...)`

Memory and multilevel caching

Example: Intel i7-4770 Haswell architecture

	Size	Latency	Total
L1 cache	64 KB/core	4–5 cycles	1.3 ns
L2 cache	256 KB/core	12 cycles	3.5 ns
L3 cache	8 MB, shared	36 cycles	11 ns
Main memory	32 GB, shared	36 cycles $+ 57$ ns	68 ns

Source: `http://www.7-cpu.com/cpu/Haswell.html`

- When accessing memory, contiguous chunks of adjacent memory will be copied into cache
- A failed cache lookup is called a “cache miss”
- Upon cache miss, element is looked up at next (slower) level

Arrays and memory layout

- Standard mathematical matrix notation:

$$\begin{bmatrix} 1,1 & 1,2 & 1,3 \\ 2,1 & 2,2 & 2,3 \\ 3,1 & 3,2 & 3,3 \end{bmatrix}$$

- Elements of the array are stored in a contiguous chunk of memory, but the ordering depends on language
- Fortran memory order is column-major:

$1,1$	$2,1$	$3,1$	$1,2$	$2,2$	$3,2$	$1,3$	$2,3$	$3,3$

- C memory order is row-major:

$1,1$	$1,2$	$1,3$	$2,1$	$2,2$	$2,3$	$3,1$	$3,2$	$3,3$

- Accessing elements in memory order is fast.

Optimizing cache use

- Work on contiguous chunks of memory

```
// fast
for(i=0; i < I; i++) {
  for(j=0; j < J; j++) {
    a[i * J + j] = ...
  }
}

// Slow
for(j=0; j < J; j++) {
  for(i=0; i < I; i++) {
    a[i * J + j] = ...
  }
}
```

Benchmark

Matrix multiplication $c_{ij} = \sum_k a_{ik} b_{kj}$

```
void matmul_ikj(int I, int J, int K,
                double *A, double *B, double *C)
{
  int i, j, k;
  for(i=0; i < I; i++) {
    for(k=0; k < K; k++) {
      for(j=0; j < J; j++) {
        C[i * J + j] += A[i * K + k] * B[k * J + j];
      }
    }
  }
}
```

