HPC Programming Bootcamp

Day 1: Introduction

Abhinav Bhatele, Department of Computer Science

UNIVERSITY OF MARYLAND

Bootcamp information

- Location: Iribe 4105 from 9:30-11:45 am, 1:15-4:00 pm
- Labs will be in the afternoon
- Website: https://hpcbootcamp.readthedocs.io
- Lecture slides and lab info posted online before class

Overview

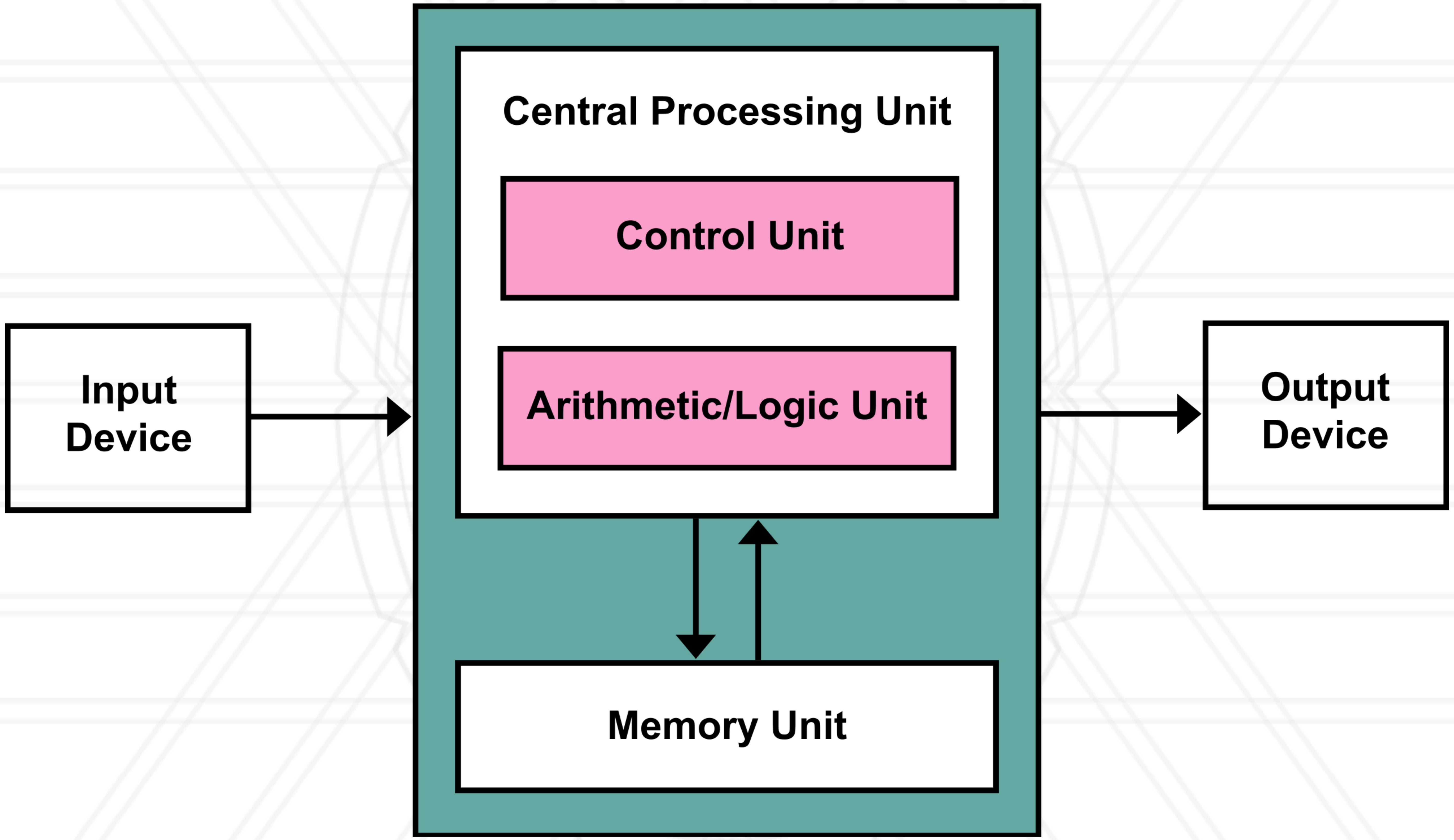
- Day 1: Introduction to serial and parallel programming
  - Computer architecture
  - Measuring performance and optimizing serial code
  - Parallel hardware
- Day 2: Writing OpenMP programs
  - Overview of parallel programming
  - Writing OpenMP programs
  - Profiling parallel applications

Overview

- Day 3: Writing MPI programs
  - Writing MPI programs
  - Parallel performance
  - Optimizing parallel performance
- Day 4: Other programming models
  - Charm++
  - RAJA

