CS302

Parallel Computing

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi, Fatahalian and Wenisch of CMU, EPFL, Michigan, Wisconsin

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Parallel Computing
  - Why parallelism?
  - Principles of Parallel Computing
  - OpenMP
- Thursday
  - Lecture: Coherence refresher
  - Exercise session: Example OpenMP programs
- Friday lab session
  - Assignment 1 released!

Homeworks & Assignments

- Weekly homework
  - Homework one + solution posted
  - Homework two is posted
- Assignment one is posted
  - Deadline: March 23rd at 23:59
- Two more assignments with the following tentative schedule:
  - March 25th
  - May 6th

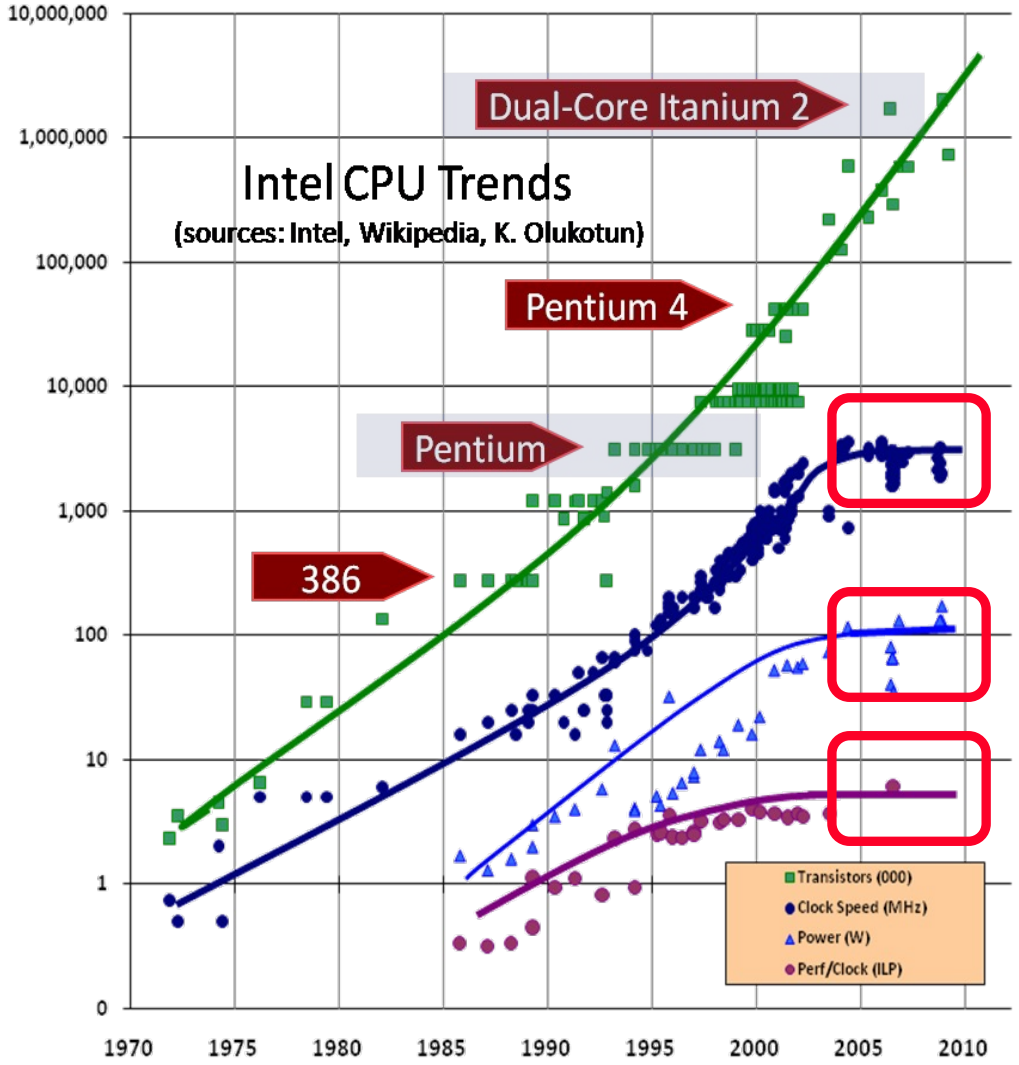
Assignment 1: Shared Memory Programming

- Objective:
  - Learn to parallelize code in a shared memory programming model
  - Understand the impact of several optimizations on performance

- Two parts:
  1. Parallelize MCI with OpenMP for two different functions
  2. Optimize two different algorithms keeping in mind hardware factors

Review: Why Parallelism?

- The free lunch is over
  - Power Wall
    - End of frequency scaling
  - ILP tapped out
    - Little hidden parallelism is left
- Moore's Law reinterpreted
  - Chip density increases slowly
  - Clock speed does not
- Parallelism is a key solution to achieving higher performance



Example program

- Compute $\sqrt[3]{v}$ using Newton's method
  - Find root of function $f(x) = x^3 - v$
  - $x_{i+1} = x_i - \frac{f(x)}{f\prime(x)}$

Example program

- Compute $\sqrt[3]{v}$ using Newton's method
  - $x_{i+1} = \frac{1}{3}\left(2x_i + \frac{v}{x_i^2}\right)$
  - For each element of an array of N floating-point numbers

```
void my_cbrt(float *v, float *result, int N) {
    for (int i = 0; i < N; ++i) {
        float x = 1.f;
        for (int j = 0; j < 10; ++j)
            x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));
        result[i] = x;
    }
}
```

Example program

- Compute $\sqrt[3]{v}$ using Newton's method
  - $x_{i+1} = \frac{1}{3}\left(2x_i + \frac{v}{x_i^2}\right)$
  - For each element of an array of N floating-point numbers

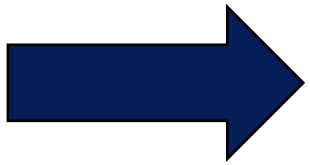
```
void my_cbrt(float *v, float *result, int N) {
    for (int i = 0; i < N; ++i) {
        float x = 1.f;
        for (int j = 0; j < 10; ++j)
            x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));
        result[i] = x;
    }
}
```

Example program - Compilation

- Compute $\sqrt[3]{v}$ using Newton's method:
  - For each element of an array of N floating-point numbers

```
for (int i = 0; i < N; ++i) {
    float x = 1.f;

    for (int j = 0; j < 10; ++j)
        x = (1.f / 3.f) *
            (2.f*x + v[i] / (x*x));

    result[i] = x;
}
```

Compile



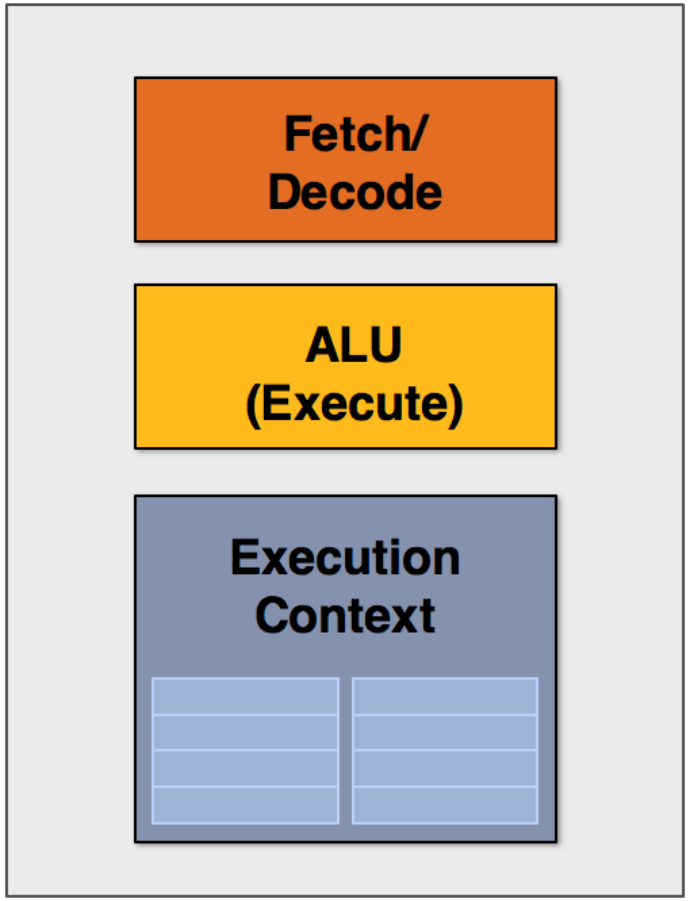
```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
...
...
...
...
...
...
st   addr[r2], r0
```

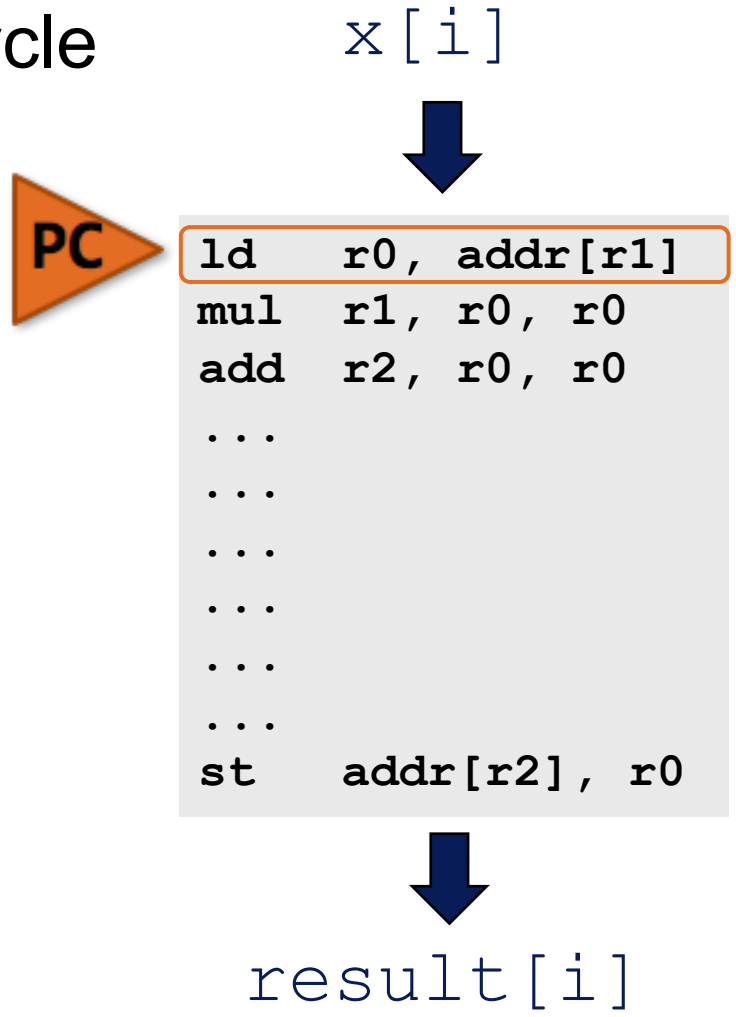
x[i]

result[i]

Example program - Execution

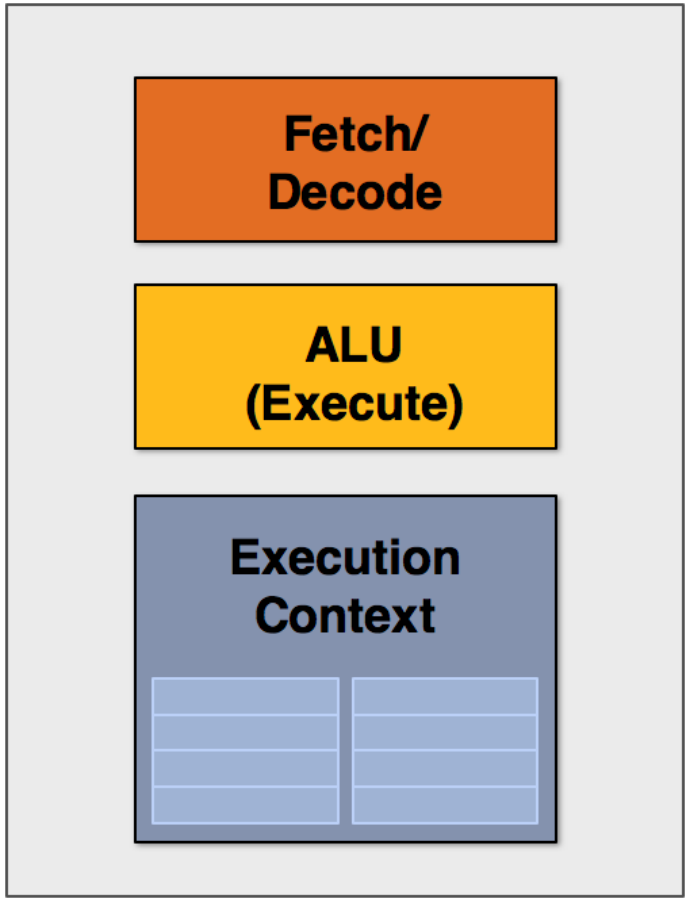
Simple core: executes one instruction per cycle

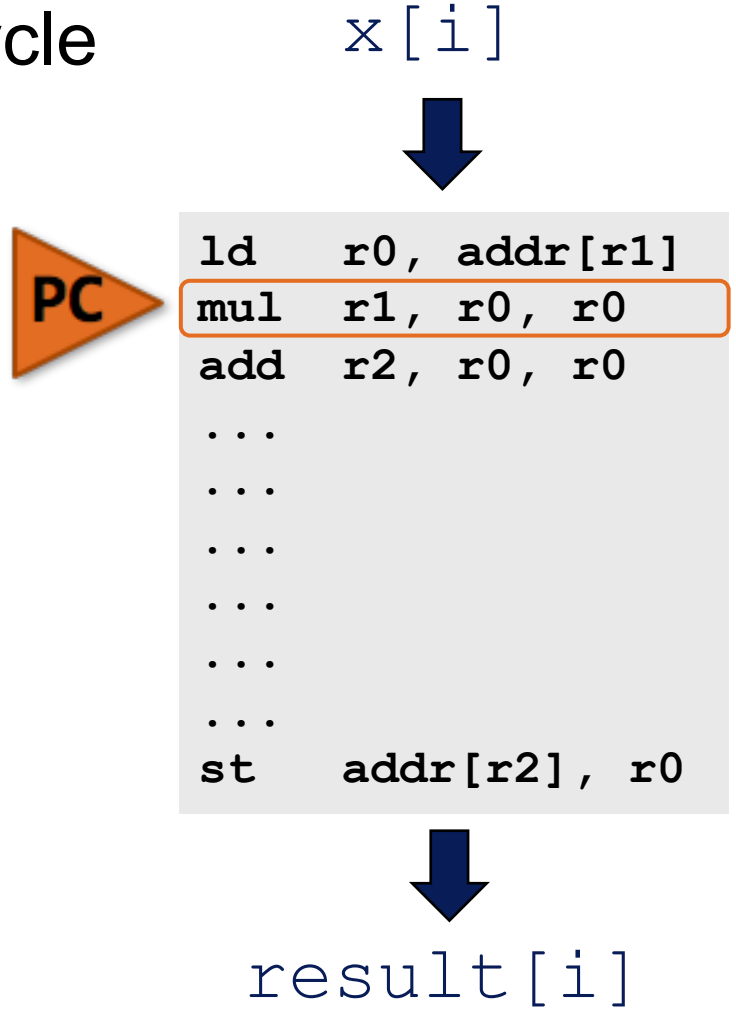




Example program - Execution

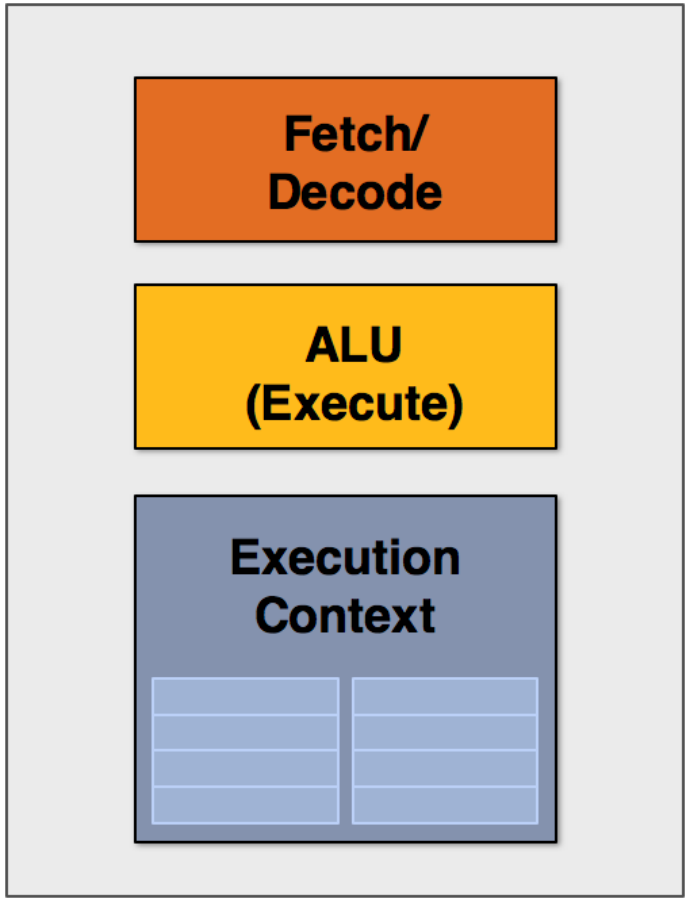
Simple core: executes one instruction per cycle