Different permutations of $\{ikj\}$ loops will perform differently

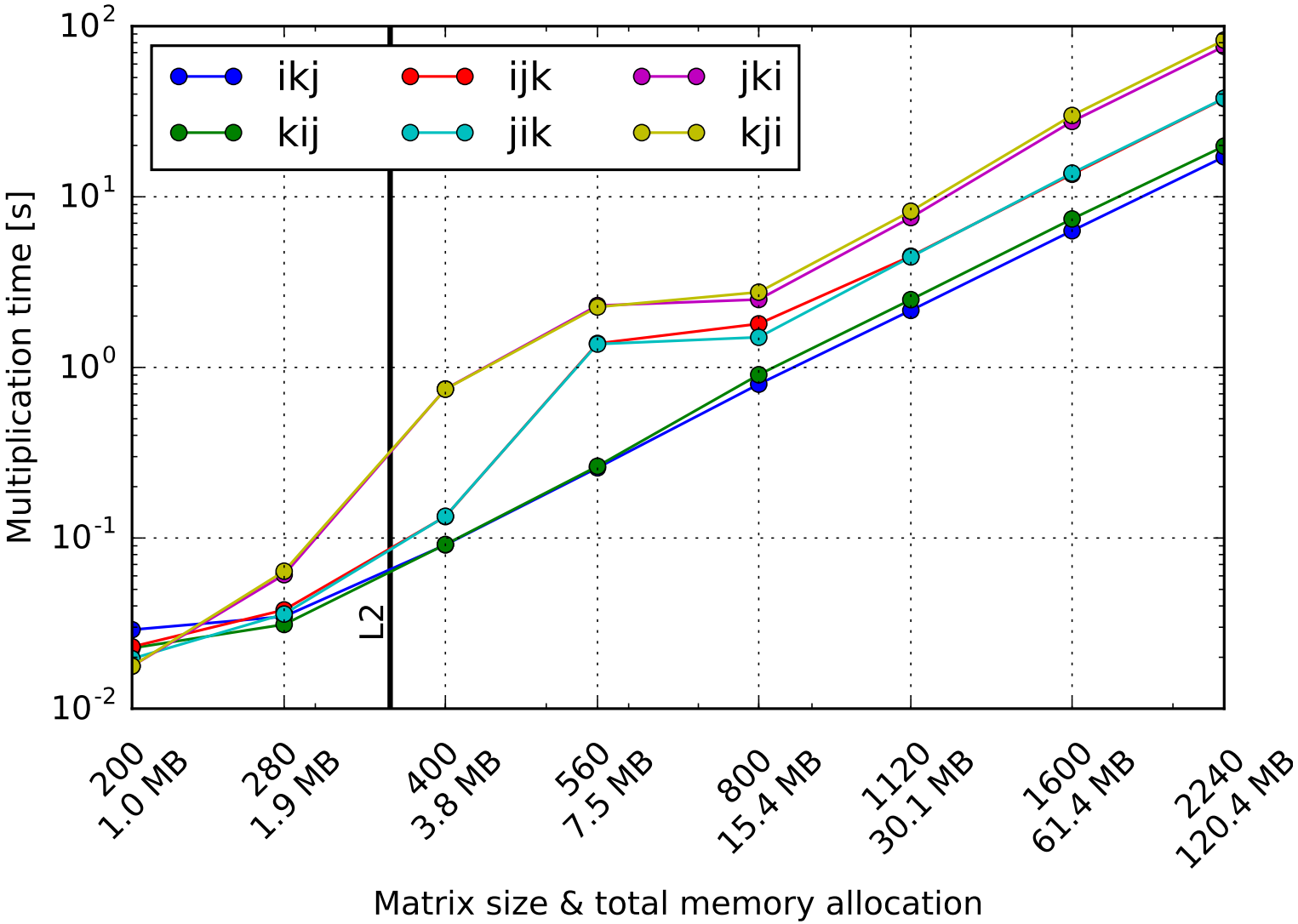


Figure: Timings for matrix multiplication at -02 optimization level

Loop unrolling

- Unrolling eliminates a fraction of loop bounds checks.

```
for(i=0; i < 4; i++) {
     a[i] = b[i] * c[i];
}
```

- Unrolled:

```
a[i] = b[i] * c[i];
a[i + 1] = b[i + 1] * c[i + 1];
a[i + 2] = b[i + 2] * c[i + 2];
a[i + 3] = b[i + 3] * c[i + 3];
```

- Compiler may be able to unroll automatically (e.g. `-funroll-loops`).

Blocking

- Compute $\mathbf{C} = \mathbf{AB}$ where each matrix is composed into blocks:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1n} \\ \vdots & & \vdots \\ \mathbf{A}_{n1} & \cdots & \mathbf{A}_{nn} \end{bmatrix}$$

- Matrix product expressed with blocks:

$$\mathbf{C}_{ij} = \sum_k \mathbf{A}_{ik}\mathbf{B}_{kj}$$

- Work on smaller blocks that fit into cache
- Optimal blocksize depends on architecture (e.g. cache size)
- Matrix product scales as $\mathcal{O}(N^3)$
- Blocking improves $\mathcal{O}(N^3)$ prefactor by working on chunks that fit in cache

BLAS

Basic Linear Algebra Subprograms

- Standard interface for standard operations: Matrix multiplication
- Highly optimized for different platforms individually

Some BLAS implementations

- RefBlas — reference implementation from Netlib
- OpenBlas (based on older GotoBlas)
- Atlas — automatically tuned linear algebra software
- Intel MKL
- AMD ACML