Introduction

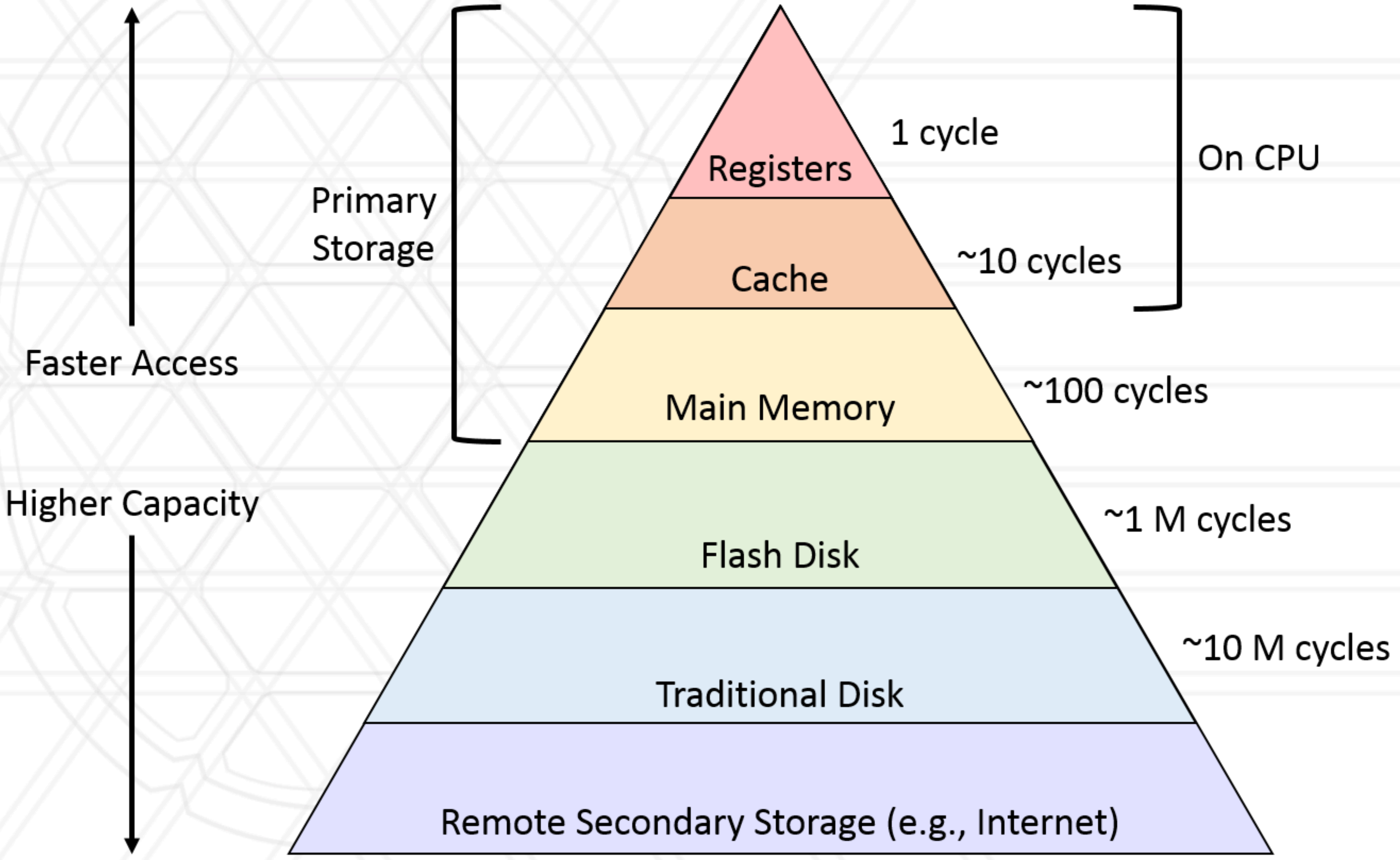
von Neumann architecture



https://en.wikipedia.org/wiki/Von_Neumann_architecture

Memory hierarchy

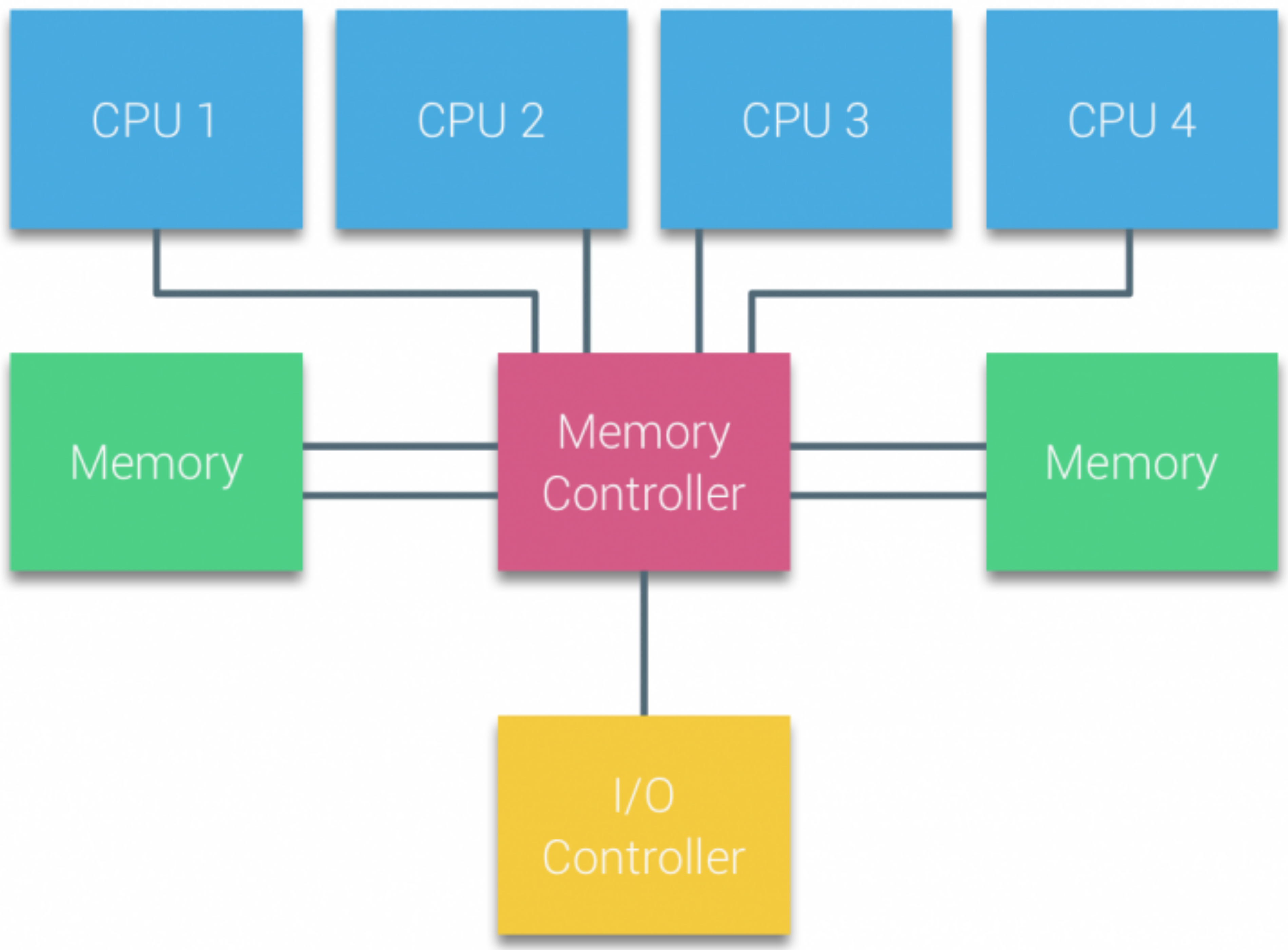
- All levels of memory hierarchy are getting faster