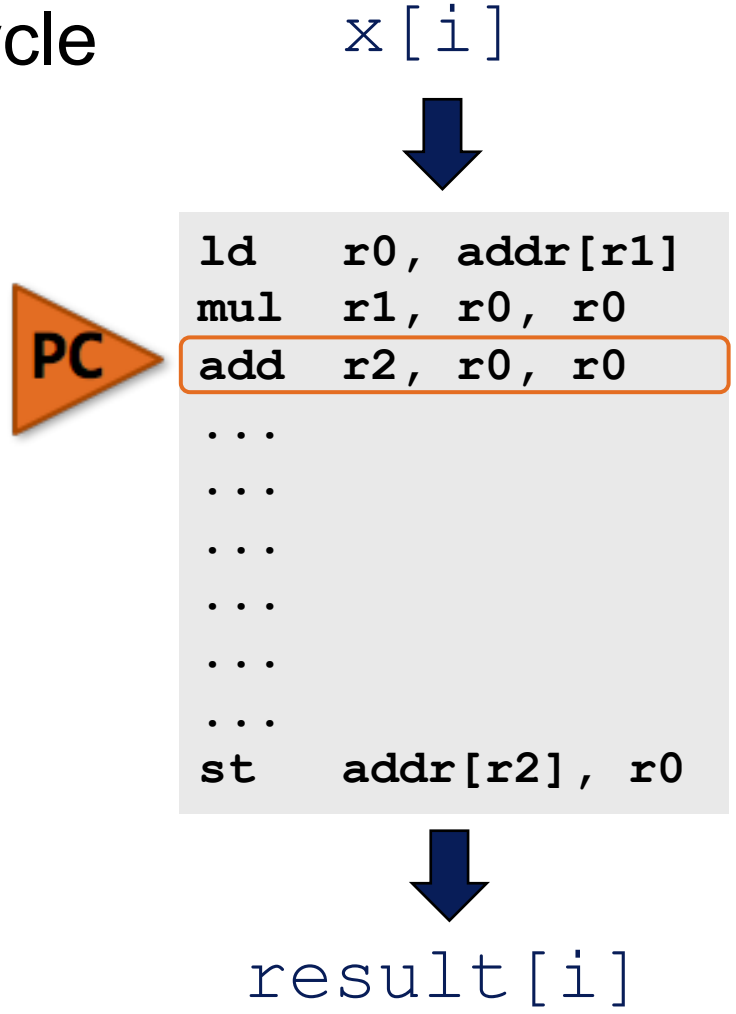




Example program - Execution

Simple core: executes one instruction per cycle





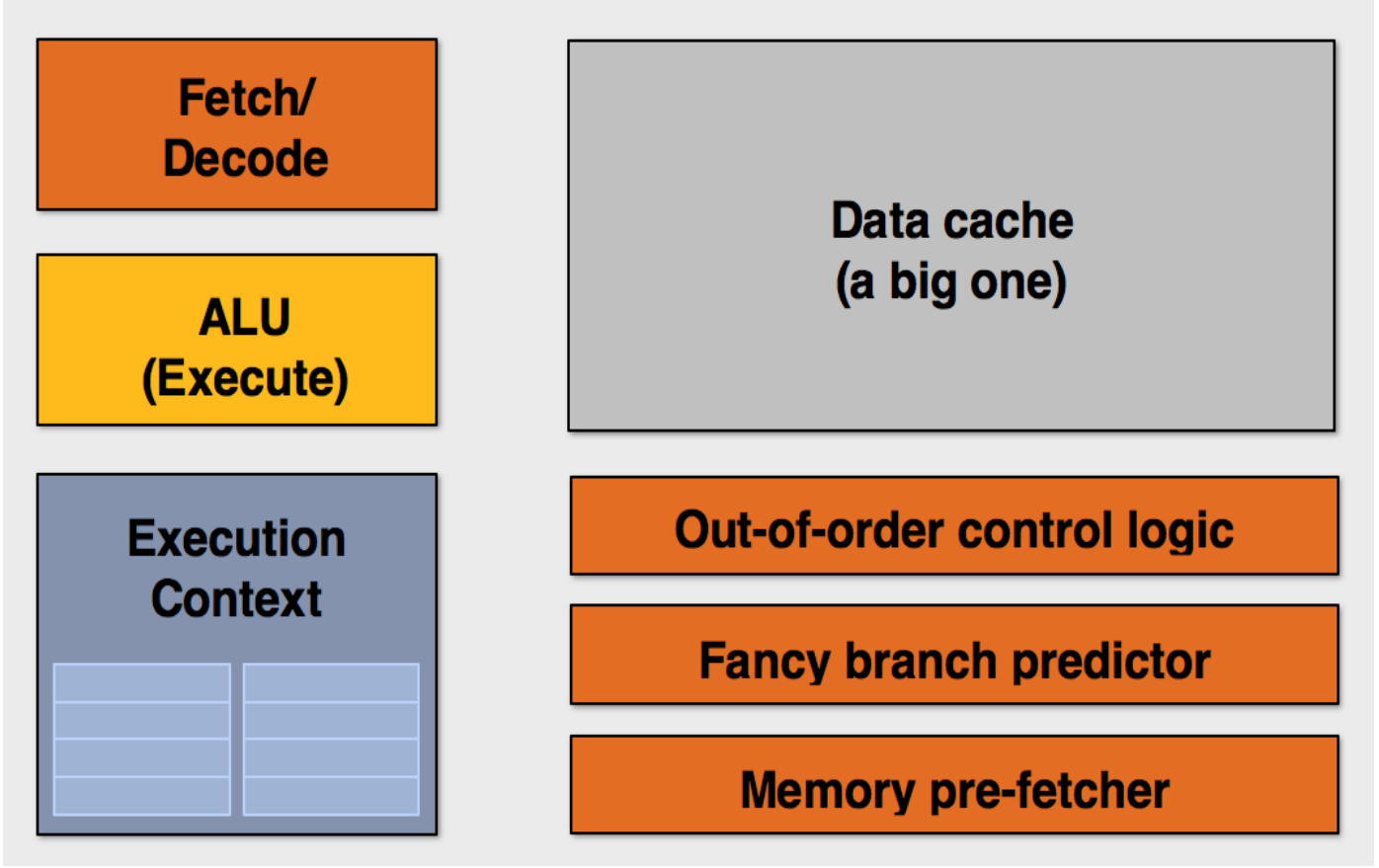
Pre-multicore Era: One BIG fancy core

Majority of transistor budget spent on:

- run single instruction stream faster

More transistors means

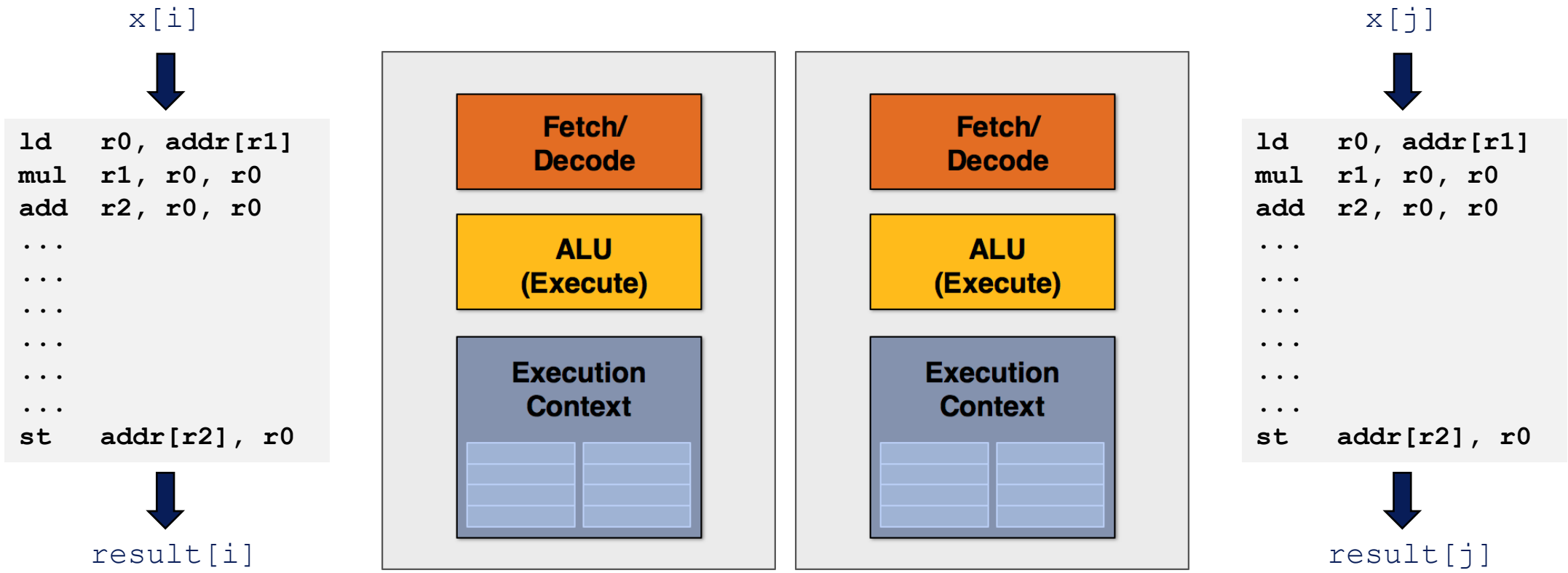
- Wider superscalar
- Smarter branch predictor
- Larger cache hierarchy
- etc.



CPU in Multicore Era – Idea #1

Use more transistors, but in simpler cores

- Each core is slower than original “fat” core (e.g., 25% slower)
- But there are now two: 2 × 0.75 = 1.5 (potential for speedup!)



But what about our program?

No parallelism is expressed in our code

- It will be only executed on one of the cores
- 25% slower than the original one

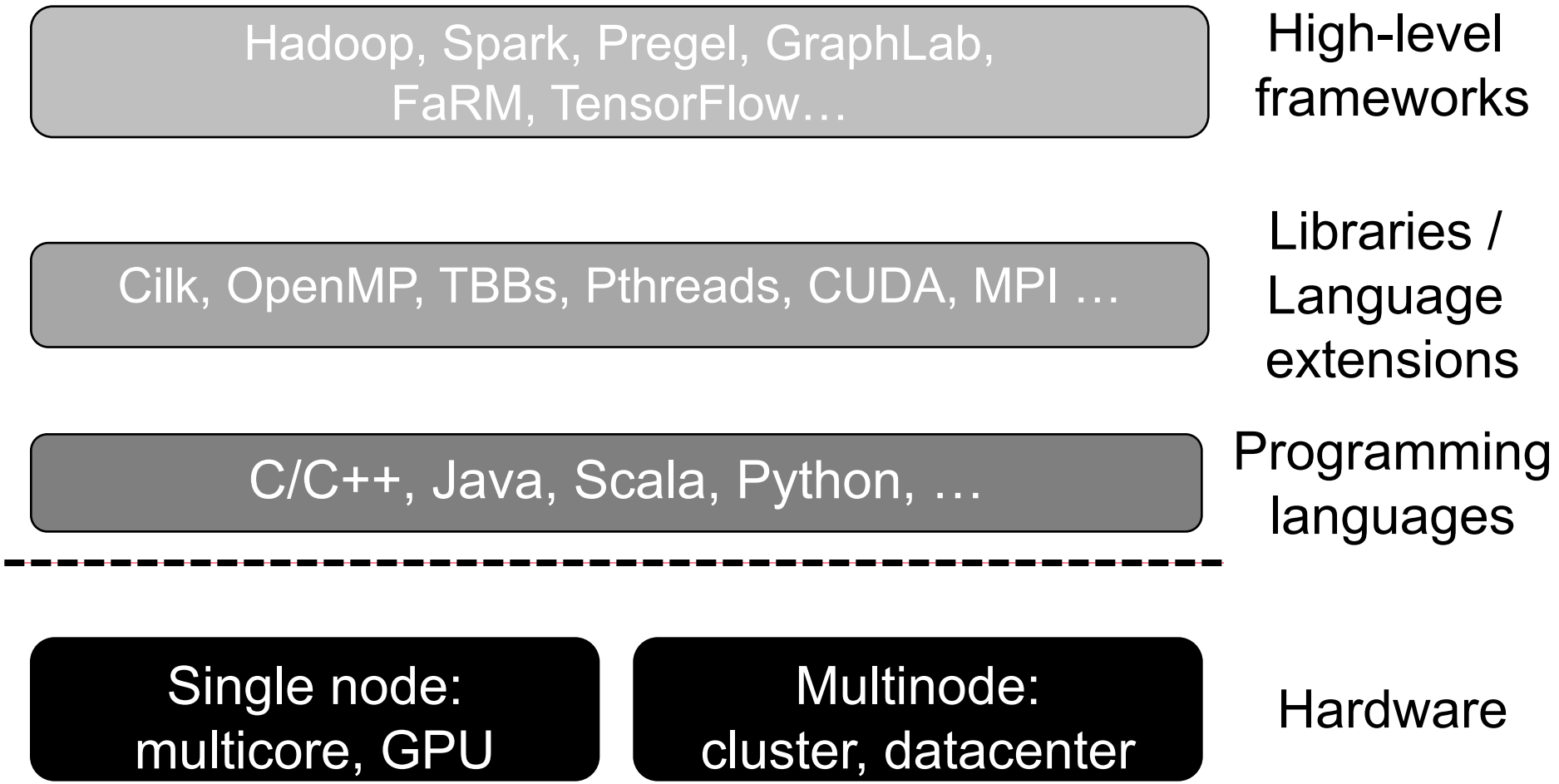
```
void my_cbrt(float *v, float *result, int N) {
    for (int i = 0; i < N; ++i) {
        float x = 1.f;
        for (int j = 0; j < 10; ++j)
            x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));
        result[i] = x;
    }
}
```

Expressing Parallelism

Programmer

- Use existing parallel code through libraries
  - Spiral, ScaLAPACK, BLISS
- Writing in a parallel programming language
  - HPF, CAF, UPC, CxC, Cilk, Java, Scala
- Writing in compiler directives
  - OpenMP, Intel Threading Building Blocks
- Writing using a threading library
  - MPI, PVM, Pthreads
- Domain Specific Languages (DSLs)
  - Spiral, Halide, CUDA

Software layering



PThreads

```
typedef struct {
  int N; float* v; float* result;
} my_args;

void my_thread_start(void* thread_arg) {
  my_args* thread_args = (my_args*)thread_arg;
  my_cbrt(args->v, args->result, args->N);
}
```

PThreads

```
typedef struct {
  int N; float* v; float* result;
} my_args;

void my_thread_start(void* thread_arg) {
  my_args* thread_args = (my_args*)thread_arg;
  my_cbrt(args->v, args->result, args->N);
}

void parallel_cbrt(float* v, float* result, int N) {
  pthread_t thread_id;
  my_args args;
  args.N = N/2;
  args.v = v;
  args.result = result;
  pthread_create(&thread_id, NULL, my_thread_start, &args);
  my_cbrt(v + args.N, result + args.N, N - args.N);
  pthread_join(thread_id, NULL);
}
```