Some BLAS functions

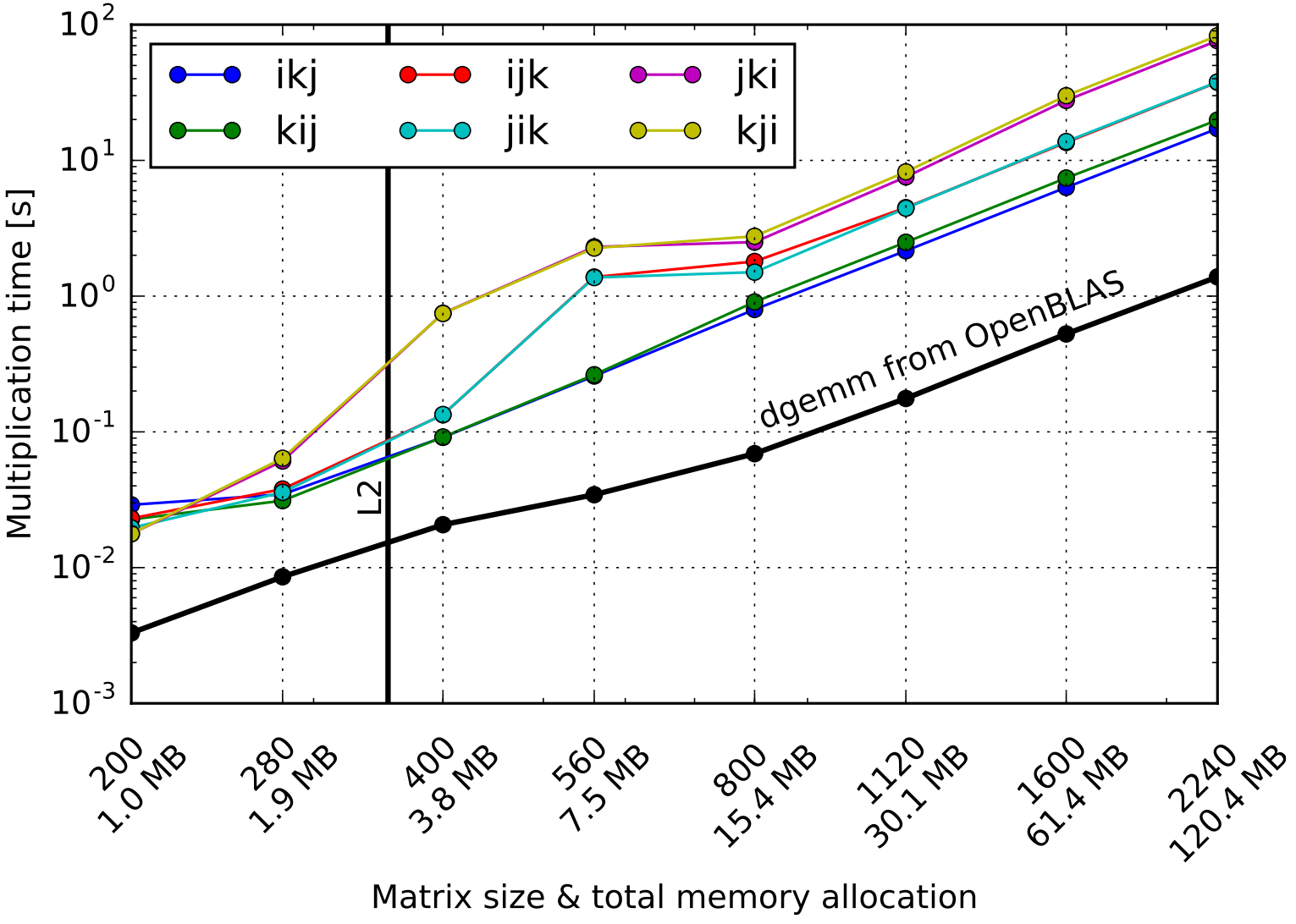
- `dgemm`: double-precision general matrix–matrix multiply
- `dsymv`: double-precision symmetric matrix–vector multiply
- `daxpy`: double-precision $a\mathbf{X}$ plus y
- `zsyr2k`: "complex double-precision (z) symmetric rank-2 update", $\mathbf{X}\mathbf{Y}^T + \mathbf{Y}\mathbf{X}^T$
- Etc.