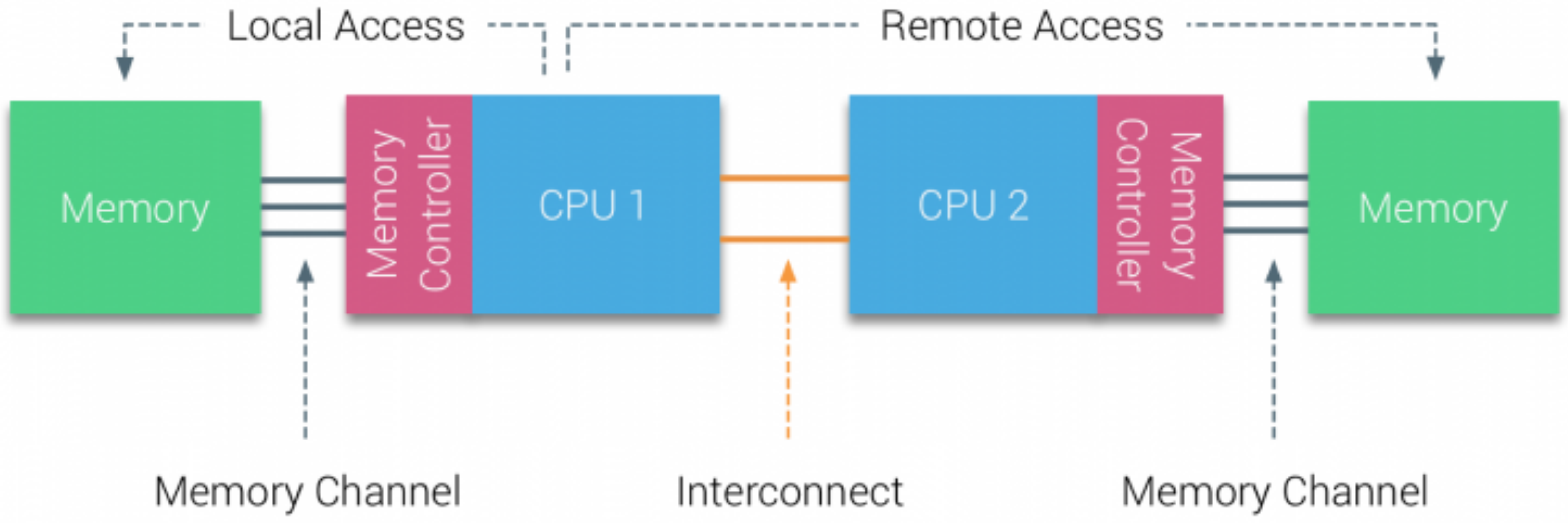
The Memory Hierarchy

https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html

Memory access: UMA vs. NUMA



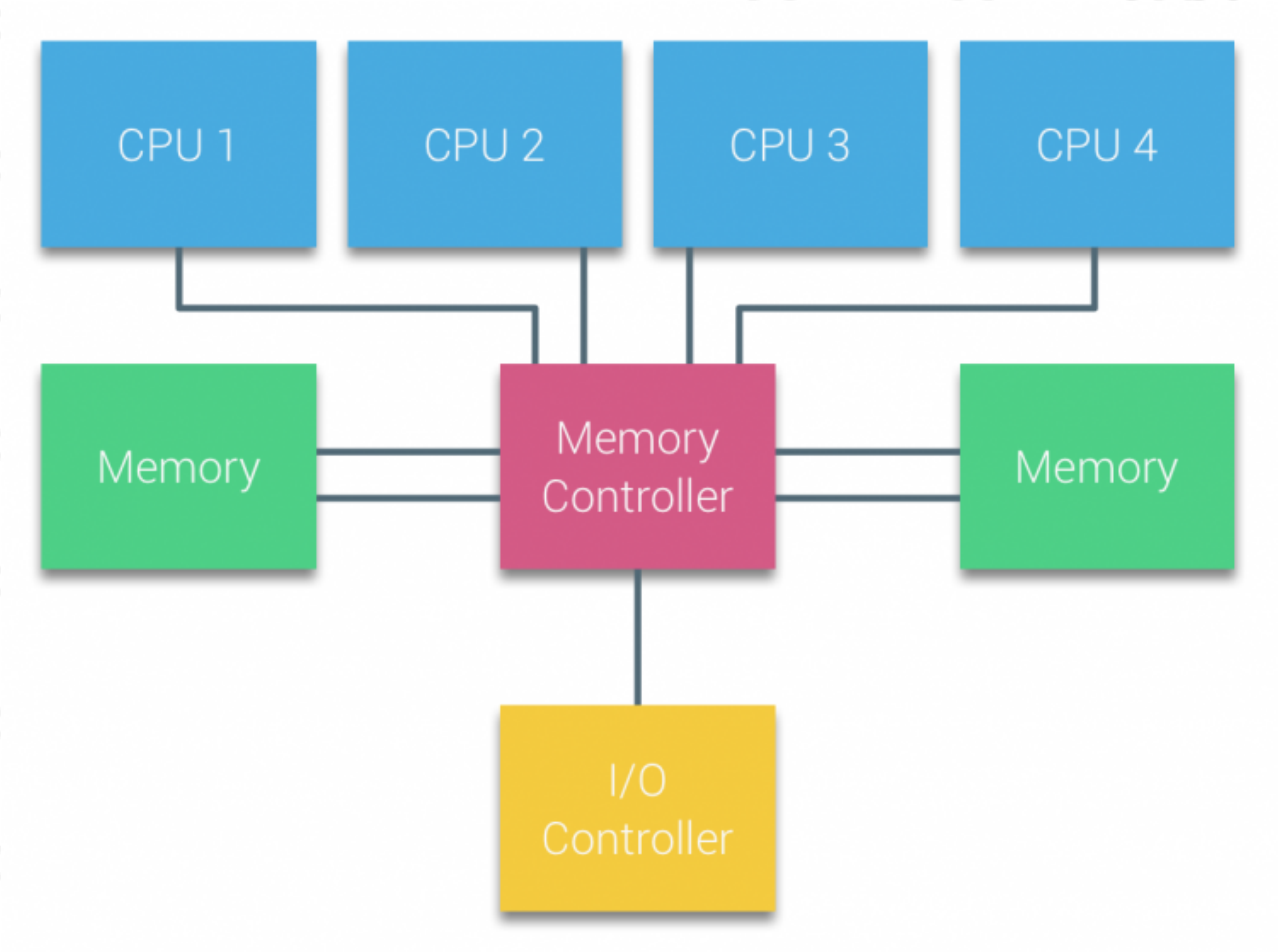
Uniform Memory Access



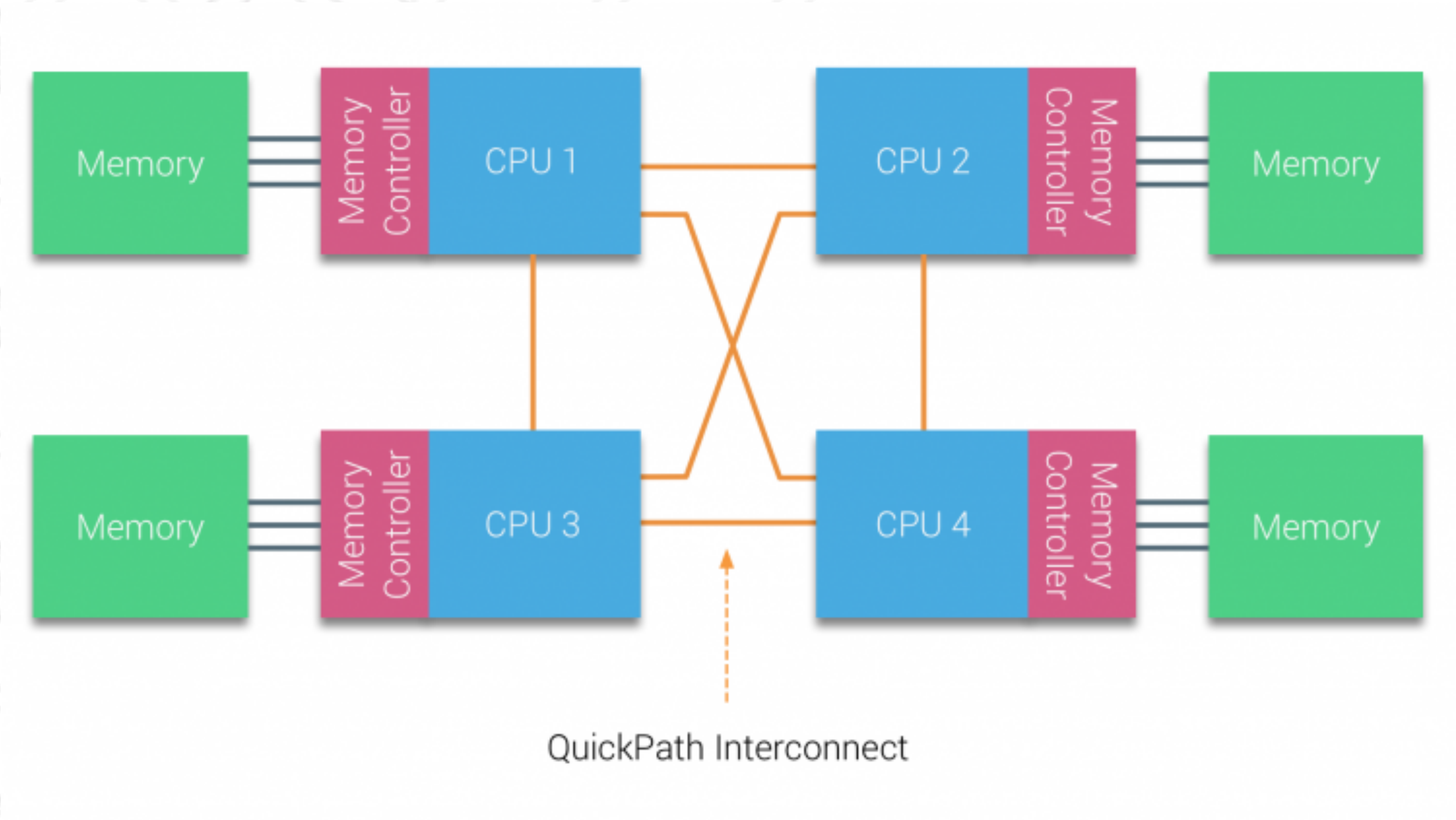
Non-uniform Memory Access

https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/

Memory access: UMA vs. NUMA



Uniform Memory Access

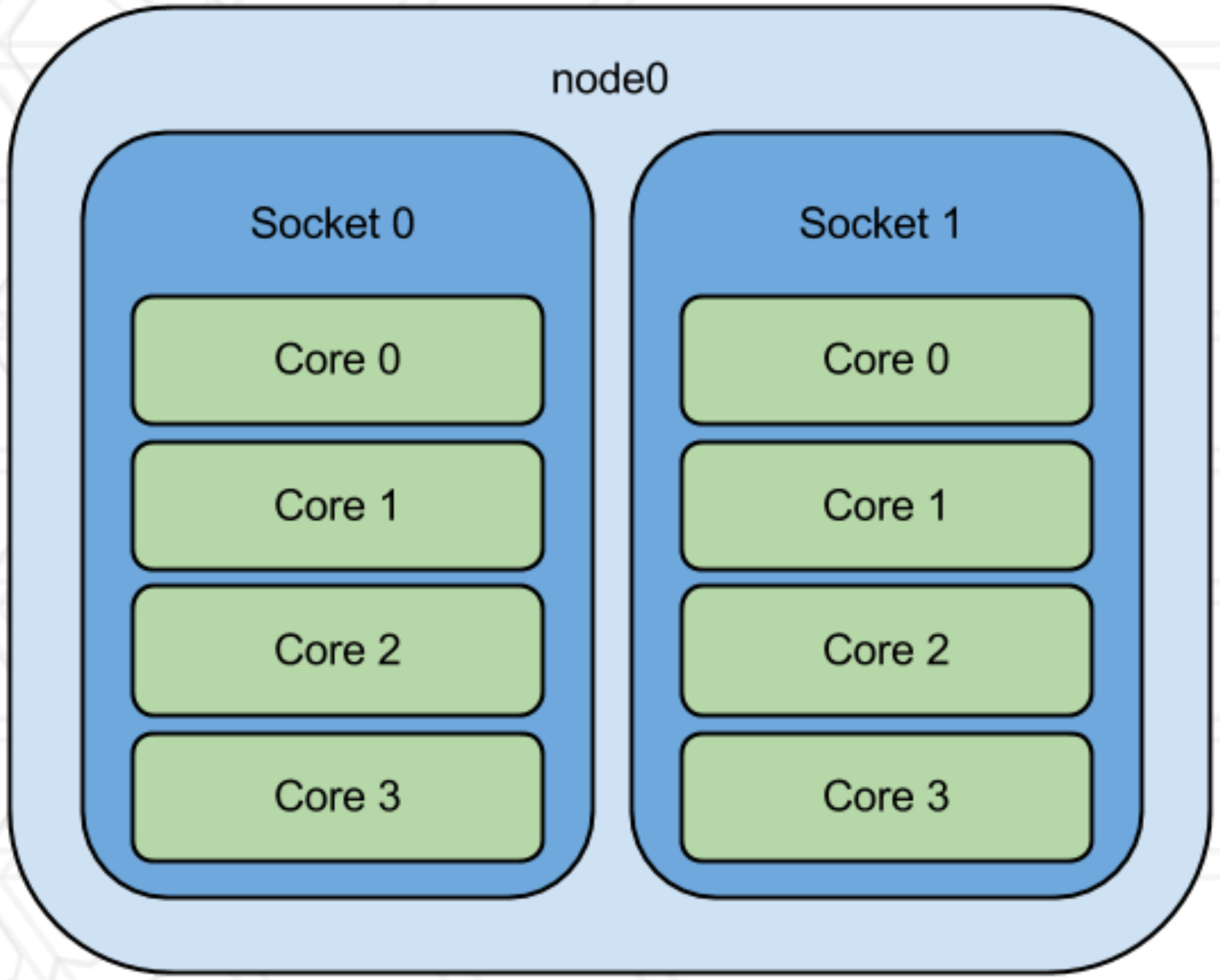


Non-uniform Memory Access

https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/

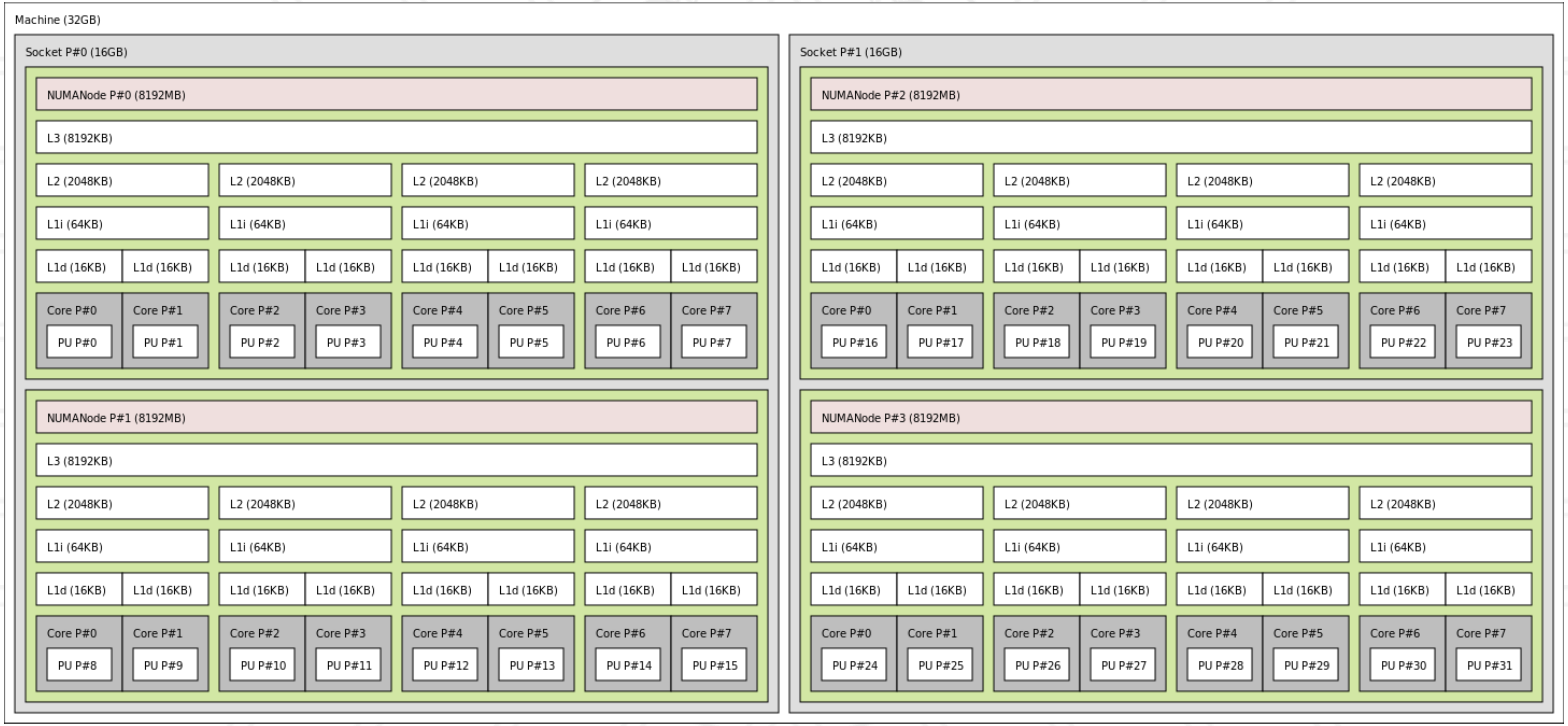
Definitions: Cores, sockets, nodes

- CPU: processor
  - Single or multi-core: core is a processing unit, multiple such units on a single chip make it a multi-core processor
- Socket: chip
- Node: packaging of sockets



https://www.glennklockwood.com/hpc-howtos/process-affinity.html

A multi-socket node



AMD Bulldozer: https://en.wikipedia.org/wiki/Memory_hierarchy

Definitions: Serial vs. parallel code

- Thread: a thread or path of execution managed by the OS