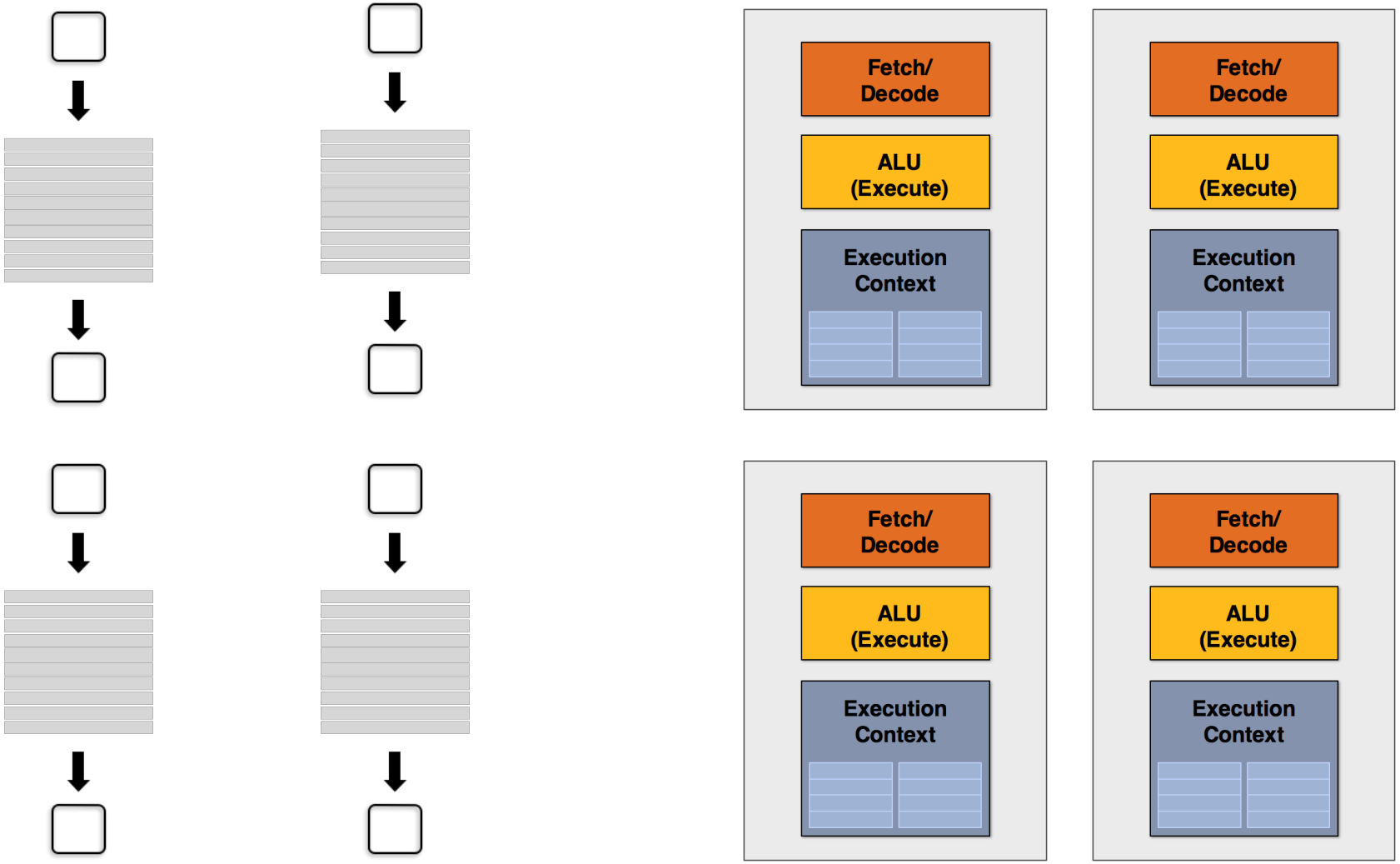
Expressing Parallelism Using OpenMP

Loop iterations are independent of each other

- Compiler can automatically generate parallel threaded code

```
void my_cbrt(float *v, float *result, int N) {
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        float x = 1.f;
        for (int j = 0; j < 10; ++j)
            x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));
        result[i] = x;
    }
}
```

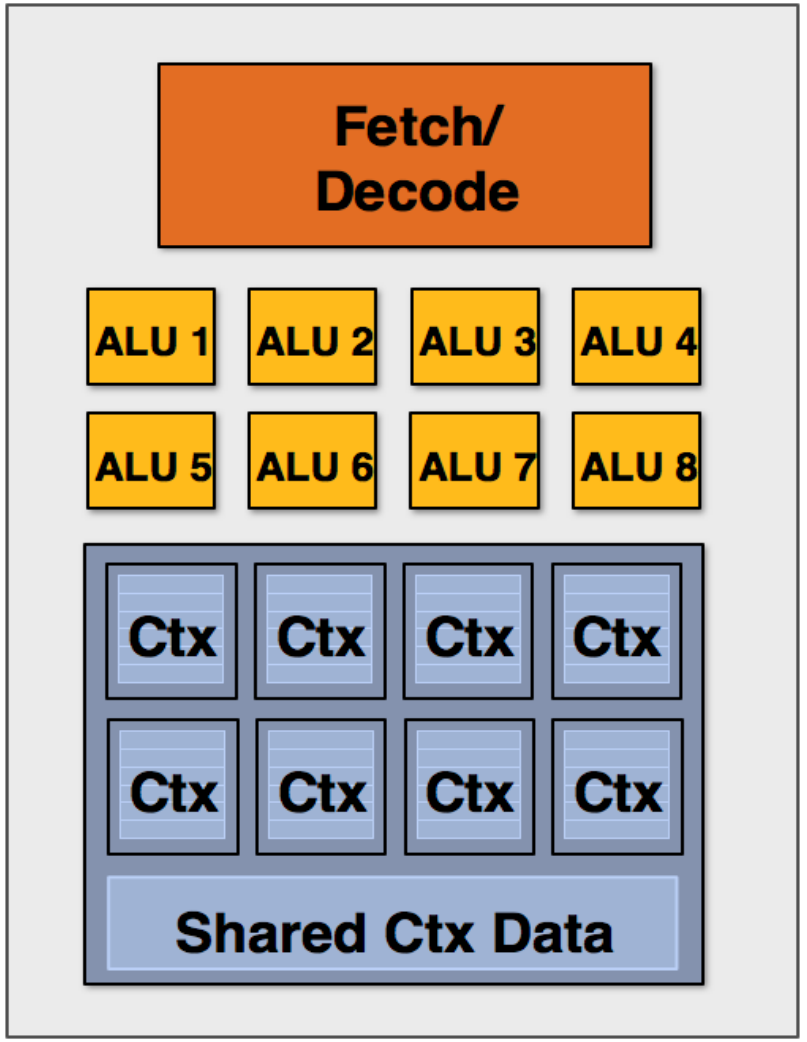
Four Cores: Compute 4 Elements in Parallel



CPU in Multicore Era – Idea #2

Add ALUs to increase compute capability

- Amortize cost of managing an inst. stream across many ALUs
- SIMD Processing:
  Single Instruction, Multiple Data
- Same inst. broadcast to all ALUs, executed in parallel on all ALUs



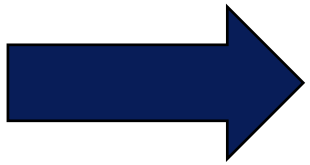
Scalar Program

Recall our original compiled program:

- Processes one element using scalar insts on scalar registers

```
for (int i = 0; i < N; ++i) {
    float x = 1.f;

    for (int j = 0; j < 10; ++j)
        x = (1.f / 3.f) *
            (2.f*x + v[i] / (x*x));

    result[i] = x;
}
```

Compile



```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
...
...
...
...
...
...
st   addr[r2], r0
```

x[i]

result[i]

Vector Program (Using AVX Instructions): Detail Next Week

```
#include <immintrin.h>

__m256 my_cbrt_vec(__m256 v) {
    __m256 x = _mm256_set1_ps(1.f);

    for (int j = 0; j < 10; ++j) {
        __m256 tmp = _mm256_add_ps(_mm256_add_ps(x, x),
                     _mm256_div_ps(v, _mm256_mul_ps(x, x)));
        x = _mm256_mul_ps(_mm256_set1_ps(1.f / 3.f), tmp);
    }

    return x;
}
```

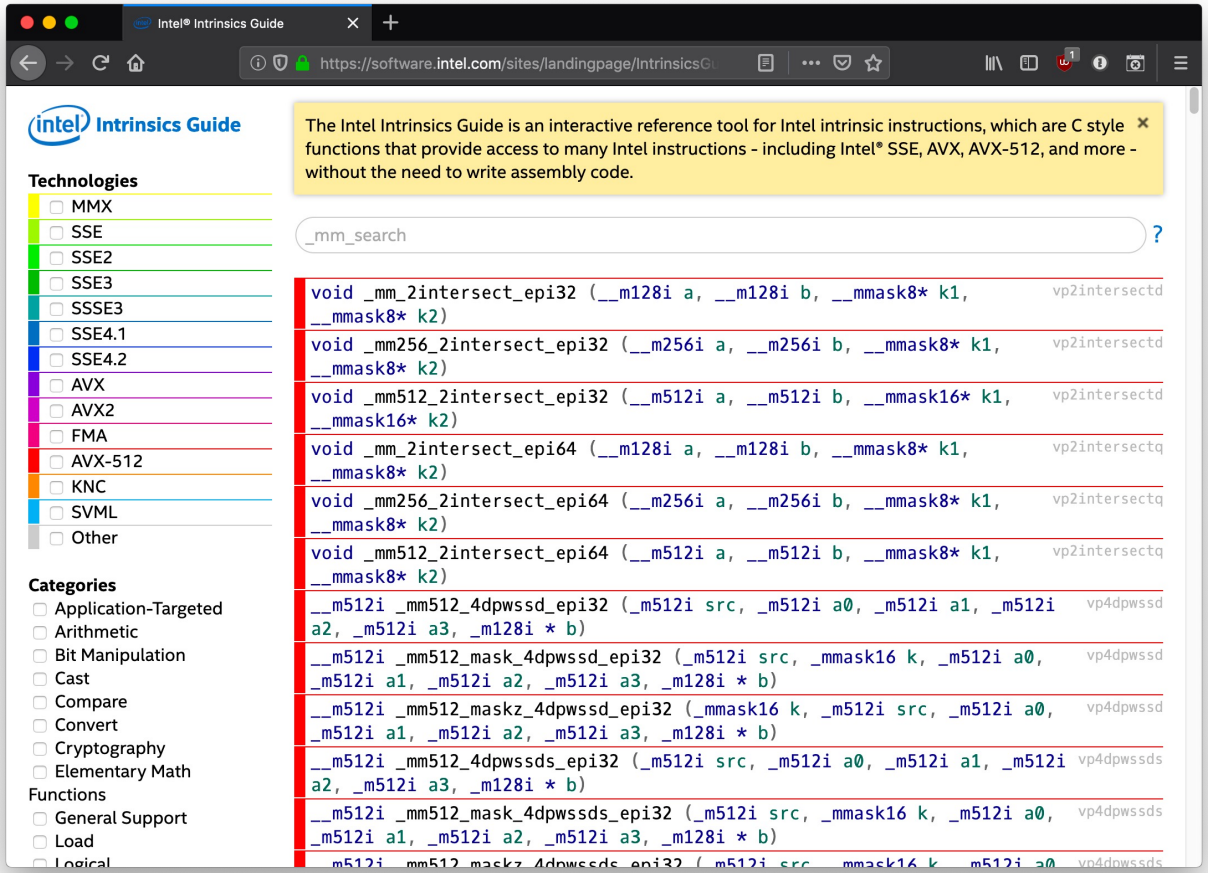
Processor intrinsics (e.g., on Intel Processors)

Example: `_mm512_mul_ps`

SIMD Width
(e.g., 512 bits = 16 floats)

Operation

Operand type



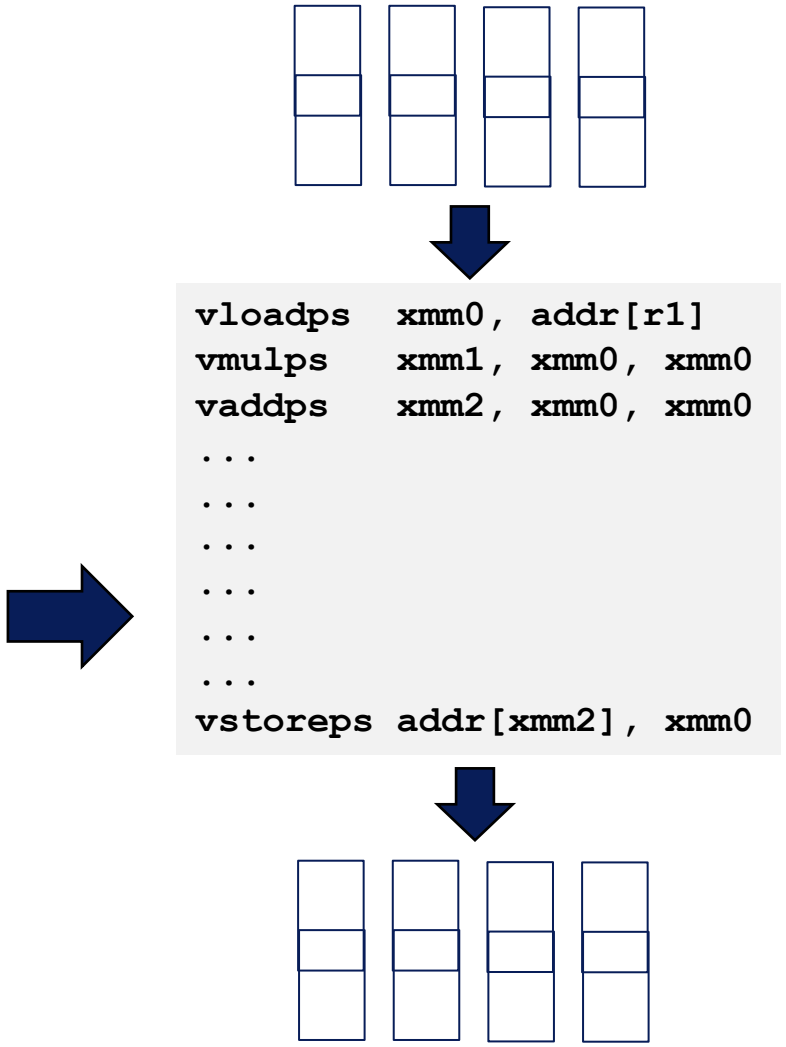
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Vector Program (Using AVX Instructions)

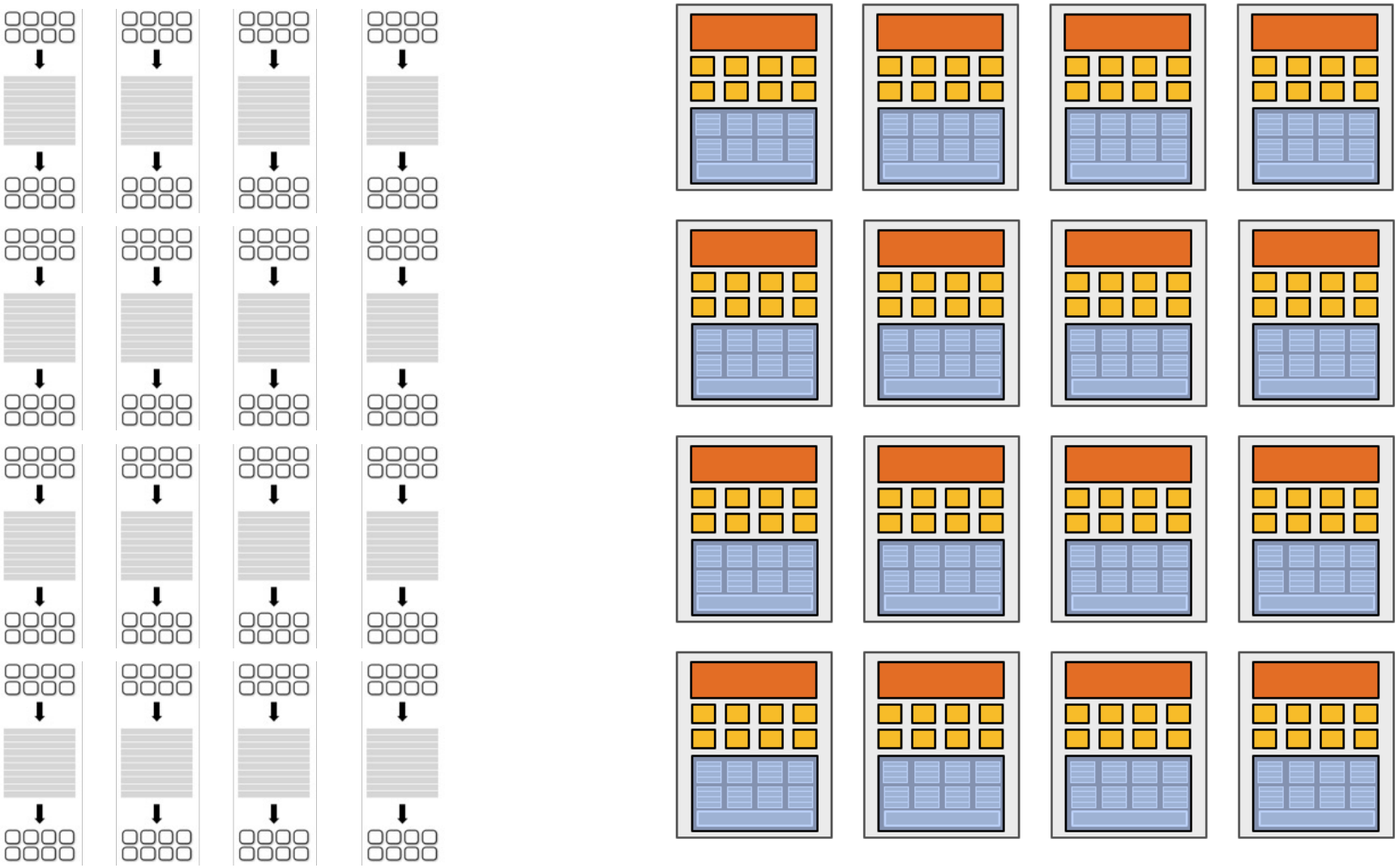
Our vector program:

- Processes 8 elements using vector instructions on 256-bit vector registers

```
#include <immintrin.h>

__m256 my_cbrt_vec(__m256 v) {
    __m256 x = _mm256_set1_ps(1.f);

    for (int j = 0; j < 10; ++j) {
        __m256 tmp = _mm256_add_ps(
         _mm256_add_ps(x, x),
         _mm256_div_ps(v, _mm256_mul_ps(x, x)));
        x = _mm256_mul_ps(
              _mm256_set1_ps(1.f / 3.f), tmp);
    }

    return x;
}
```



16 SIMD Cores: 128 Elements in Parallel



8 elements (256 bits) per SIMD core

SIMD Execution on Modern CPUs

Generated by compiler (or assembly programmer)

x86:

- SSE2 instructions: 128-bit operations (2x64, 4x32, 8x16, 16x8)
- AVX instructions: 256-bit operations (8x32 or 4x64 bits)
- NEW: AVX512 instructions: 512-bit operations (16x32 or 8x64 bits)

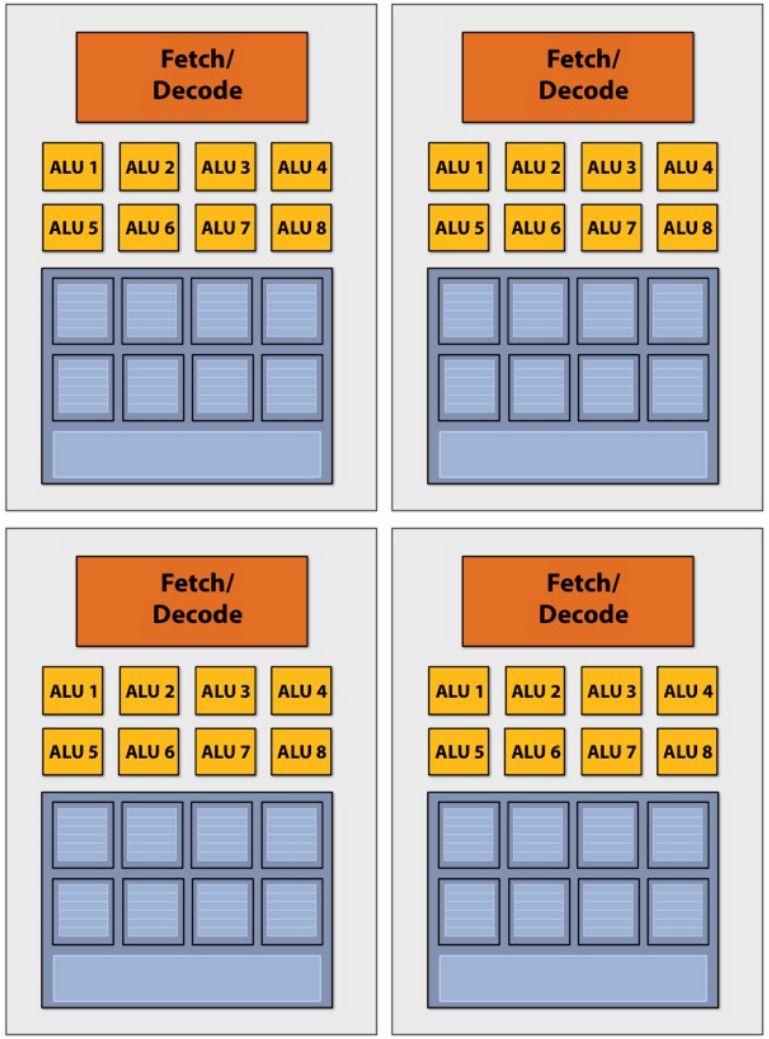
ARM:

- Neon instructions: 64-bit operations (2x32, 4x16, 8x8)

Parallelism:

- explicitly requested by programmer using intrinsics
- conveyed using parallel language semantics
- inferred by dependence analysis of loops (hard problem)

Example: Intel Core i7 (Sandy Bridge)



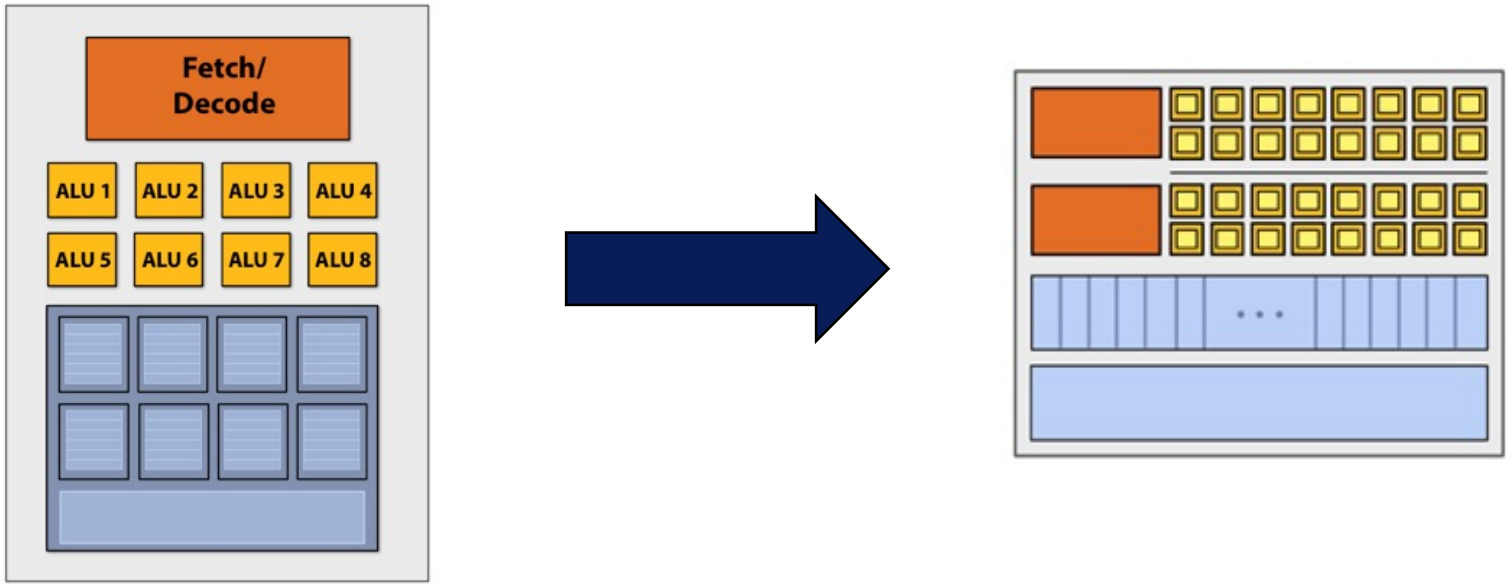
4 Cores
8 SIMD ALUs per core

GPUs: SIMD Execution at Massive Scale

How to scale SIMD to thousands of ALUs?

Use single instruction stream and no OoO execution

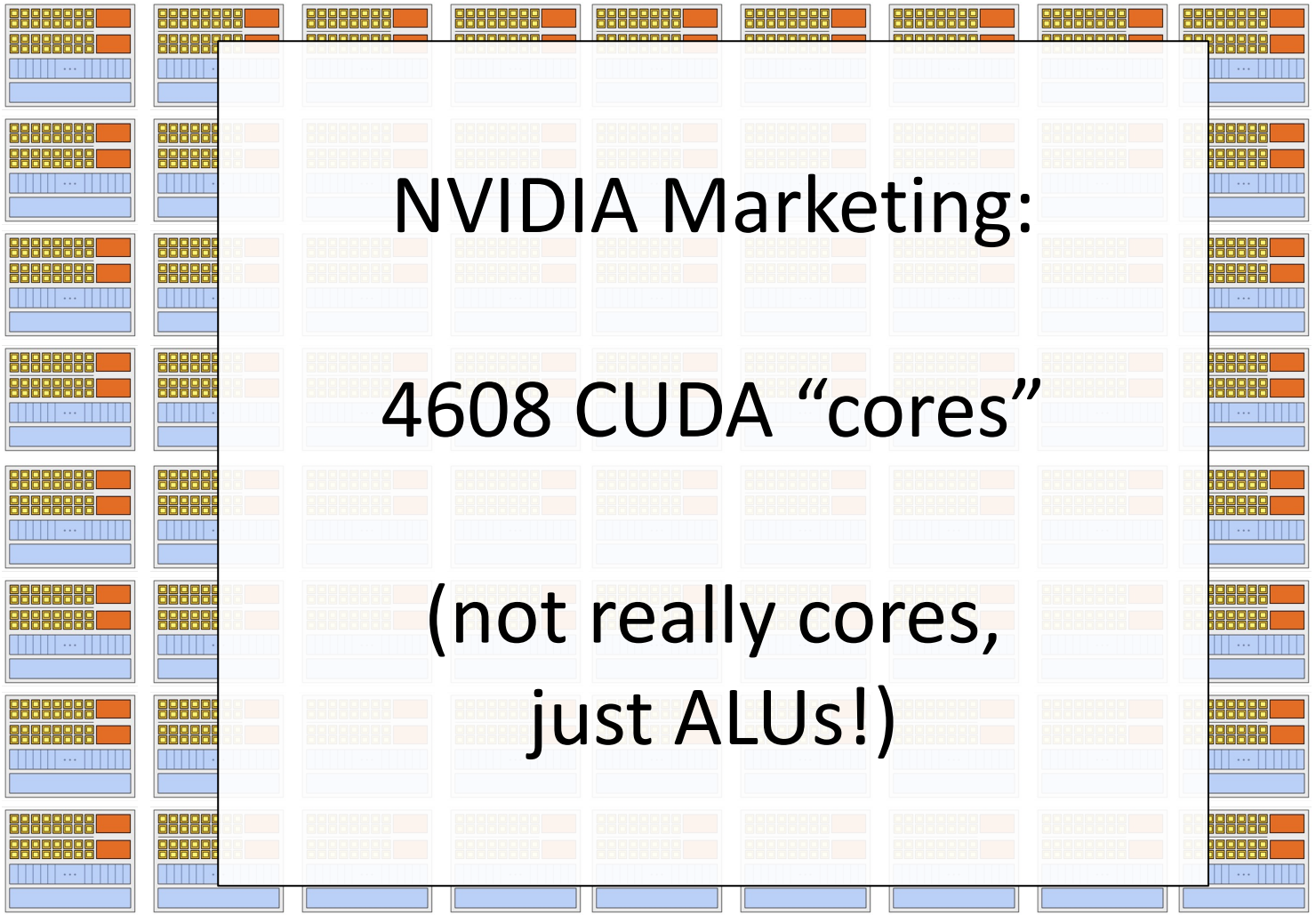
- One instruction controls hundreds of simultaneous ALUs



CPU Core

GPU Core

Example: NVIDIA TITAN RTX



72 Cores
32 ALUs
/core

Principles of Parallel Computing

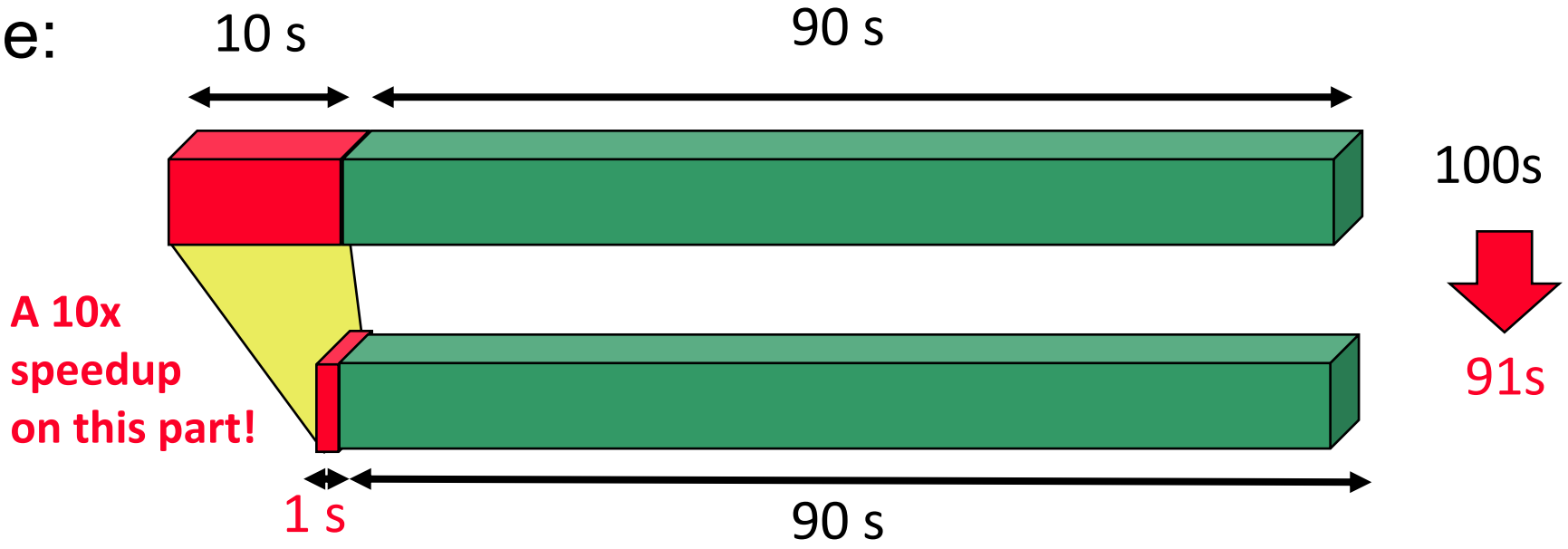
Principles of Parallel Computing

- Finding enough parallelism
- Division of work (granularity & load balancing)
- Scaling
- Communication & synchronization

Finding Enough Parallelism

- Amdahl's law
  - In English: if you speed up only a small fraction of the execution time of a computation, the speedup you achieve on the whole computation is limited!

- Example:



Amdahl's Law

$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Example:

Program runs for 100 seconds on a uniprocessor

50% of the program can be parallelized on a multiprocessor.

Assume a multiprocessor with 10 processors:

Amdahl's Law

$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Example:

Program runs for 100 seconds on a uniprocessor

50% of the program can be parallelized on a multiprocessor.

Assume a multiprocessor with 10 processors:

$$Speedup = \frac{1}{\frac{0.5}{10} + (1 - 0.5)} = \frac{1}{0.05 + 0.5} = \frac{1}{0.55} = 1.82$$

Amdahl's Law

$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Example:

Assume that 10% of the program cannot be parallelized. What is the maximum achievable speedup?