LAPACK: Linear Algebra PACKage

- Higher-level linear algebra operations
- LU-decomposition, eigenvalues, ...
- `dsyev`: double-precision symmetric eigenvalues
- Etc.

For best performance, use BLAS/LAPACK whenever possible

Simple matrix multiplication vs BLAS



Shared memory

- Multiple threads work simulaneously, access same variables
- Threads may read the same memory simultaneously, but simultaneous writing leads to race condition
- Threads must therefore synchronize access to the memory (e.g. `synchronized` methods and blocks in Java)
- Synchronize means: "Lock, run, unlock"

Distributed memory

- Each process has its own chunk of memory, probably on different physical computers
- No problem with synchronizing memory (unless also threading)
- Must manually send/receive all data; much more difficult

Parallel pitfalls and deadlocks

Example 1

- Process 1 sends 5 numbers to process 2
- Process 1 expects something from process 2
- Process 2 expects 6 numbers from process 1, receives 5
- Both processes now wait forever

Parallel pitfalls and deadlocks

Example 2

- Process A reserves Sala Capitular for this talk
- Process B attempts to reserve Sala Capitular for this talk, but Sala Capitular is already reserved for some talk
- Process B schedules this talk for another lecture room
- ...

MPI — Message Passing Interface

- Programming interface specification for distributed-memory parallelization
- Implementations: OpenMPI, MPICH, ...
- Communicator: Object which represents a group of processes that may communicate amongst themselves
- `MPI_COMM_WORLD` — the communicator of all processes
- The size of a communicator is how many processes participate
- Each process has a rank within the communicator: 0, 1, 2, ..., `size`−1.
- Run on 8 cores: `mpirun -np 8 myprogram`

Parallel hello world

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Comm_rank(comm, &rank); // Ranks enumerate processes
    MPI_Comm_size(comm, &size);
    printf("hello world from process %d/%d\n", rank, size);
    MPI_Barrier(comm);  // Wait for all processes to print

    int ranksum;
    MPI_Allreduce(&rank, &ranksum, 1, MPI_INTEGER, MPI_SUM, comm);
    printf("rank %d: I got %d\n", rank, ranksum);
    MPI_Finalize();
    return 0;
}
```