- Process: heavy-weight, processes do not share resources such as memory, file descriptors etc.
- Serial or sequential code: can only run on a single thread or process
- Parallel code: can be run on one or more threads or processes

Measuring performance

Measuring performance (execution time)

- Use the `time` system call
- Add *timers* to your code
- Use a performance tool: gprof

Definitions: Wall clock vs CPU time

- Elapsed or wall clock time is the total time from start to finish
- CPU or process time is the time spent in a process
  - Doesn't include time when the process was stopped by others such as for I/O
  - Includes time when the system is running user code and system code

Using the time command

- Prefix time on the command line before your executable

```
$ time ./program <args>

real  0m0.809s
user  0m0.734s
sys   0m0.019s
```

- `Real`: Elapsed time
- `User`: Time spent in the user code
- `Sys`: Time spent in the kernel

int gettimeofday(struct timeval *tv, struct timezone *tz);

```
#include <sys/time.h>

...
struct timeval start, end;

gettimeofday(&start, NULL);
/* do work */
gettimeofday(&end, NULL);

long long elapsed = (end.tv_sec - start.tv_sec) * 1000000000ll
                  + (end.tv_usec - start.tv_usec) * 1000ll;
```

int getrusage(int who, struct rusage *usage);

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

...
struct rusage start, end;

getrusage(RUSAGE_SELF, &start);
/* do work */
getrusage(RUSAGE_SELF, &end);

long long elapsed = (end.ru_utime.tv_sec - start.ru_utime.tv_sec)
* 1000000000ll
                + (end.ru_utime.tv_usec - start.ru_utime.tv_usec)
* 1000ll;
```

int getrusage(int who, struct rusage *usage);

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

...
struct rusage start, end;

getrusage(RUSAGE_SELF, &start);
/* do work */
getrusage(RUSAGE_SELF, &end);

long long elapsed = (end.ru_utime.tv_sec - start.ru_utime.tv_sec)
* 1000000000ll
                + (end.ru_utime.tv_usec - start.ru_utime.tv_usec)
* 1000ll;
```

who:

RUSAGE_SELF
RUSAGE_CHILDREN
RUSAGE_THREAD

Tools to measure performance: gprof

- Compile program with -pg

```
$ gcc -pg -O3 -o pgm pgm.c
```

- Run the program
  - Outputs gmon.out

```
$ ./pgm
```

- Run gprof on the output

```
$ gprof pgm gmon.out
```

Sample gprof output

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 60.03      0.03     0.03                             element_matrices
 40.02      0.05     0.02                             smvp
  0.00      0.05     0.00    35025     0.00     0.00  inv_J
  0.00      0.05     0.00     1303     0.00     0.00  area_triangle
  0.00      0.05     0.00        1     0.00     0.00  arch_parsecl
```

Things to consider

- Performance variation from run-to-run
  - Better to take multiple measurements and then take the mean
- Input arguments
  - Are they representative of a production run

Optimizing code

Optimizations done by hardware

- Instruction pipelining
  - Execute different parts of instructions in parallel
- Branch prediction
  - Speculatively execute the most likely branch

Optimizations done by the compiler

- Important to remember the compiler option -O*N*, *N* = 1, 2, 3
  - Should only enable safe optimizations that do not change the result of a correct program
  - May discover latent bugs
- Compiler optimizations:
  - https://en.wikipedia.org/wiki/Category:Compiler_optimizations
  - Loop-invariant code motion
  - Loop unrolling
  - Dead code elimination

Typical performance problems

- Slow algorithm — needs a significant re-write
- Forget to turn on compiler optimization
- Debugging printfs in the code
- Inefficient input/output (I/O)
- Cache/memory performance

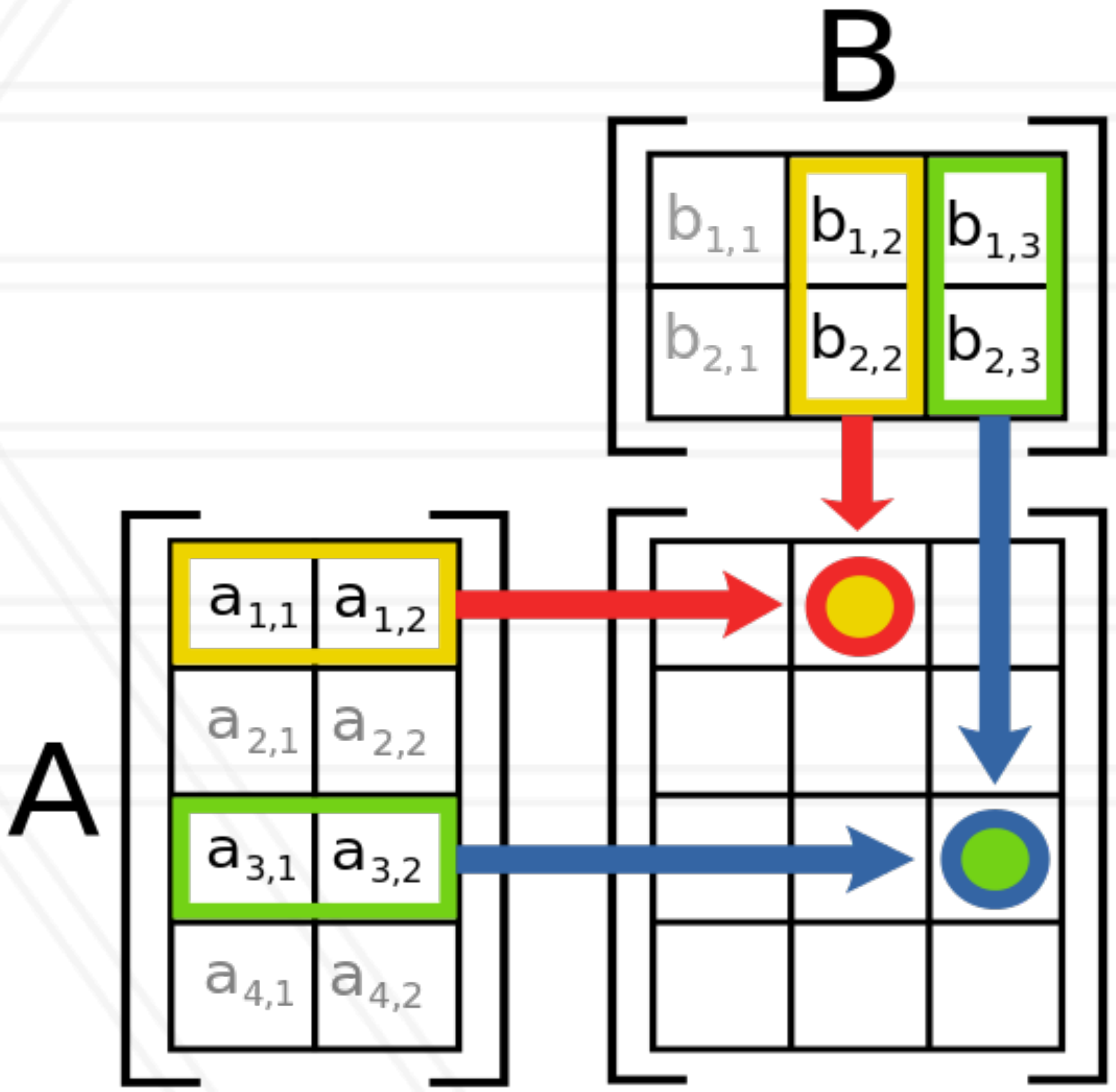
Good software practices

- Function inlining
- Efficient data layout and access
- Remove unnecessary data movement

Principle of locality

- Temporal locality: Data that was referenced recently is likely to ne referenced again
- Spatial locality: Data nearby tends to be referenced together

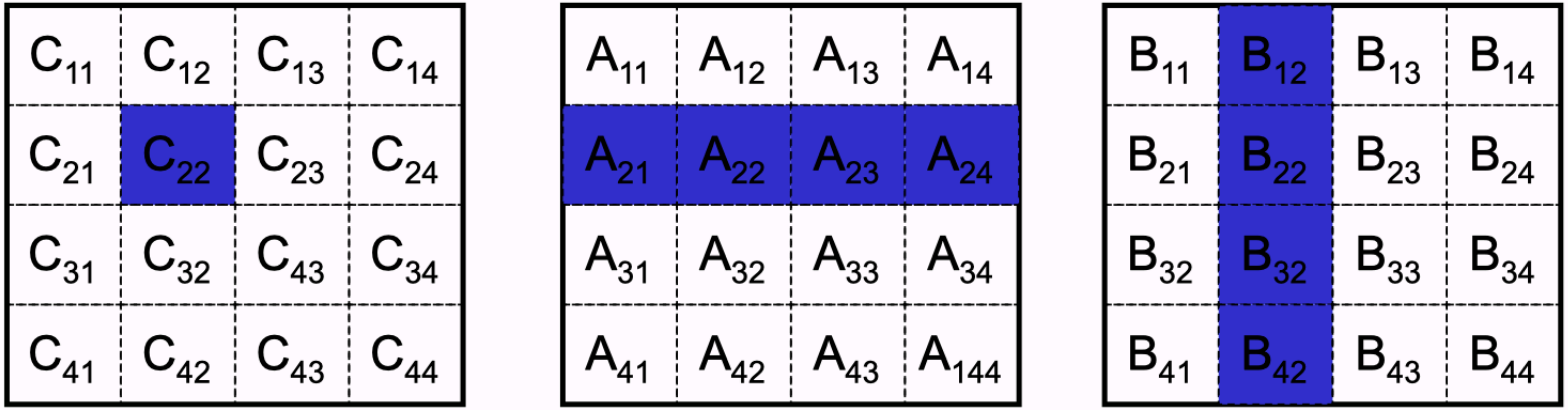
```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
```



https://en.wikipedia.org/wiki/Matrix_multiplication