Amdahl's Law

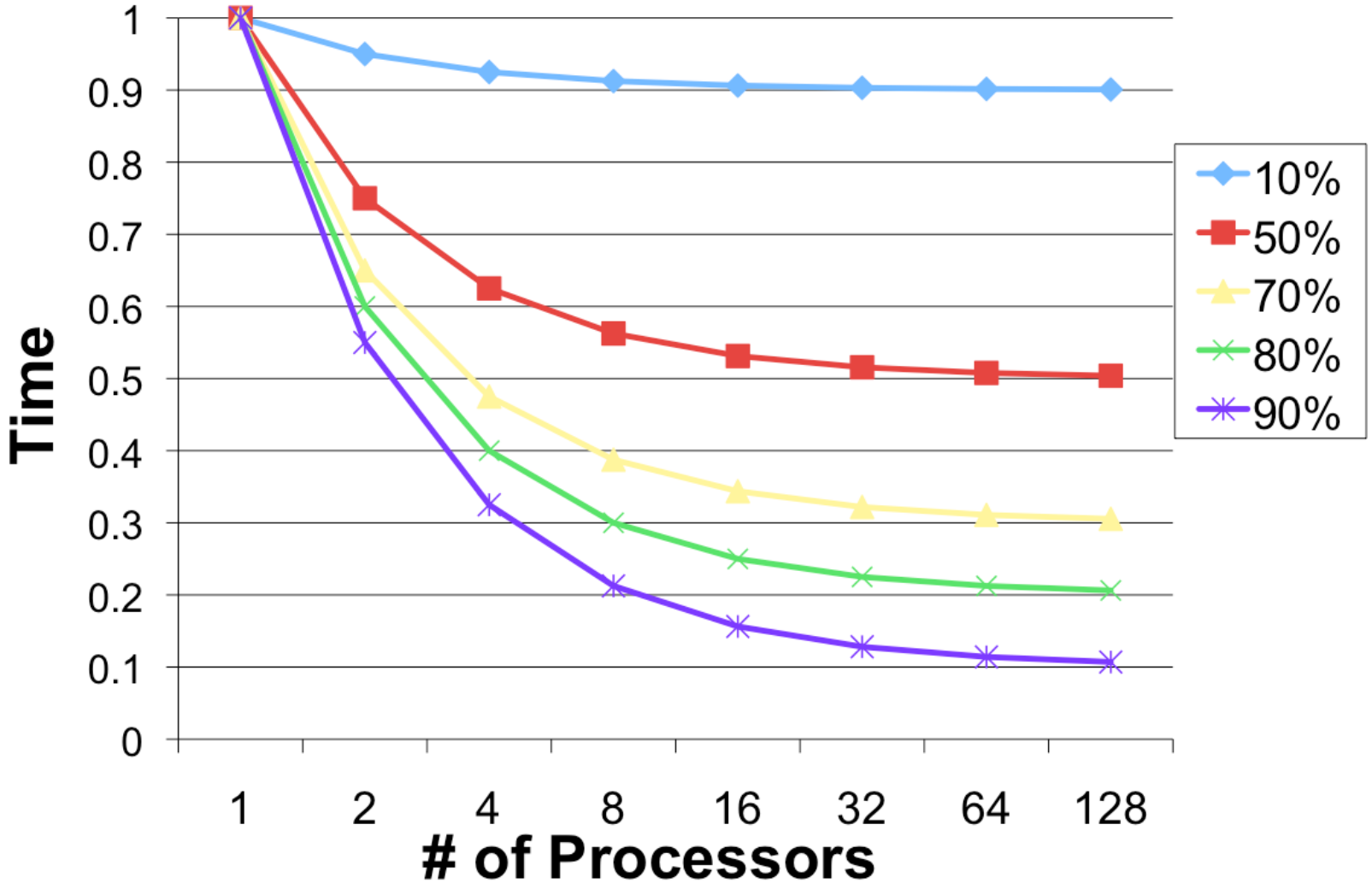
$$Speedup = \frac{1}{\frac{Fraction_{enhanced}}{Speedup_{enhanced}} + (1 - Fraction_{enhanced})}$$

Example:

Assume that 10% of the program cannot be parallelized. What is the maximum achievable speedup?

$$Speedup = \lim_{s \to \infty} \frac{1}{\frac{0.9}{s} + (1 - 0.9)} = \frac{1}{0 + 0.1} = 10$$

Implications of Amdahl's Law



Terminology

- A Task is a piece of work
  - Deep learning: one tensor product in the neural net
  - Social networks: one graph node in graph processing
- Task grain
  - small ➔ fewer instructions executed per task
  - large ➔ more instructions executed per task
- Process (thread) performs tasks
  - According to OS: process = thread(s) + address space
- Process (threads) executed on processor(s)

Division of Work: It's about Performance

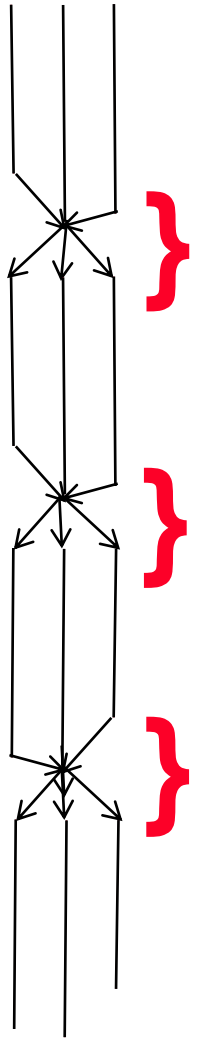
- Balance workload
  - Give each parallel task the same rough amount of work
- Reduce communication
  - Balance computation time with communication time
  - Computation ➔ useful work, Communication ➔ overhead
- Reduce extra work
  - Scheduling tasks on processors, OS, etc.
- These are at odds with each other

Division of Work: Granularity

- Granularity: ratio of computation to communication
- Fine-grained parallelism:
  - Less work between communication events
  - Better load imbalance
  - Favors faster communication
- Coarse-grained parallelism:
  - Less likely to suffer from system delays (e.g., $/TLB misses)
  - Potentially higher load imbalance
  - Can live with slow/bulk communication

Example: Division of Work

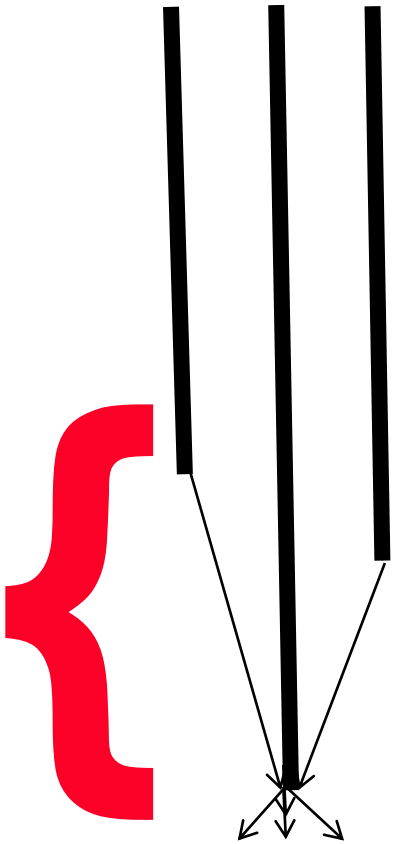
Small tasks



Frequent Communication
Frequent Scheduling

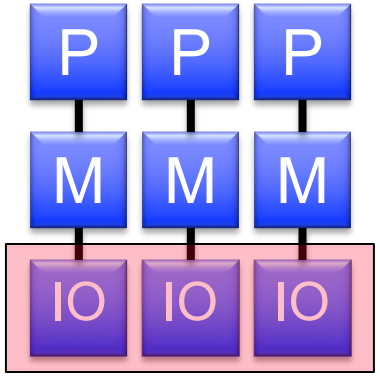
Load imbalance

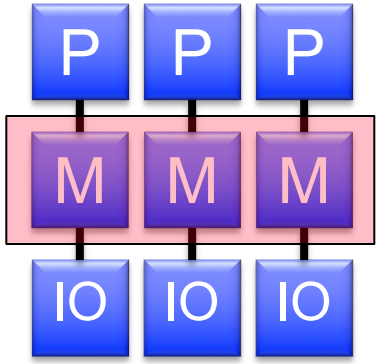
Large tasks

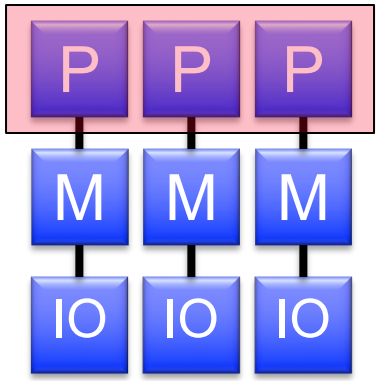


Take a Break!

Communication & Synchronization

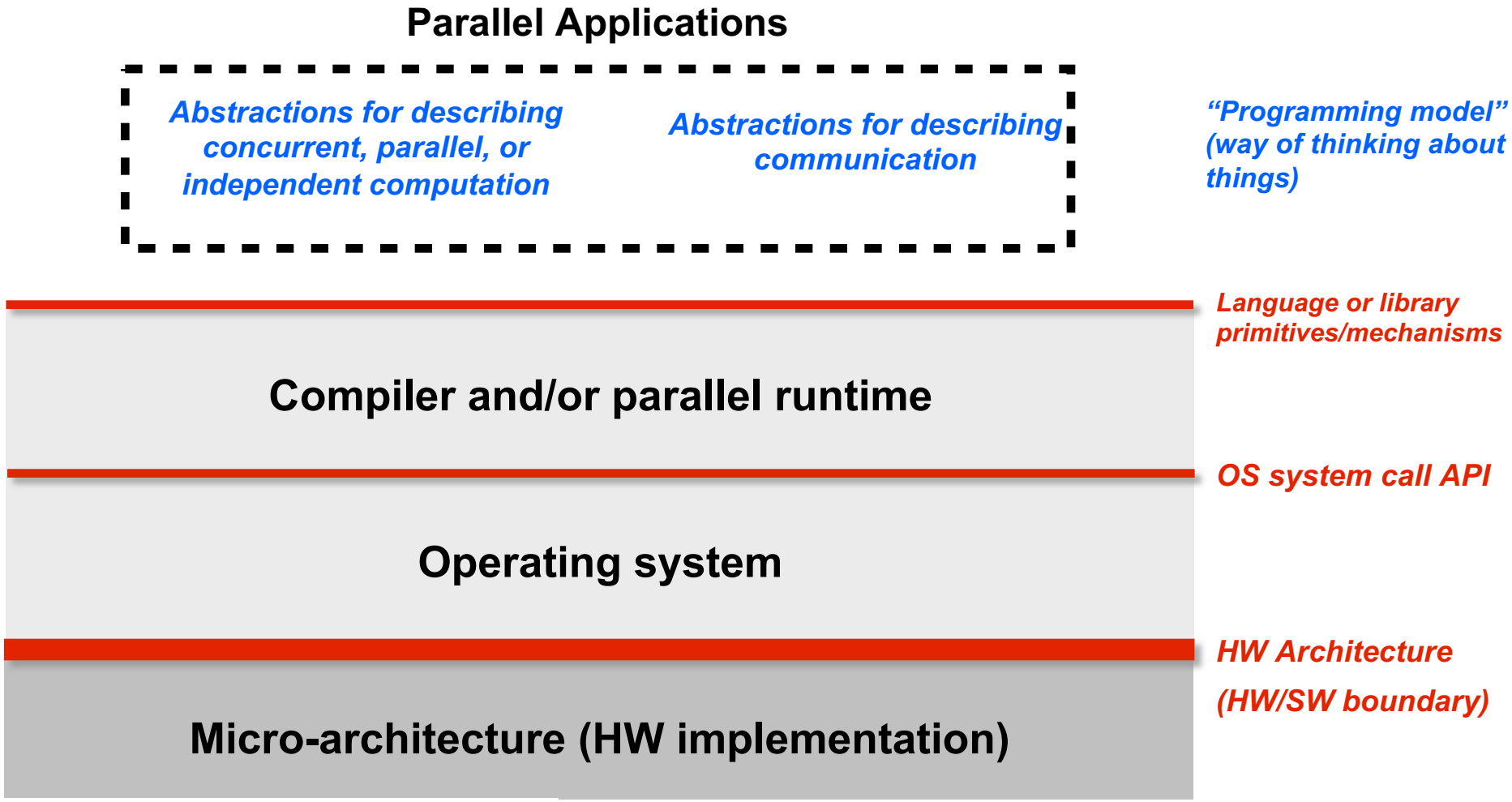






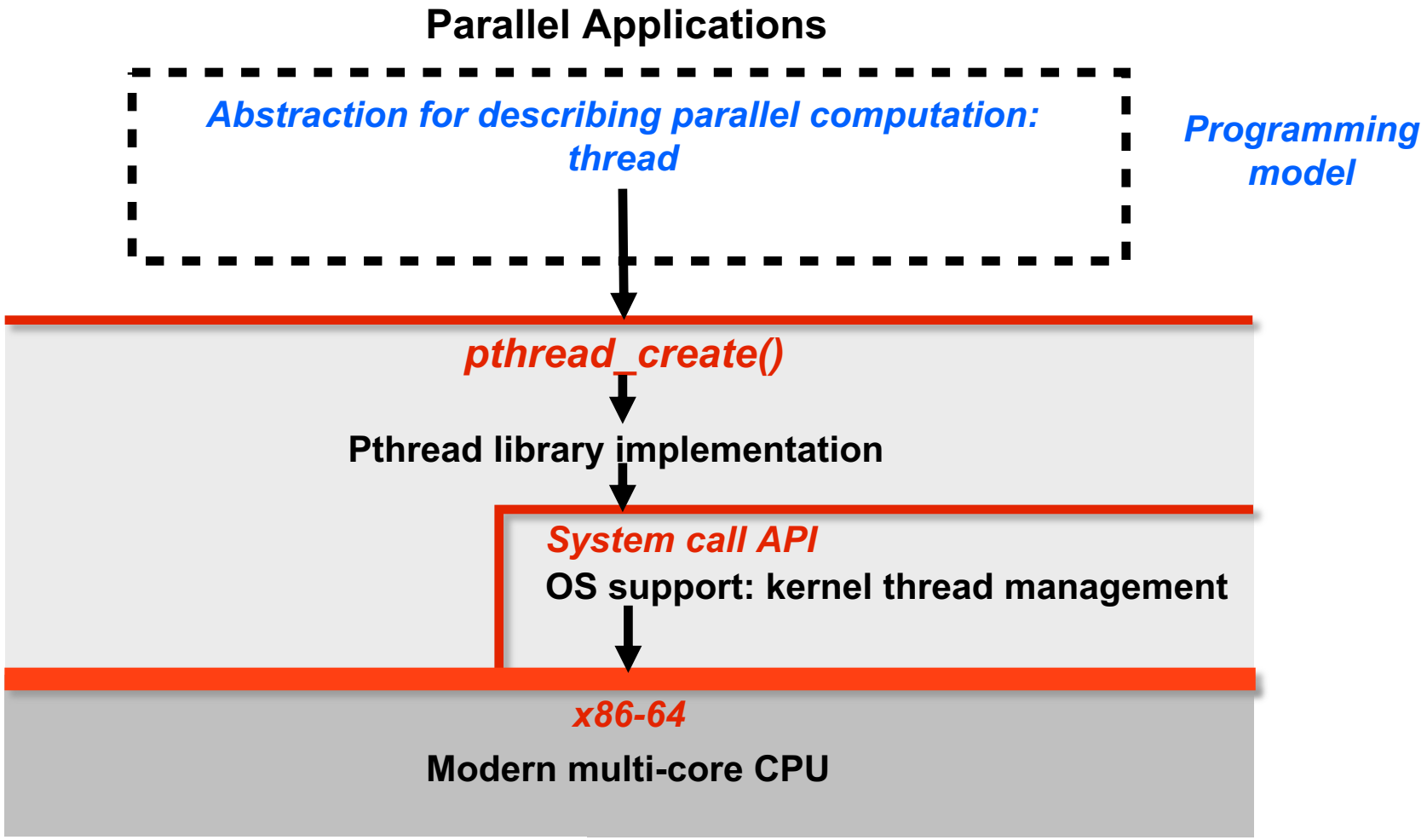
Sync at:	I/O	Memory	Processor
Model:	Message Passing (e.g., datacenter)	Shared Memory (e.g., CPU)	Dataflow (e.g., TPU)

System layers: Abstractions, Interfaces & Implementations



Blue italic text: abstraction/concept
Red italic text: system interface
Black text: system implementation

Example: Expressing Parallelism (Pthreads)



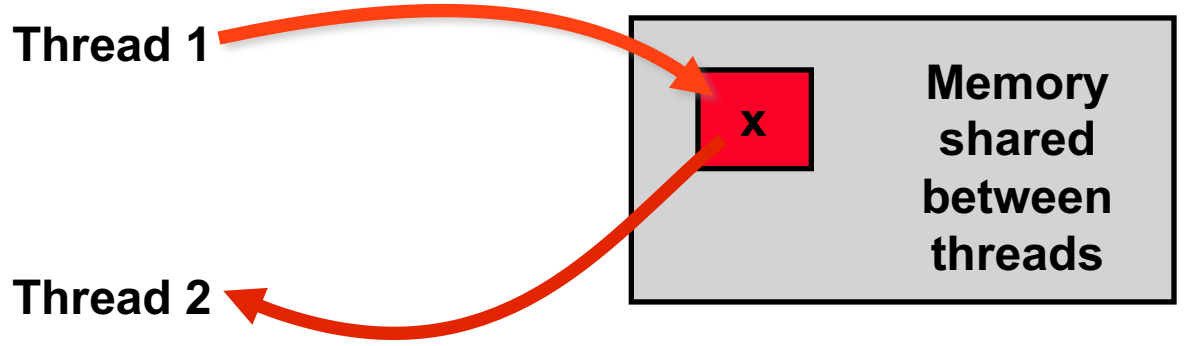
Blue italic text: abstraction/concept
Red italic text: system interface
Black text: system implementation