```c
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    double num;
    MPI_Status status;
    if(rank == 0) {
        num = 42.0;  // Pass number around to each process
        printf("rank 0 sends %f to rank 1\n", num);
        MPI_Send(&num, 1, MPI_DOUBLE, 1, 0, comm);
        MPI_Recv(&num, 1, MPI_DOUBLE, size - 1, 0, comm, &status);
        printf("rank 0 finally received %f\n", num);
    } else {
        MPI_Recv(&num, 1, MPI_DOUBLE, rank - 1, 0, comm, &status);
        printf("rank %d received %f from %d, sends to %d\n",
                rank, num, rank - 1, (rank + 1) % size);
        MPI_Send(&num, 1, MPI_DOUBLE, (rank + 1) % size, 0, comm);
    }
    MPI_Finalize();
    return 0;
}
```

BLACS/ScaLAPACK

- BLACS: Basic Linear Algebra Communication Subprograms
- ScaLAPACK: Like LAPACK, but in parallel
- BLACS uses "2D block-cyclic memory layout": Processes are arranged in a 2D grid, arrays are distributed in blocks
- Distribution of blocks among ranks:

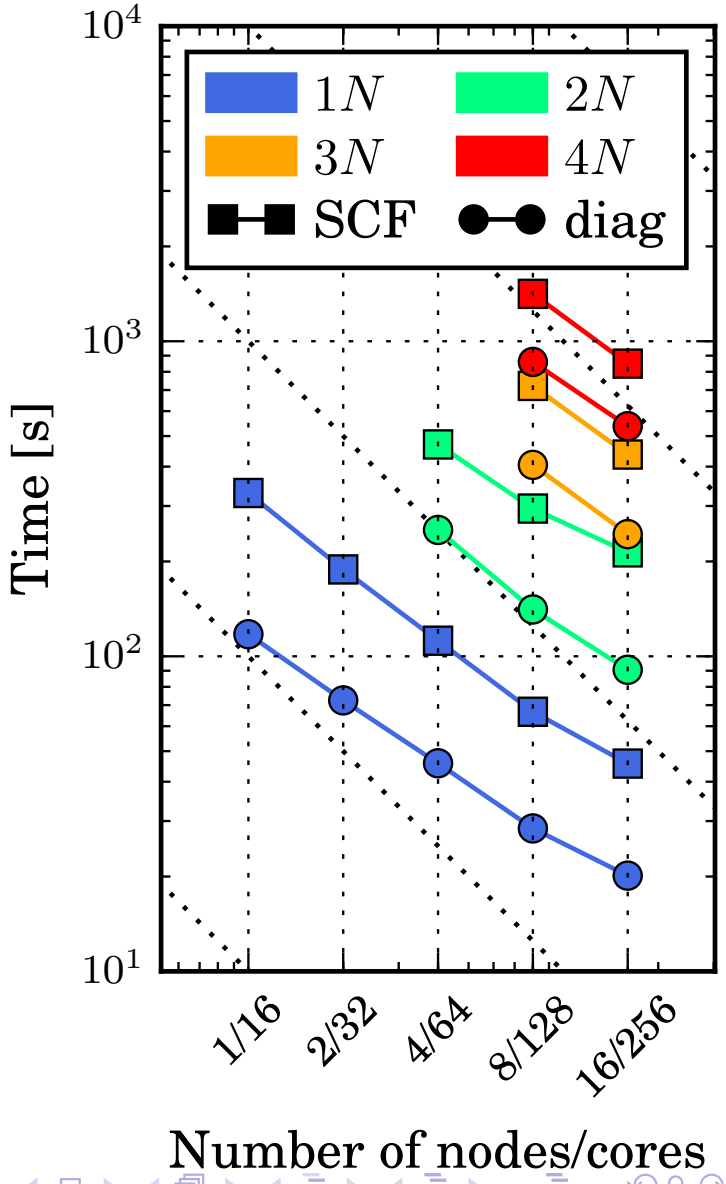
0	1	2	0	1	2
3	4	5	3	4	5
0	1	2	0	1	2
3	4	5	3	4	5

- `pdgemm`, `pdsyev`, ...

Parallel scaling

GPAW/LCAO performance

- Time per iteration, 2500–10k atoms
- "Strong-scaling" test: Fixed problem size, increase CPU count
- More processes increase speed, but also overhead
- More processes may be necessary when calculation does not fit into memory



Quinde supercomputer at Yachay

- 84 compute nodes, dual Power-8 10-core CPUs (20 cores/node)
- Dual NVidia K-80 graphics cards, 8 tera-FLOPS
- 128 GB memory per node
- Interconnect: Mellanox 100 Gbit InfiniBand

Massively parallel architectures

- Hundreds of thousands of cores, very scalable
- High demands on interconnect: Network topology, locality

Figure: IBM BlueGene/P (Image source: Wikipedia)

GPUs — graphics cards for computing

- A graphics card is a shared-memory processor running a very large number of threads
- Graphics cards are the cheapest way of multiplying many floating point numbers
- Special architecture: Code must be explicitly written for graphics cards

GPU performance

- On a normal processor, each thread (a, b, c) should work on a contiguous piece of data (1, 2, 3):

a1	a2	a3	b1	b2	b3	c1	c2	c3

- On a graphics card, memory bandwidth (main memory to graphics card memory) is critical
- Threads a, b, and c can start quickly only with a strided memory layout:

a1	b1	c1	a2	b2	c2	a3	b3	c3

- Here, threads a, b, c will all run once we have received three chunks
- In the previous case, b and c would still be idle after receiving three chunks

Summary & concluding remarks

- Pipelining, memory locality
- Parallelization: Threading, MPI
- HPC libraries: BLAS, LAPACK, ScaLAPACK