Blocking to improve cache performance

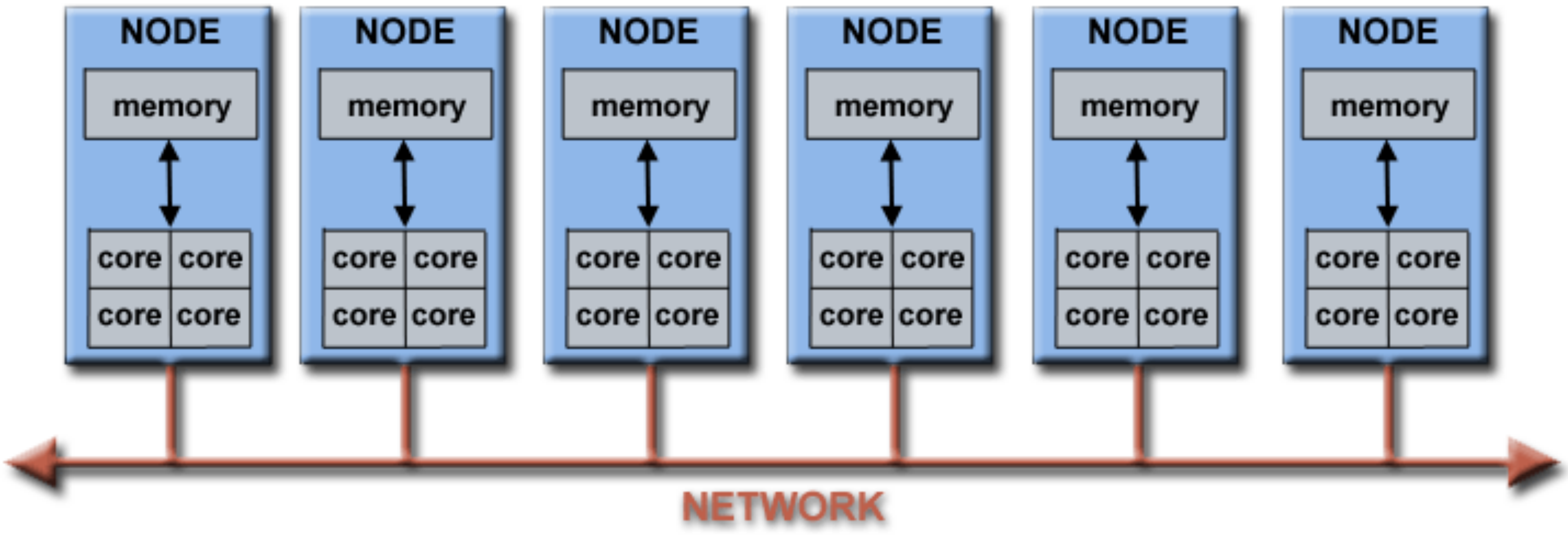
- Create smaller blocks that fit in cache
- $C_{22} = A_{21} * B_{12} + A_{22} * B_{22} + A_{23} * B_{32} + A_{24} * B_{42}$



Parallel Architecture

Parallel Architecture

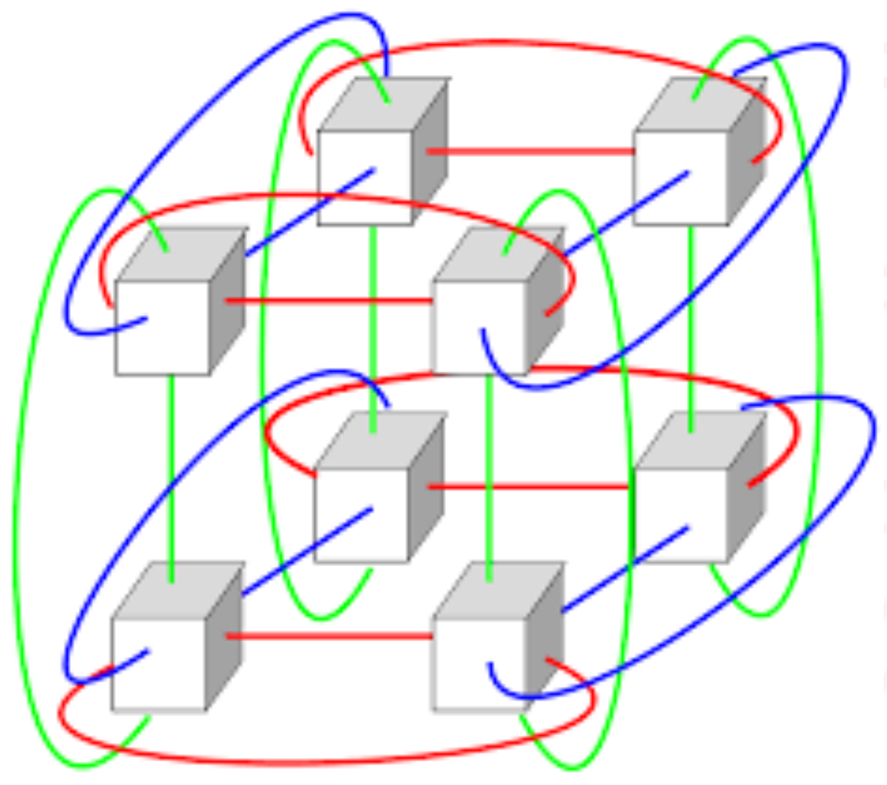
- A set of nodes or processing elements connected by a network.



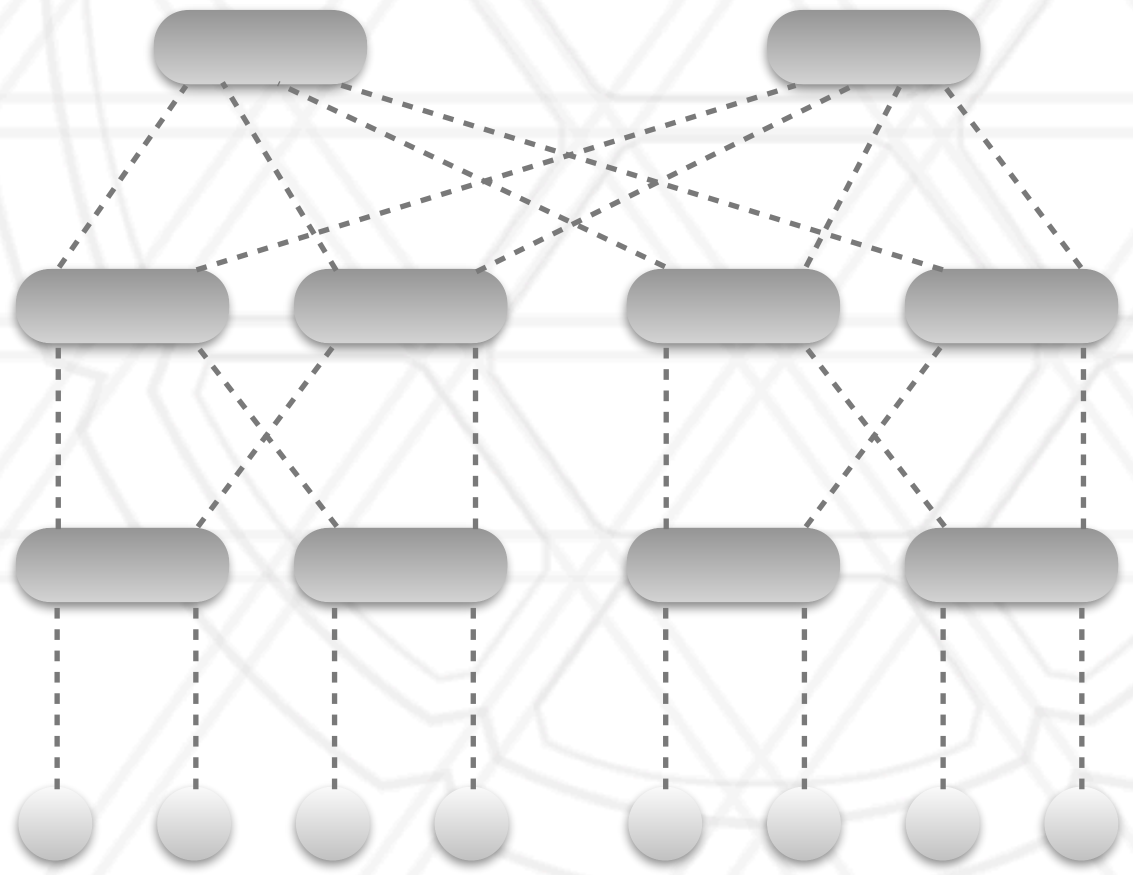
https://computing.llnl.gov/tutorials/parallel_comp

Interconnection networks

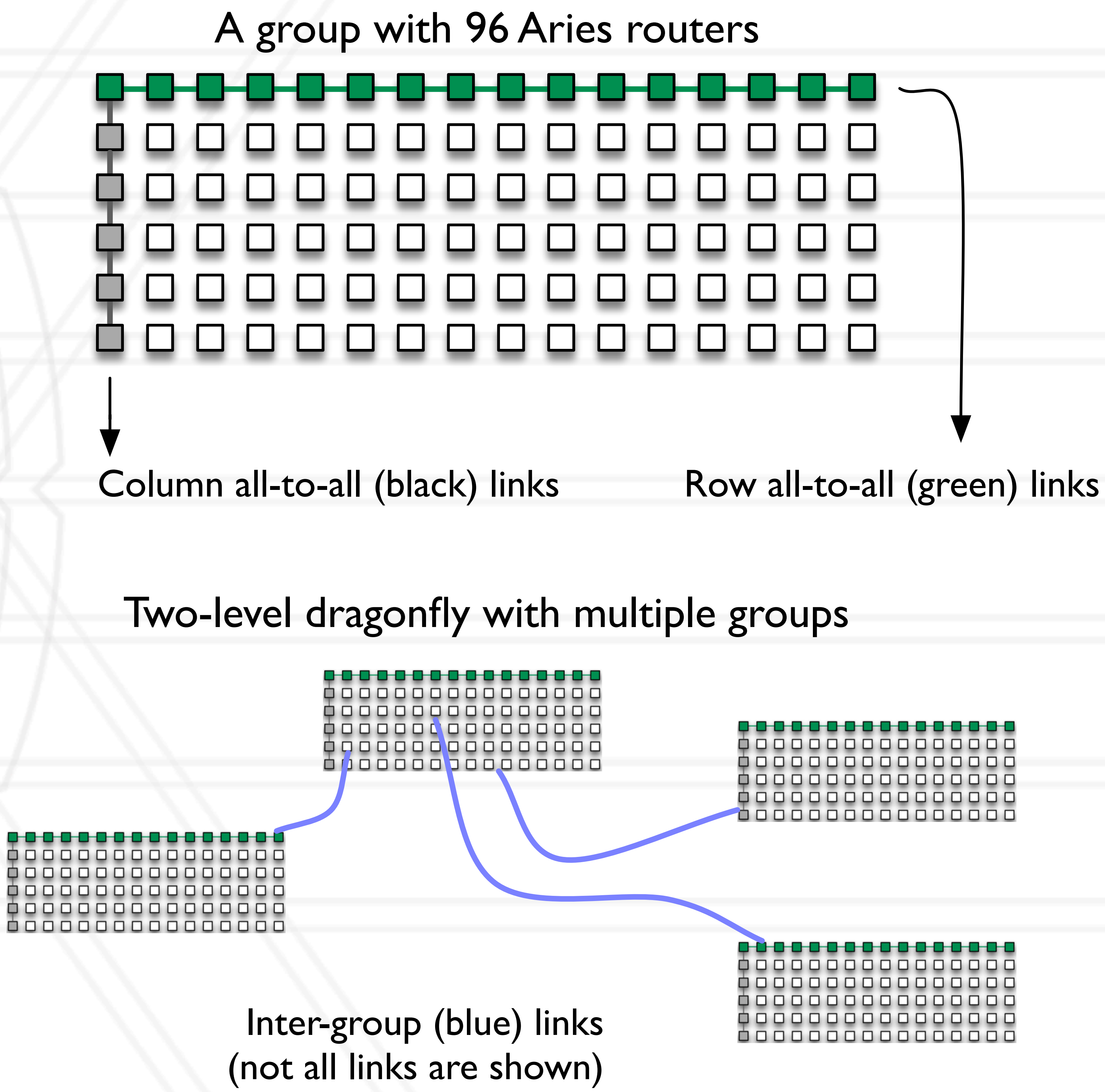
- Different topologies for connecting nodes together
- Used in the past: torus, hypercube
- More popular currently: fat-tree, dragonfly



Torus



Fat-tree



Dragonfly

Memory and I/O sub-systems

- Similar issues for both memory and disks (storage):
  - Where is it located?
  - View to the programmer vs. reality
- Performance considerations: latency vs. throughput

Questions?

UNIVERSITY OF MARYLAND

Abhinav Bhatele

5218 Brendan Iribe Center (IRB) / College Park, MD 20742

phone: 301.405.4507 / e-mail: bhatele@cs.umd.edu