Three Models of Communication (Abstractions)

1. Shared address space
2. Message passing
3. Data parallel

Shared Address Space: Abstraction

- Threads communicate by reading/writing to shared vars
- Shared variables are like a big bulletin board
- Any thread can read or write



Thread 1:

```
int x = 0;
x = 1;
```

Thread 2:

```
int x;
while (x == 0) {}

print x;
```

Shared Address Space: Abstraction

Threads communicate by:

- Reading/writing to shared variables
  - Communication is implicit in memory operations
  - Thread 1 stores to X, later, thread 2 reads X (observes update)
- Manipulating synchronization primitives
  - e.g., mutual exclusion using locks

Natural extension of sequential programming model

- In fact, all our discussions have assumed a shared address space so far

Shared Address Space: Implementation

Option 1: threads share an address space

- All data is sharable

Option 2: each thread has its own virtual address space

- Shared part maps to the same physical location

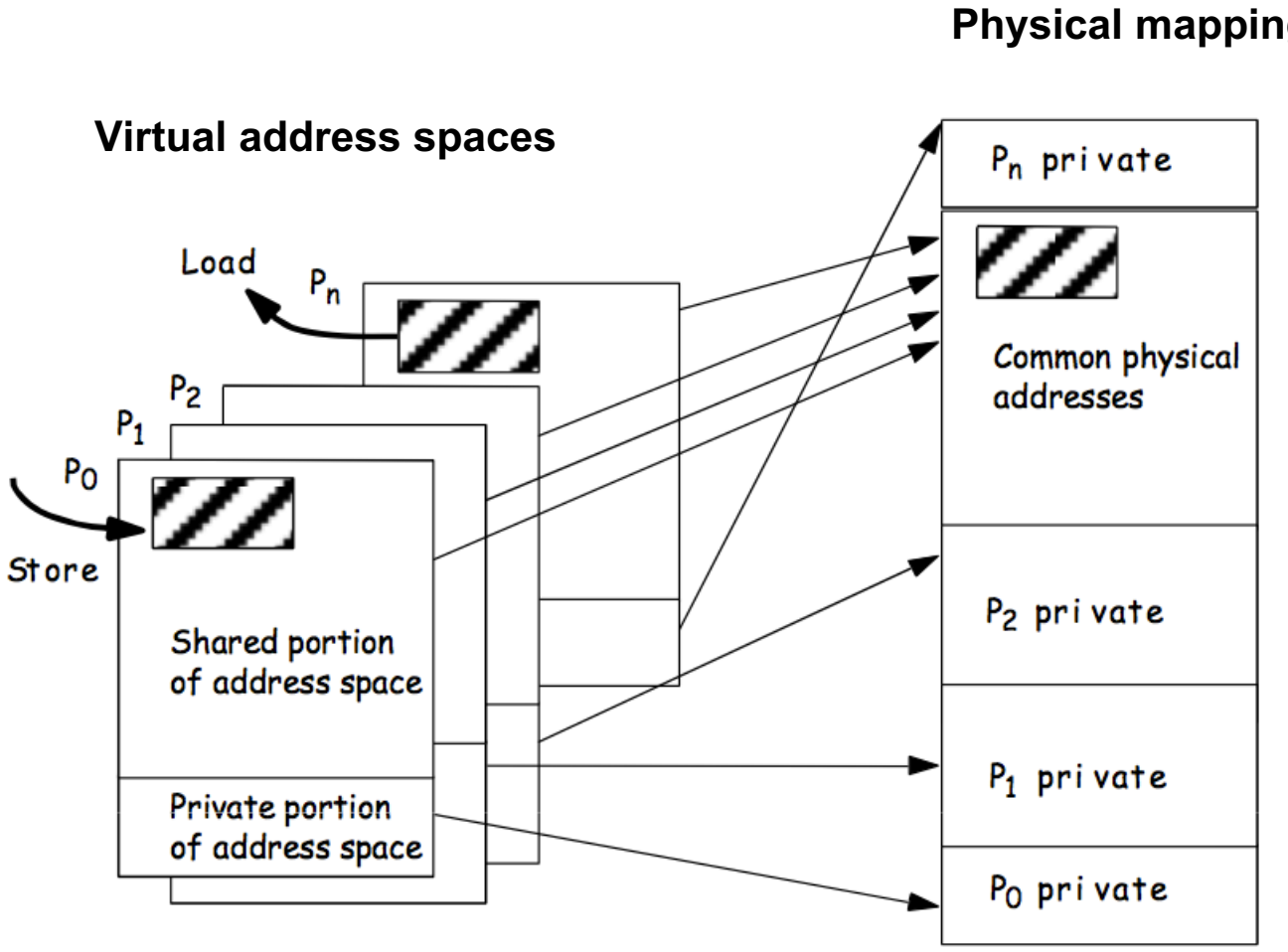
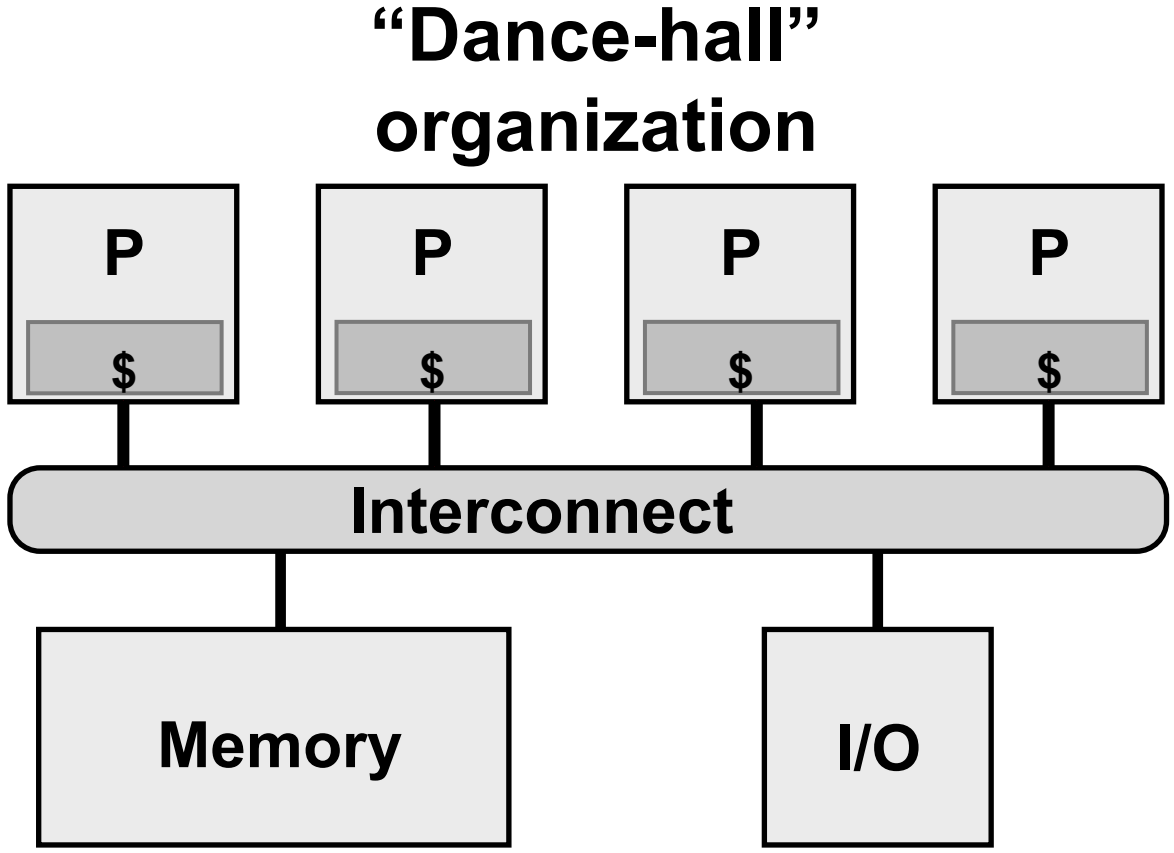
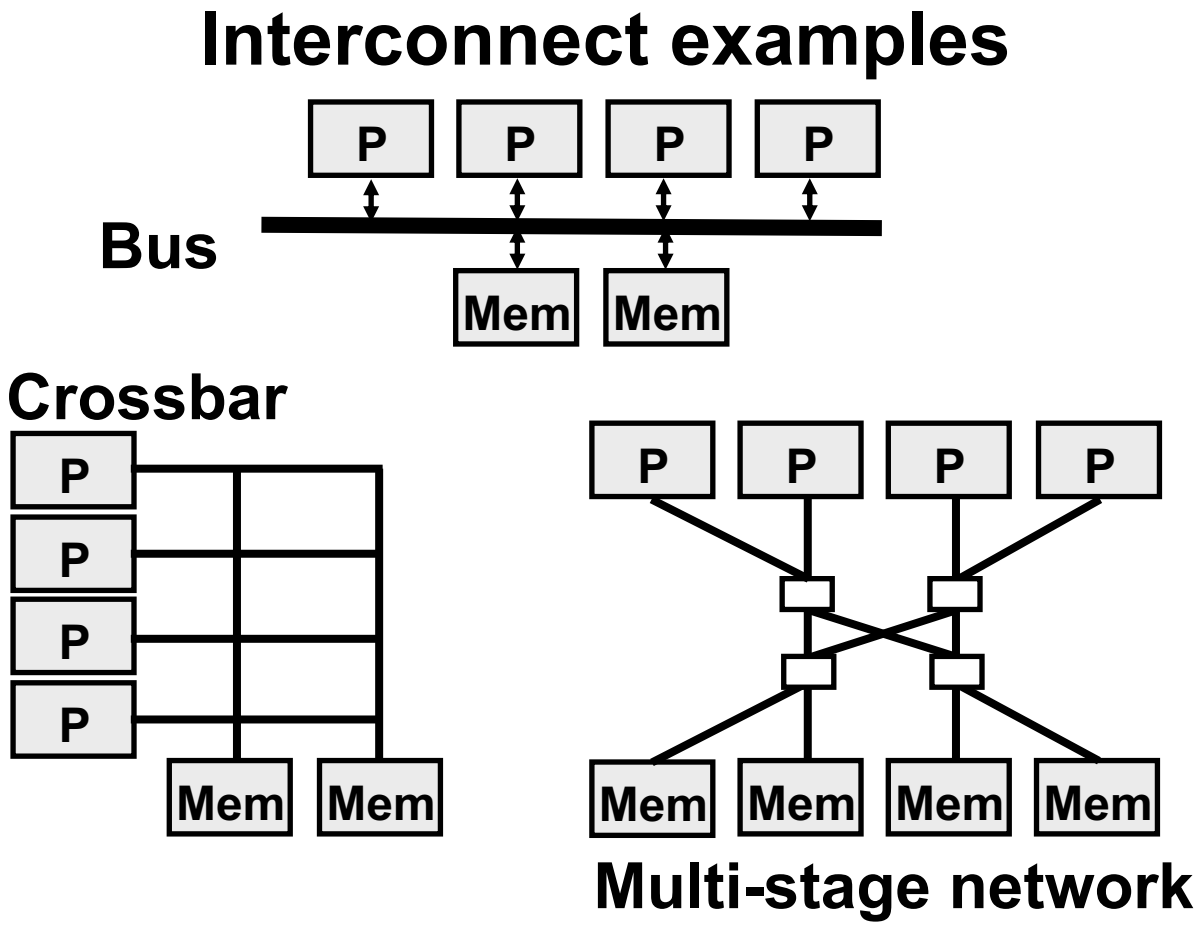


Image credit: Culler, Singh, and Gupta

Shared Address Space: HW Implementation

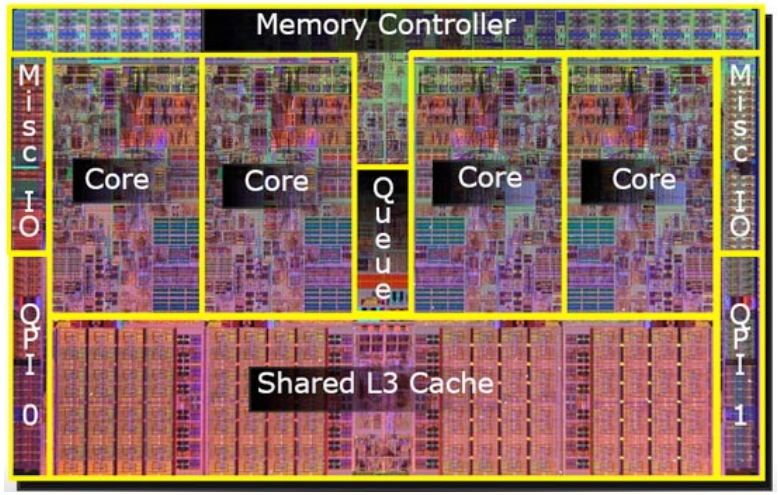
Any processor can directly reference any memory location



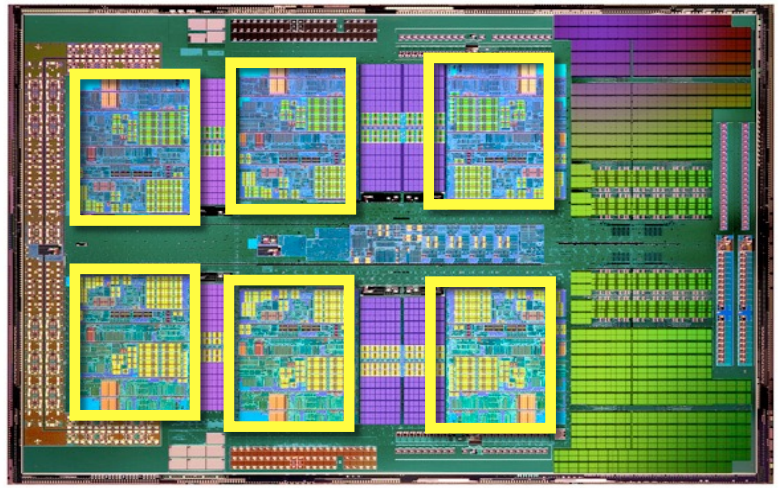


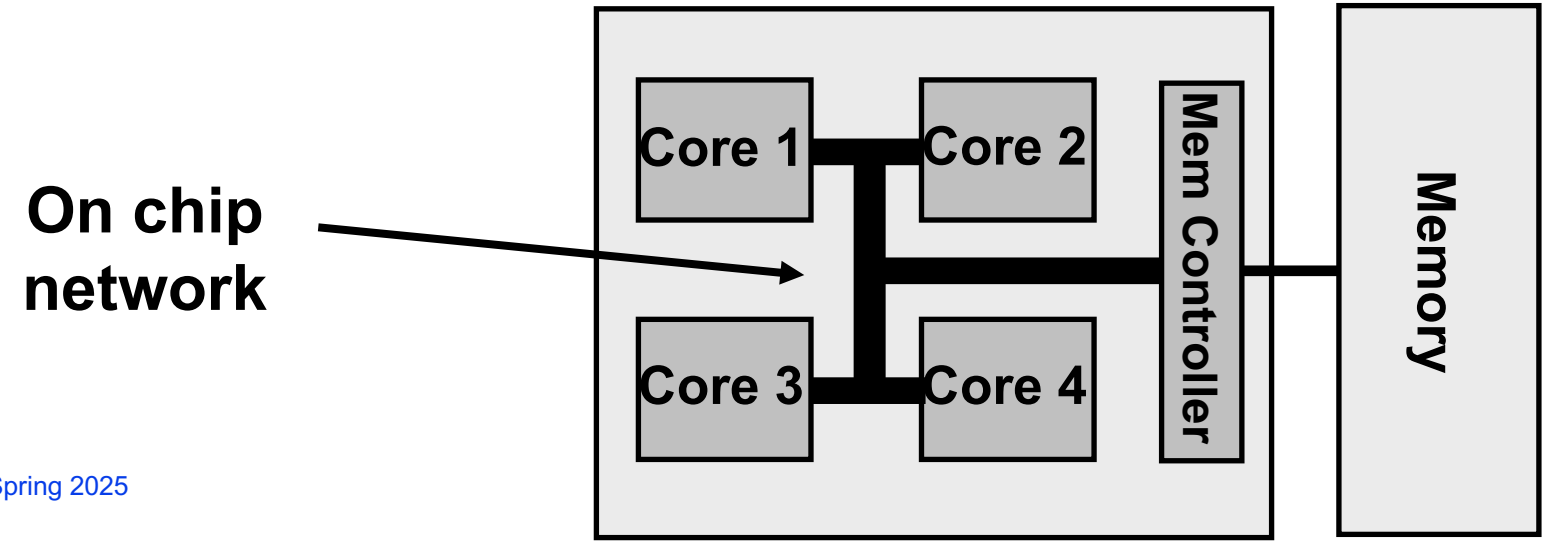
Shared Address Space Architectures: x86 examples

Intel Core i7 (quad core)



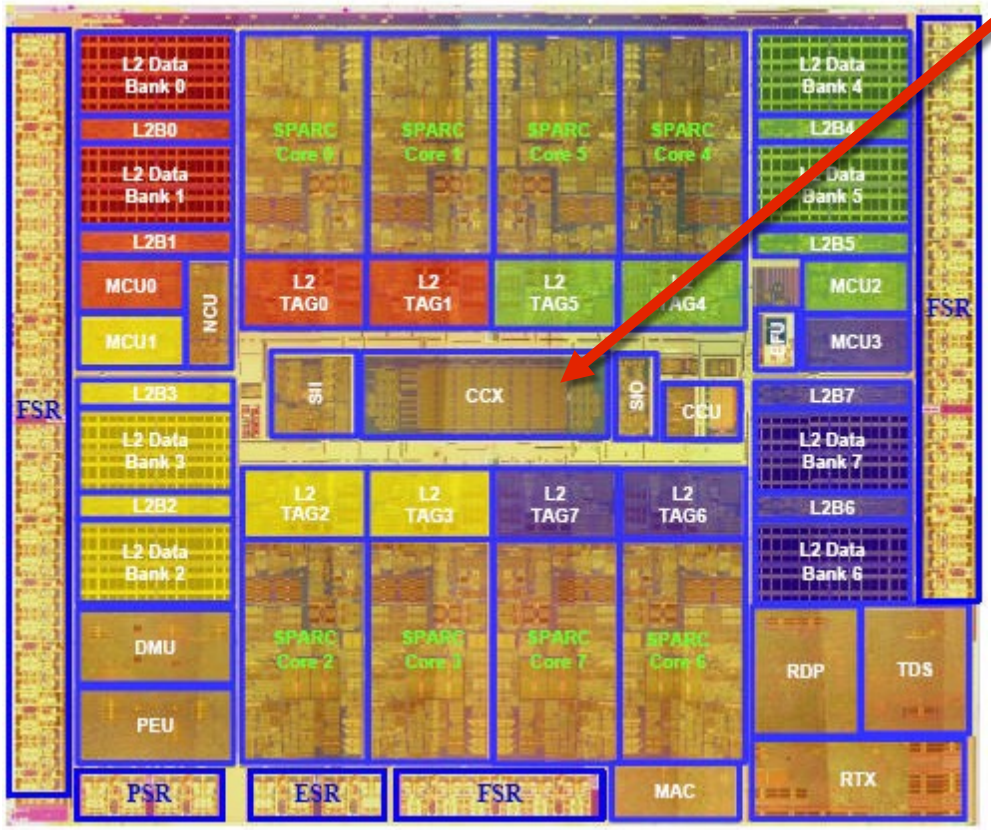
AMD Phenom II (six core)



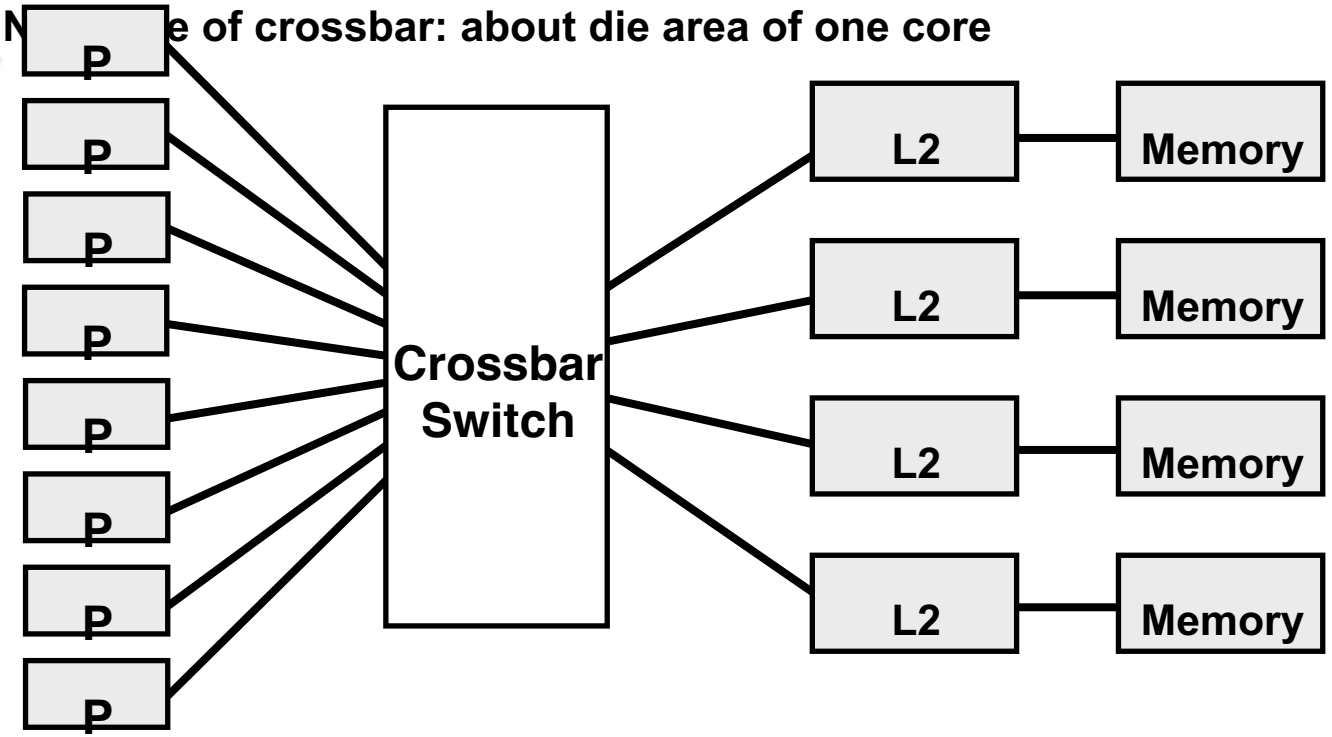


Shared Address Space Architectures: SPARC example

Sun (Oracle) Niagara 2

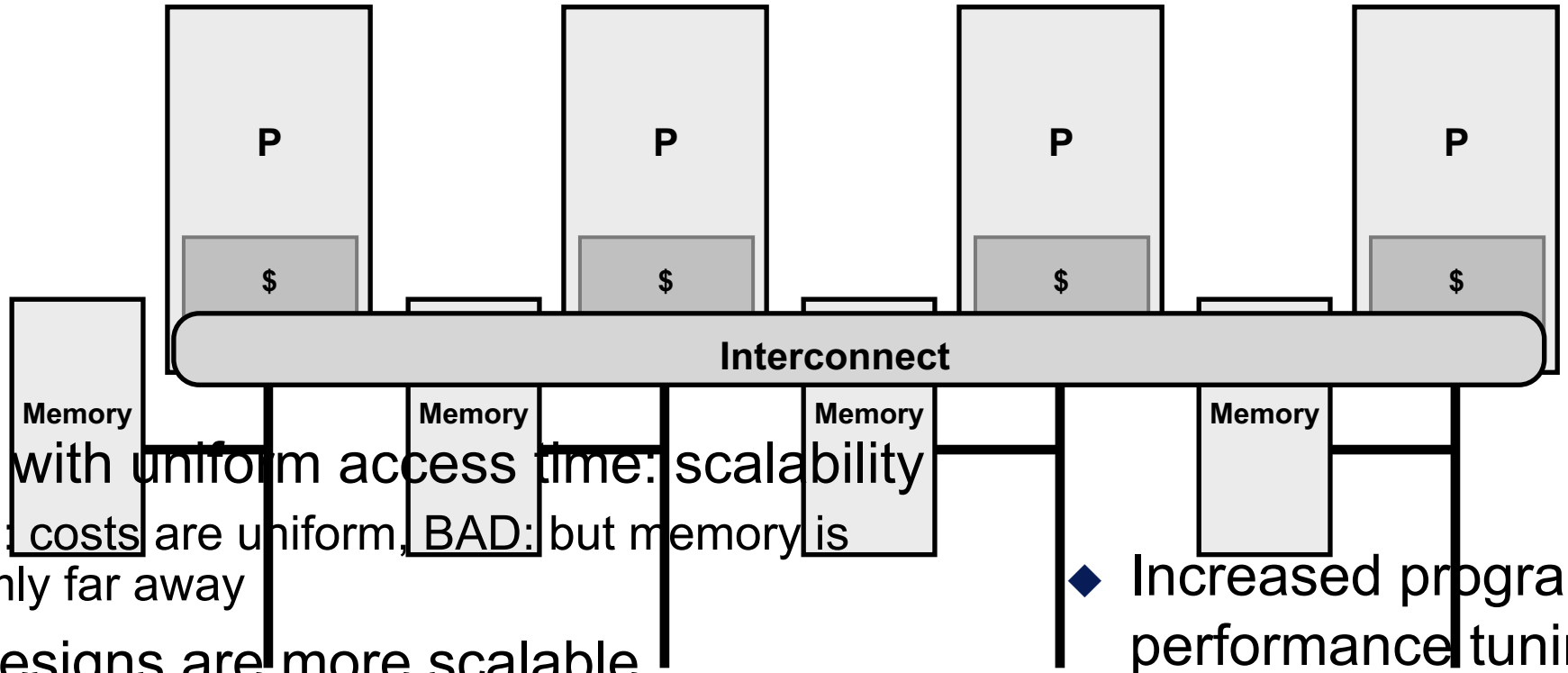


Eight cores



Non-uniform Memory Access (NUMA)

All processors can access any memory location, but... cost of memory access is different for different processors



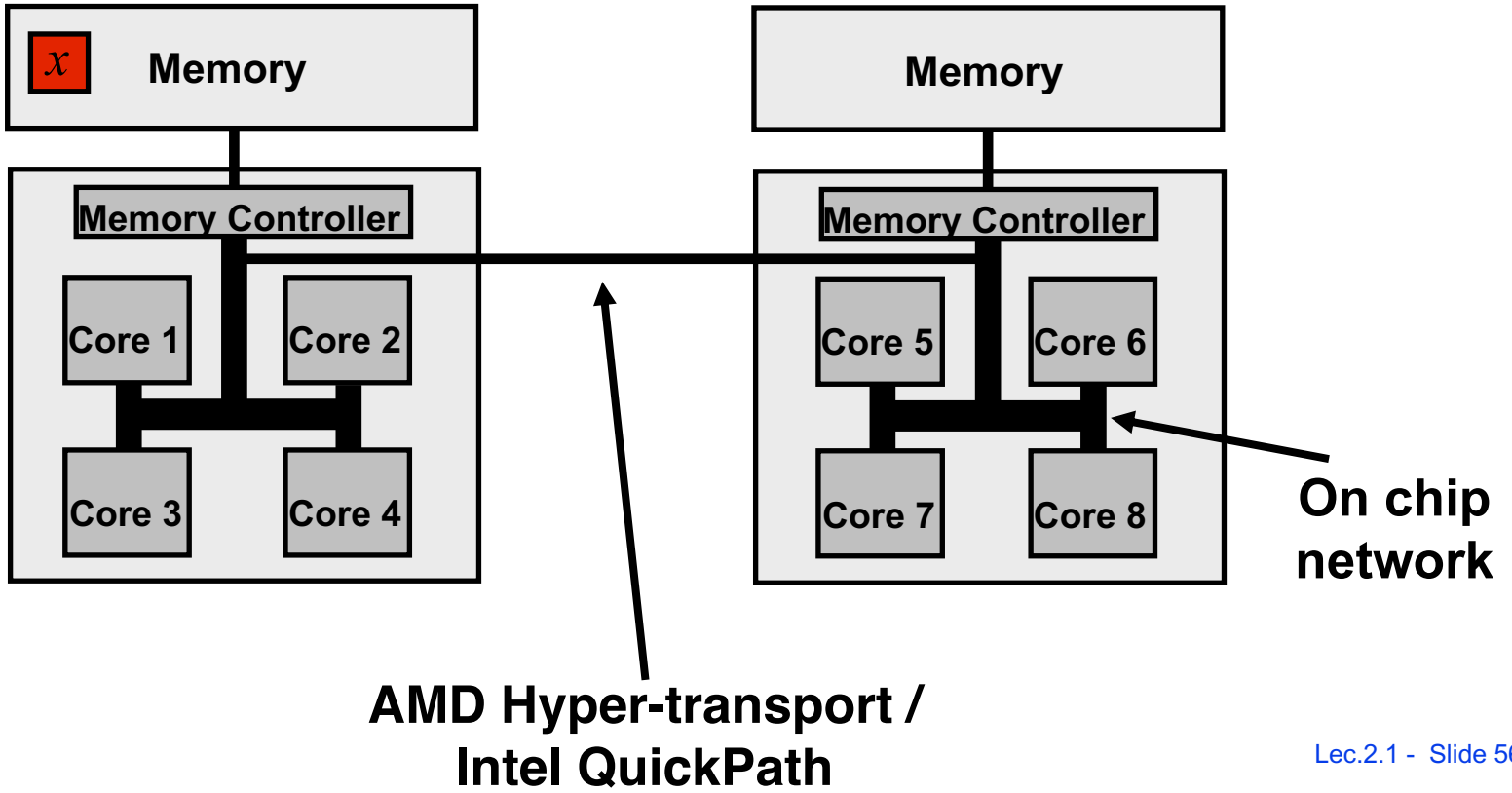
- Problem with uniform access time: scalability
  - GOOD: costs are uniform, BAD: but memory is uniformly far away
- NUMA designs are more scalable
  - High bandwidth & low latency access to local memory
  - BW scales with # of nodes if most accesses are local
- Increased programmer effort: performance tuning
  - Finding, exploiting locality

Non-uniform Memory Access (NUMA)

Example: modern dual-socket configuration

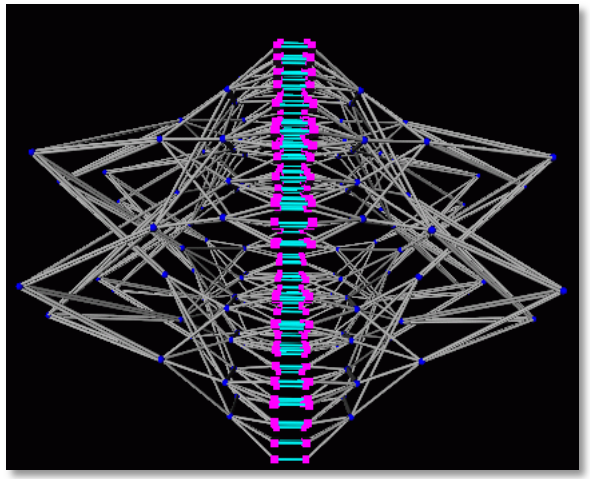
Latency to access location *x* from cores 5-8 is higher than cores 1-4!

E.g., Microsoft's
Open CloudServer



SGI Altix UV 1000

- 256 blades, 2 CPUs per blade, 8 cores per CPU = 4096 cores
- Single shared address space
- Interconnect: fat tree



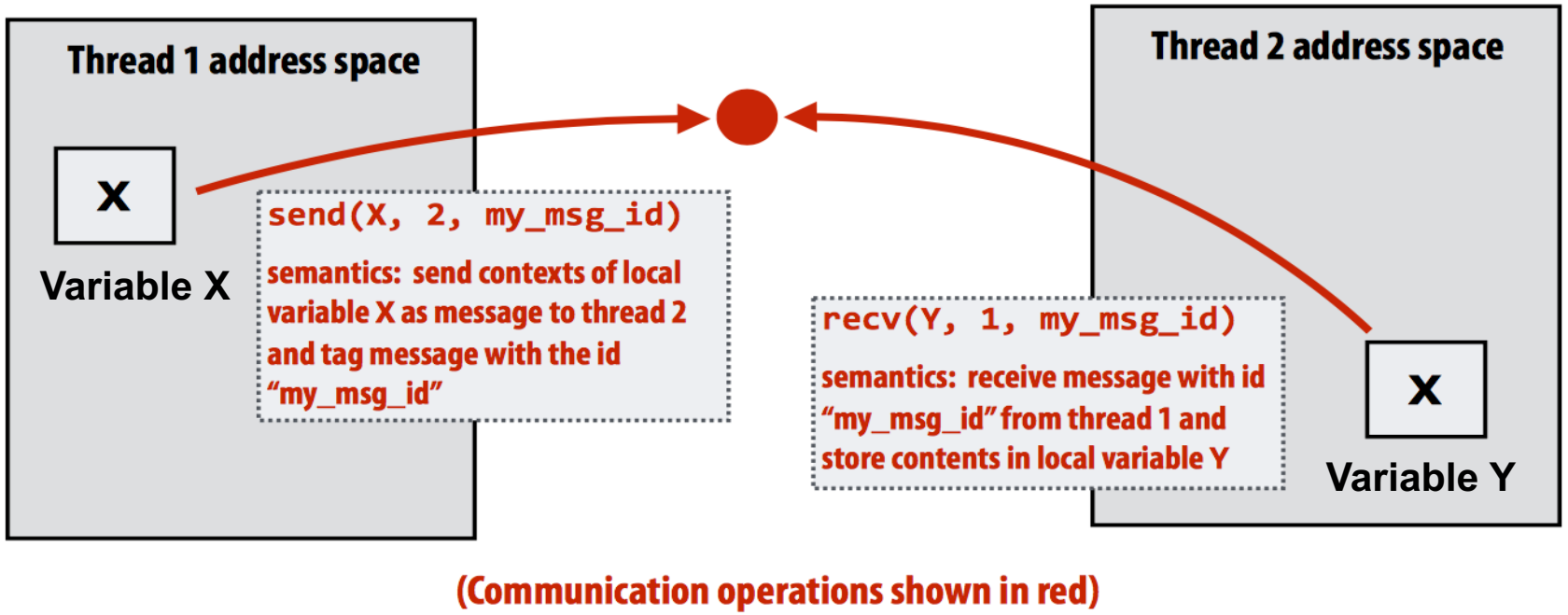
Fat tree

Image credit: Pittsburgh Supercomputing Center

Communication Model: Message Passing

Message Passing

- Different address spaces
- Communicate explicitly via sending/receiving messages
- Synchronization is implicit in the messages
- Arguably harder to program, but easier to scale



(Communication operations shown in red)

Message Passing Implementation (Swiss Supercomputer)

- Popular software library: MPI (message passing interface)
- System-wide LD/ST interface is expensive at massive scale
  - Connect commodity systems together to form large parallel machine
- E.g., Swiss “ALPS” comprised of ~3.7K server nodes
  - 2.7K hybrid CPU/GPU NVIDIA nodes
  - 1K AMD CPU nodes
  - Using MPI, can program all as one

Programming Models vs Machine Types

- Correspondence between programming models and machine types is fuzzy
- Common to implement message passing abstractions on machines that support a shared address space in HW
- Implement shared address space on machines that do not support it in HW
  - Mark all pages with shared variables as invalid
  - Page-fault handler issues appropriate network requests
- Keep in mind what is the programming model (abstractions used to specific program) and what is the HW implementation

Communication Model: Data Parallel

- Rigid computation structure
- Same function on all data elements
- Historically: same operation on each element of an array
  - E.g., Cray vector supercomputers of the 80's
  - add(A, B, n)← one inst. on vectors A, B of length n
- Now, functions can be arbitrarily large
  - Communication implied after the function terminates
- Today platforms can range from
  - SIMD: One instruction multiple data (in a single core)
  - SPMD: One program multiple data in a cluster

Data Parallel Example

- You have already seen this in our first example:
  - No order implied between parallel iterations
  - All threads converge at the termination of the parallel for

```
void my_cbrt(float *v, float *result, int N) {
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        float x = 1.f;
        for (int j = 0; j < 10; ++j)
            x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));
        result[i] = x;
    }
}
```

Communication Models Summary

- Shared Memory (aka Shared Address Space)
  - Communication is unstructured, implicit in loads and stores
  - Arguably easier to program, but harder to scale
- Message Passing
  - Structure all communication as messages
  - Arguably harder to program, but easier to scale
- Data Parallel
  - Structure computation as a big “map” over a collection
  - Severely limits communication between iterations of the map (goal: preserve independent processing of iterations)

Shared Memory Parallel Programming Using OpenMP

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - Standard API for defining multithreaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A Programmer's View of OpenMP

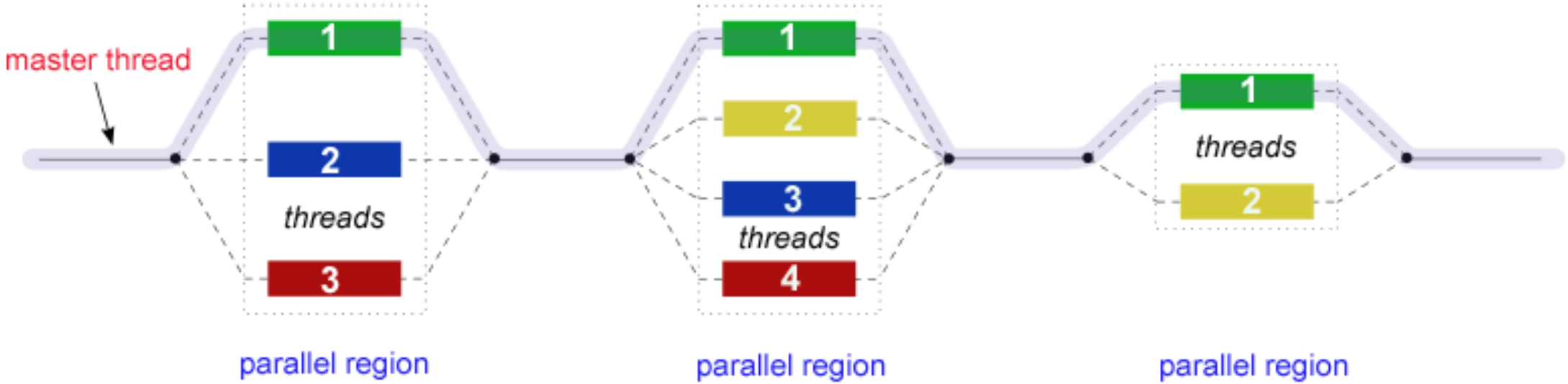
- Portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to divide a program into serial & parallel regions
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

OpenMP Execution Model

- Fork/join model
  - Initially only the master thread is active
    - Master thread executes until a parallel region is encountered
  - Fork: master thread creates a team of parallel threads
    - Statements in parallel region are executed in parallel
  - Join: threads sync & terminate at the end of parallel region
    - Master thread continues executing sequentially



Hello World!

- Simple “Hello World!” example:

```
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

Hello World!

- Simple “Hello World!” example:

```
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

- Parallelized using OpenMP:

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    // Do this part in parallel
    #pragma omp parallel
    {
       printf( "Hello, World!\n" );
    }
    return 0;
}
```

Hello World!

- Simple “Hello World!” example:

```
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

- Parallelized using OpenMP:

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    // Do this part in parallel
    #pragma omp parallel
    {
       printf( "Hello, World!\n" );
    }
    return 0;
}
```

Hello World!

- Simple “Hello World!” example:

```
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

- Parallelized using OpenMP:

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    // Do this part in parallel
    #pragma omp parallel
    {
       printf( "Hello, World!\n" );
    }
    return 0;
}
```

Hello World!

- Simple “Hello World!” example:

```
int main() {
    printf( "Hello, World!\n" );
    return 0;
}
```

- Parallelized using OpenMP:

```
#include <omp.h>
int main() {
    omp_set_num_threads(4);
    // Do this part in parallel
    #pragma omp parallel
    {
       printf( "Hello, World!\n" );
    }
    return 0;
}
```

Parallelizing `for` Loops

OpenMP easily parallelizes `for` loops

- Race free (reads/write or write/write pairs) between iterations!
- for loop must have canonical shape
  - No break, return, exit or goto statements

Preprocessor calculates loop bounds for each thread from *serial* source

```
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
  printf("Hello World!");
}
```

Canonical Shape

$$\text{for}(\text{index} = \textit{start};\ \text{index} \geq \left\{ \begin{array}{l} < \\ <= \\ >= \\ > \end{array} \right\} \textit{end};\ \left\{ \begin{array}{c} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} \mathrel{+}= \textit{inc} \\ \text{index} \mathrel{-}= \textit{inc} \\ \text{index} = \text{index} + \textit{inc} \\ \text{index} = \textit{inc} + \text{index} \\ \text{index} = \text{index} - \textit{inc} \end{array} \right\})$$

Recall Our Earlier Example

Loop iterations are independent of one another

- Compiler can automatically generate parallel threaded code

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    float x = 1.f;

    for (int j = 0; j < 10; ++j)
        x = (1.f / 3.f) * (2.f*x + v[i] / (x*x));

    result[i] = x;
}
```

Division of Work – Controlling Granularity

`#pragma omp parallel if (expression)`

- Can be used to disable parallelization some cases
- E.g., if the input size is too small to be beneficially multithreaded

```
#pragma omp parallel for if (n > 5000)
for( i=0; i < n; i++ ) {
  printf("Hello World!");
}
```

`#pragma omp num_threads (expression)`

- Control the number of threads used for this parallel region

Division of Work – Load Balancing

Schedule clause determines how loop iterations are divided among a thread team:
`schedule(<type>[,<chunk> ])`

- static([chunk]) divides iterations statically between threads
  - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
  - Default [chunk] is ceil(# iterations / # threads)

- dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
  - Forms a logical work queue, consisting of all loop iterations
  - Default [chunk] is 1

- guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation

Execution Context (EC)

- Every thread has its own execution context
  - Address space containing all the variables a thread may access

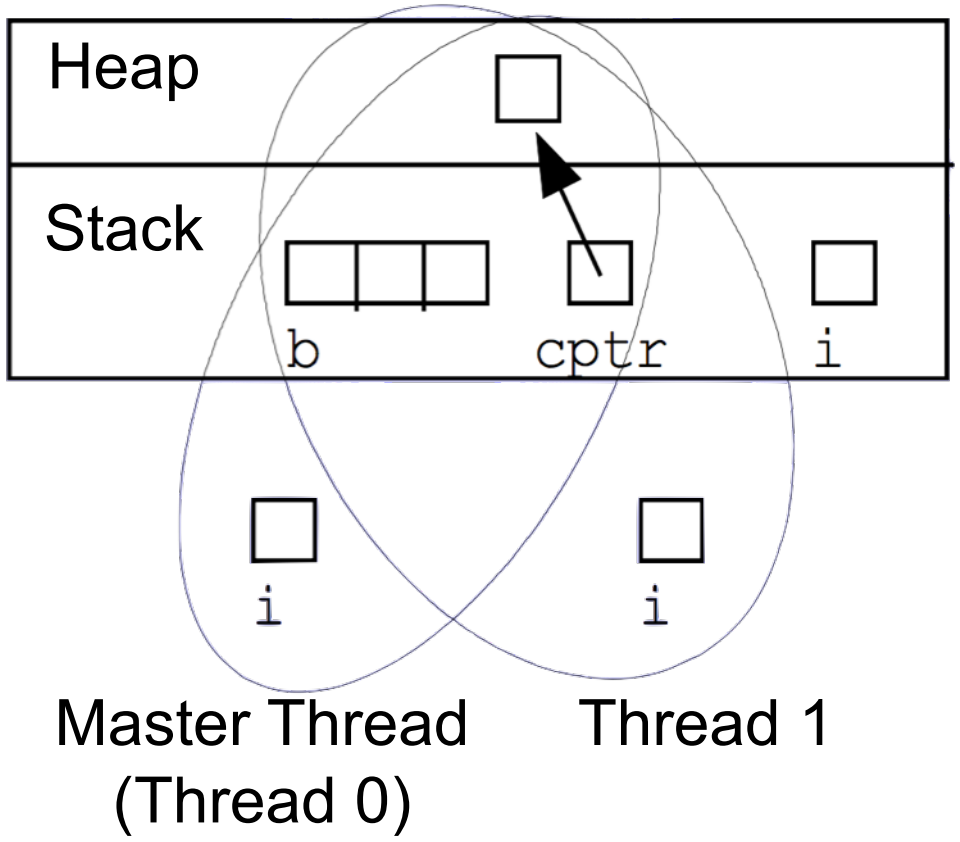
- Contents of the execution context:
  - Static variables
  - Dynamically allocated data structures in the heap
  - Variables on the run-time stack
  - Additional run-time stack for functions invoked by the thread

Shared vs. Private Variables

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- Shared *var*: has same address in EC of every thread
- Private *var*: has different address in EC of every thread
- A thread cannot access the private vars of another thread
- In parallel region, all *vars* are *shared* by default
- Private *vars* must be declared in the pragma statement
  - private (<variable list>)

Example: Shared vs. Private Variables

```
int b[3];
char *cptr;
int i;

cptr = malloc(1);

#pragma omp parallel for
for (i=0; i<3; i++)
  b[i] = i;
```



Fancier Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

More Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

More Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

Check how many processors are available

More Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

Set the number of threads to 4

More Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

Get my thread ID

More Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

Fork a team of threads giving them their own copies of variables

More Sophisticated Hello World!

Results from running on a 8-core machine:

```
There are 8 processors
Hello World from thread = 0
Hello World from thread = 3
Hello World from thread = 1
Number of threads = 4 and my tid = 0
Hello World from thread = 2
```

Synchronization

- Critical Section
  - A portion of code that only one thread at a time may execute

```
#pragma omp critical
{
  /* Critical code here */
}
```

- Atomic Execution
  - Protects a single variable update

```
#pragma omp atomic
/* Update statement here */
```

Synchronization

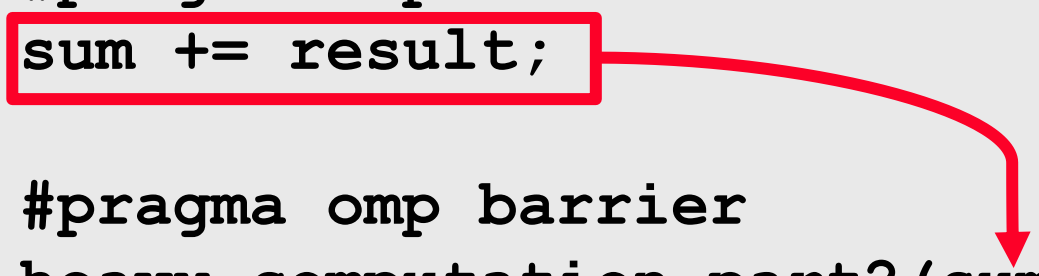
- Barrier
  - Performs a barrier synchronization among all threads in a team at a given point
  - All threads wait at the barrier point
  - Continue when all threads have reached the barrier point

```
#pragma omp parallel {
   int result = heavy_computation_part1();

   #pragma omp atomic
   sum += result;

   #pragma omp barrier
   heavy_computation_part2(sum);
}
```

Synchronization

- Barrier
  - Performs a barrier synchronization among all threads in a team at a given point
  - All threads wait at the barrier point
  - Continue when all threads have reached the barrier point

```
#pragma omp parallel {
    int result = heavy_computation_part1();

    #pragma omp atomic
    sum += result;

    #pragma omp barrier
    heavy_computation_part2(sum);
}
```



Synchronization

- Single-threaded region within a parallel region
  - master and single directives

```
#pragma omp single
{
  /* Only executed once */
}
```

```
#pragma omp master
{
  /* Only executed by master*/
}
```

Our Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }

  } /* All threads join master thread and terminate */
  return 0;
}
```

Our Sophisticated Hello World!

```
#include <omp.h>
int main () {
  int nthreads, tid;
  printf("There are %d processors\n", omp_get_num_procs());

  omp_set_num_threads(4);

  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    #pragma omp barrier
    /* Only one of the threads does this */
    #pragma omp single
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d and my tid = %d\n", nthreads, tid);
    }
  } /* All threads join master thread and terminate */
  return 0;
}
```

Our Sophisticated Hello World!

Results from running on the same machine:

```
There are 8 processors
Hello World from thread = 1
Hello World from thread = 0
Hello World from thread = 3
Hello World from thread = 2
Number of threads = 4 and my tid = 1
-----------------------------------
```

Previous results:

```
There are 8 processors
Hello World from thread = 0
Hello World from thread = 3
Hello World from thread = 1
Number of threads = 4 and my tid = 0
Hello World from thread = 2
```

OpenMP Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications

More Information

- openmp.org
  - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial