# Multi-Core Programming

Revision

# Multi-core Programming

- Parallel Programming Models
  - Message Passing
  - Shared Address Space
  - Data Parallel
- Execution Models
  - Cluster
  - Symmetric Multiple Processors (or Multi-core)
  - Single Instruction Stream in Single core
- Performance

# Performance Models

- Instruction execution time (cycles, response time)
- Throughput when multi-core

# Improving Performance

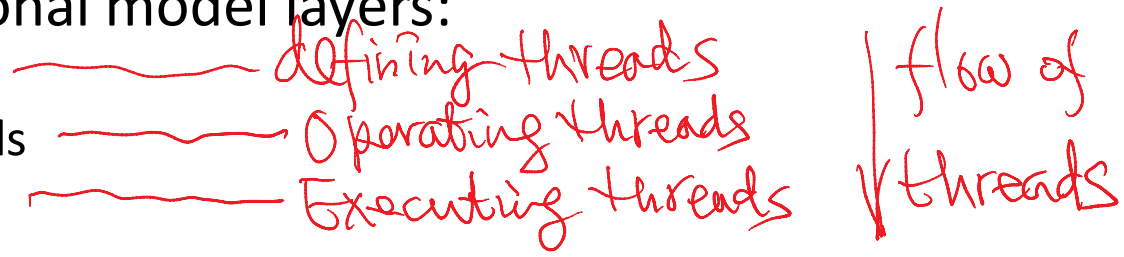
- Hardware Tech.
  - Cache
  - Hardware multi-threading
  - Prefetching
  - Pipelining
  - …
- Programming

# Programming for Performance

- Good Algorithms
- Utilization of Runtime and System
- Utilization of Computation Units
- Workload Balance
- Memory access
- Communication
- Synchronization
- Scheduling and Mapping
- …

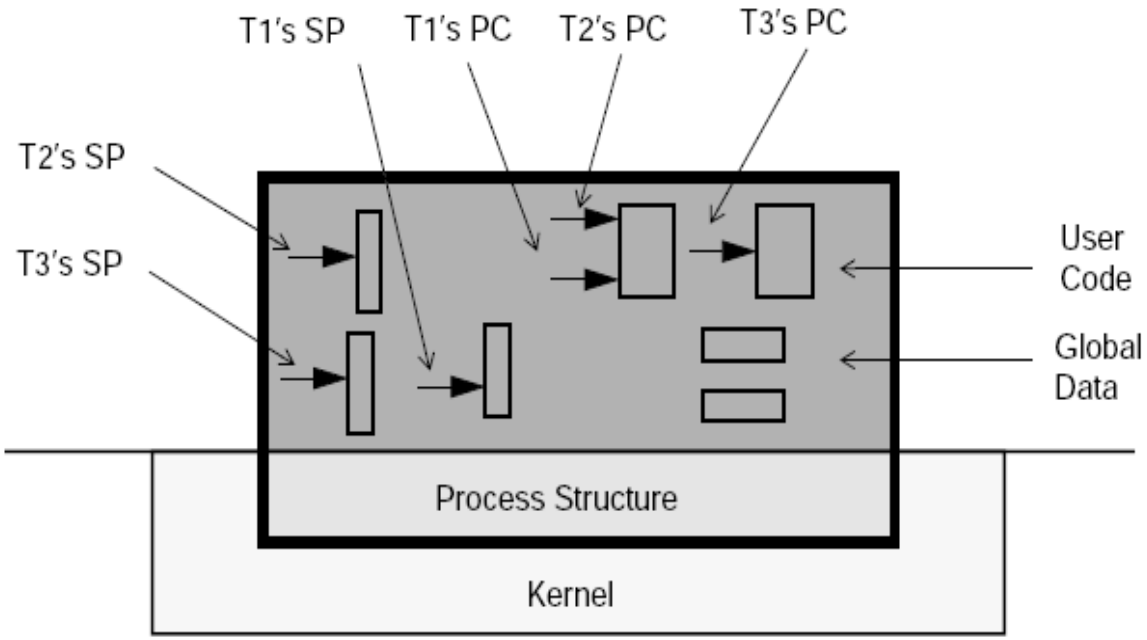
# Lecture02.Multithreading

- Thread computational model layers:
  - User level threads
  - Kernel level threads
  - Hardware threads




- Mapping
  - Thread Level Scheduler in user space
  - OS scheduler in Kernel space

# Explain what is thread according to the following diagram.



- Multiple threads in a process
- All threads share the global data of the process
- Each thread has its own program counter and stack point
- All stacks of threads are all within the memory space of the process

# Lecture03.Parallel Programming Models

- Programming Abstraction and Implementation
- 3 Programming Models
  - Shared Address Space Model
  - Message Passing Model
  - Data Parallel Model
- 3 Architectures
  - Multi-core
  - Cluster
  - SIMD

Why Gather and scatter operations sometimes are needed for data parallel communication? How can these operations be implemented in an old ISPC code?

- Because SIMD Units require a aligned contiguous data
- ISPC may use indirect reference to gather or scatter
- Gather is supported in AVX2 and scatter exists in AVX512

# Lecture04.Message Passing Model

- Non-Buffered Blocking Message Passing Operations
- Buffered Blocking Message Passing Operations
- Non-Blocking Message Passing Operations
- Process Grouping: Communicators
- MPI Point to Point Communication
- MPI Collective Communication

- Simple programs

What things you have to ensure when you write a program with non-blacking message passing operations ?

- The **programmer** must ensure semantics of the send and receive.

MPI uses Communicator to group processes. Why commuincator is important in MPl?

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- MPI Collective communications are all based on the communicator they belong.

# Lecture05.SAS-PThreads

- POSIX Thread Creation and Exit
- *Thread Attributes*
- *Thread Synchronization*
- Write simple parallel programs using synchronizations

# Lecture07.OpenMP - Components of OpenMP

- **The OpenMP Execution Model**
- **Components of OpenMP**
  - Directives
  - Environment Variables
  - Runtime
- **Parallel Region**
- **Work-sharing constructs**
- **Tasks**
- **Synchronization and Scheduling**

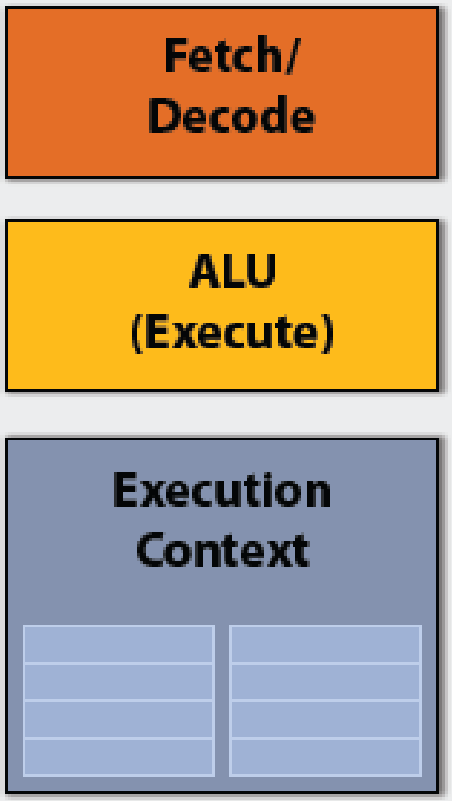
# Lecture09.ISPC Parallel Programming Model

- Gang of  ISPC program instances
- SIMD Implementation
  - Assignment
- ISPC Tasks

# Lecture11.Mixed Parallel Programming

- MPI+OpenMP Hybrid Model
- Usually MPI+X

# Lecture12.Modern Processor

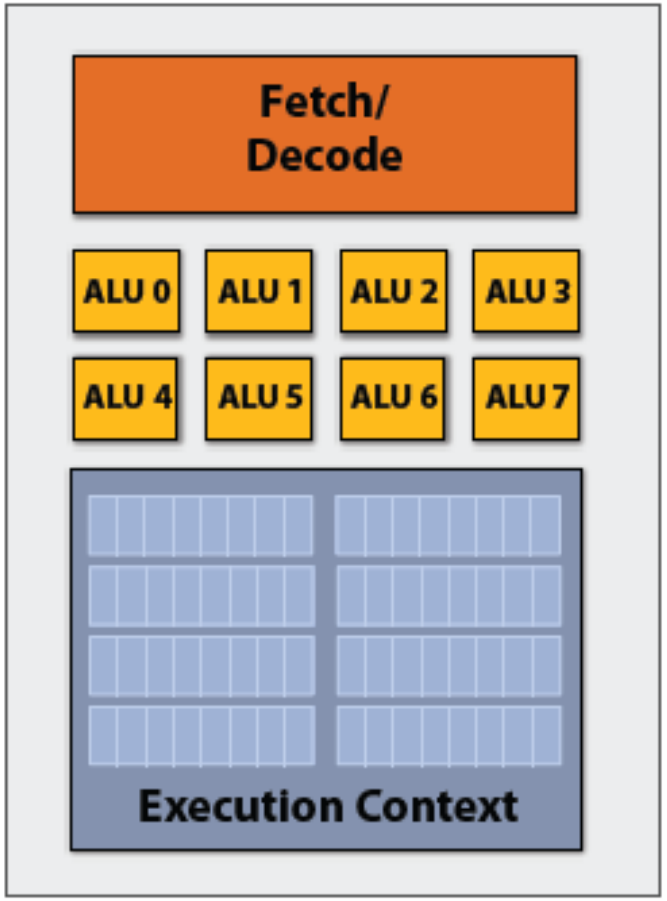



Idea #1:

Use increasing transistor count to add more cores to the processor

Rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

# Add ALUs to increase compute capability




**Idea #2:**
**Amortize cost/complexity of managing an instruction stream across many ALUs**

## SIMD processing

**Single instruction, multiple data**

**Same instruction broadcast to all ALUs**
**Executed in parallel on all ALUs**

向量程序

支持向量的内核函数

# Vector program (using AVX intrinsics)

**Intrinsics available to C programmers**

```
#include <immintrin.h>

void sinx(int N, int terms, float* x, float* result)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         // value += sign * numer / denom
         __m256 tmp = _mm256_div_ps(_mm256_mul_ps(_mm256_set1ps(sign), numer), denom);
         value = _mm256_add_ps(value, tmp);

         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
         sign *= -1;
      }
      _mm256_store_ps(&result[i], value);
   }
}
```

跨度8个float

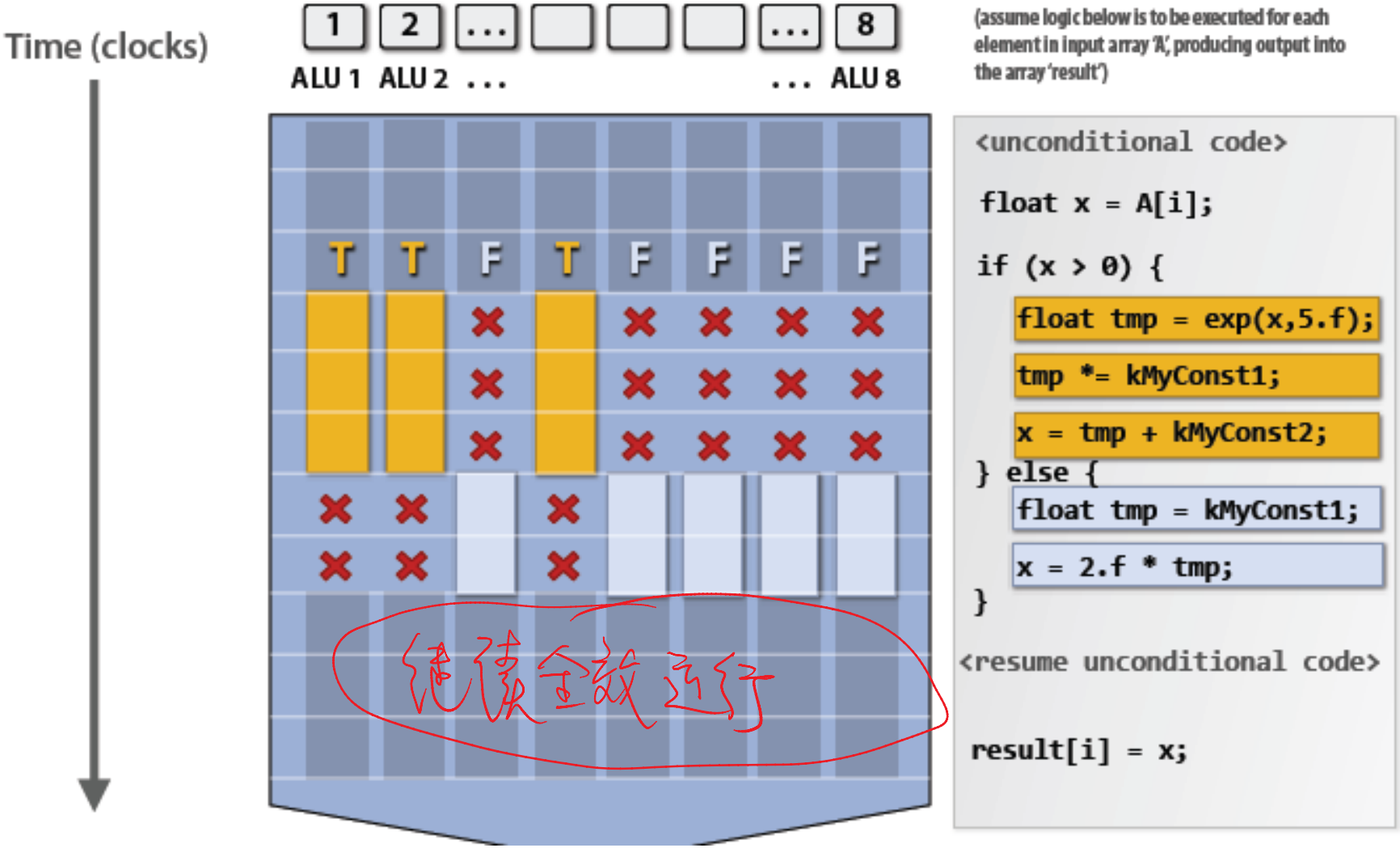
single precise float

32x8 = 256 装18个float

同时8个走(向量)

广播3!

# After branch: continue at full performance



# Lecture13.Accessing Memory

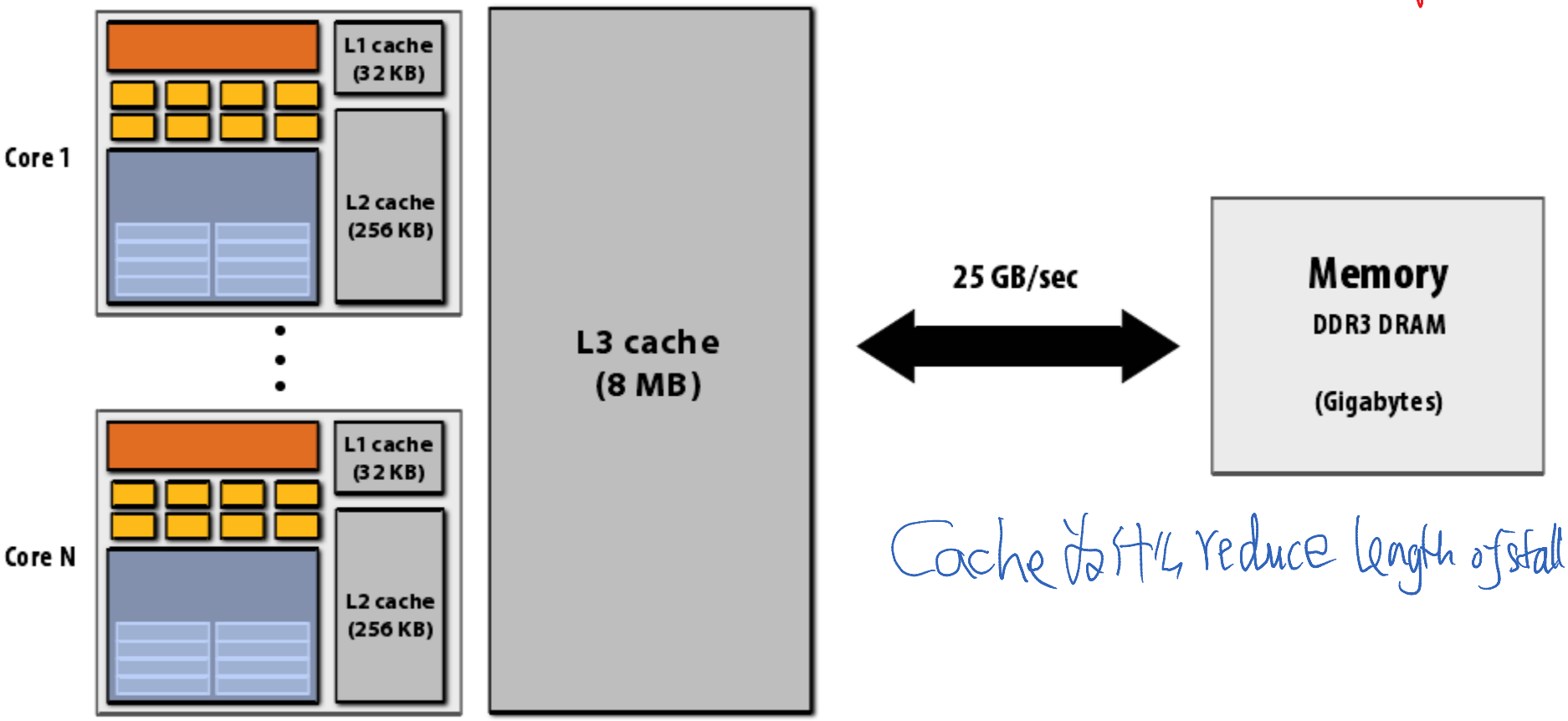
- **Memory latency**
  - **The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system**
  - **Example: 100 cycles, 100 nsec**

- **Memory bandwidth**
  - **The rate at which the memory system can provide data to a processor**
  - **Example: 20 GB/s**

# Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches

Caches reduce memory access latency (* Caches also provide high bandwidth data transfer to CPU) 缓冲降低存储器延时，提高带宽



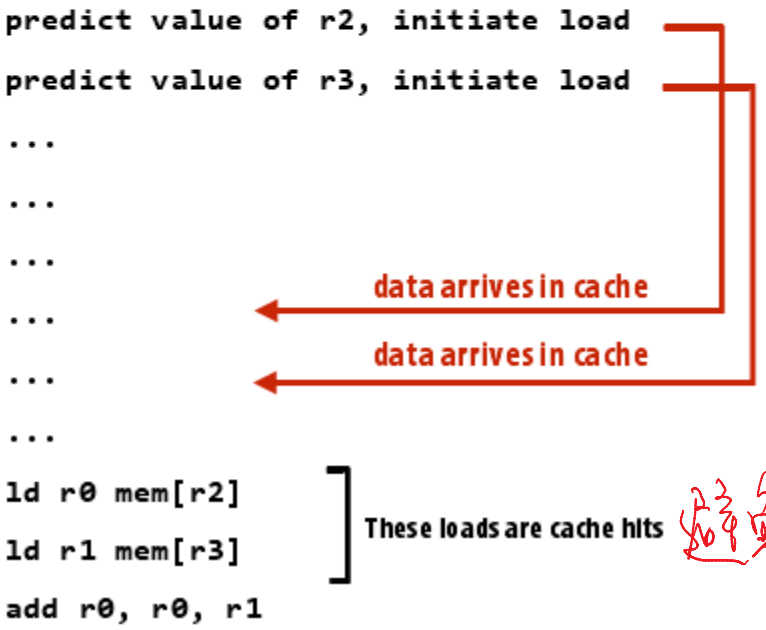

Cache为什么 reduce length of stall?

# Prefetching reduces stalls (hides latency)

- **All modern CPUs have logic for prefetching data into caches**
  - **Dynamically analyze program's access patterns, predict what it will access soon**

现代处理器都有预取数据到 Cache 的机制

- **Reduces stalls since data is resident in cache when accessed**




**Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)**

**(more detail later in course)**

避免了延迟(隐藏延迟)(已在Cache中)

(交错) 多线程减少拖延周期数

# Multi-threading reduces stalls

交错(交替)线程

- Idea: interleave processing of multiple threads on the same core to hide stalls
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

这里先指interleave多线程，后面会讨论SMT。

同核上的交替多线程可以掩盖拖延(隐藏拖延)
(hide stalls)

# Bandwidth is a critical resource

**Performant parallel programs will:**

- **Organize computation to fetch data from memory less often**
  - **Reuse data previously loaded by the same thread (traditional intra-thread temporal locality optimizations)**
  - **Share data across threads (inter-thread cooperation)**
- **Request data less often (instead, do more arithmetic: it's "free")**
  - **Useful term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream**
  - **Main point: programs must have high arithmetic intensity to utilize modern processors efficiently**

不要那么频繁地取数据.

不要那么快(频繁)地申请数据

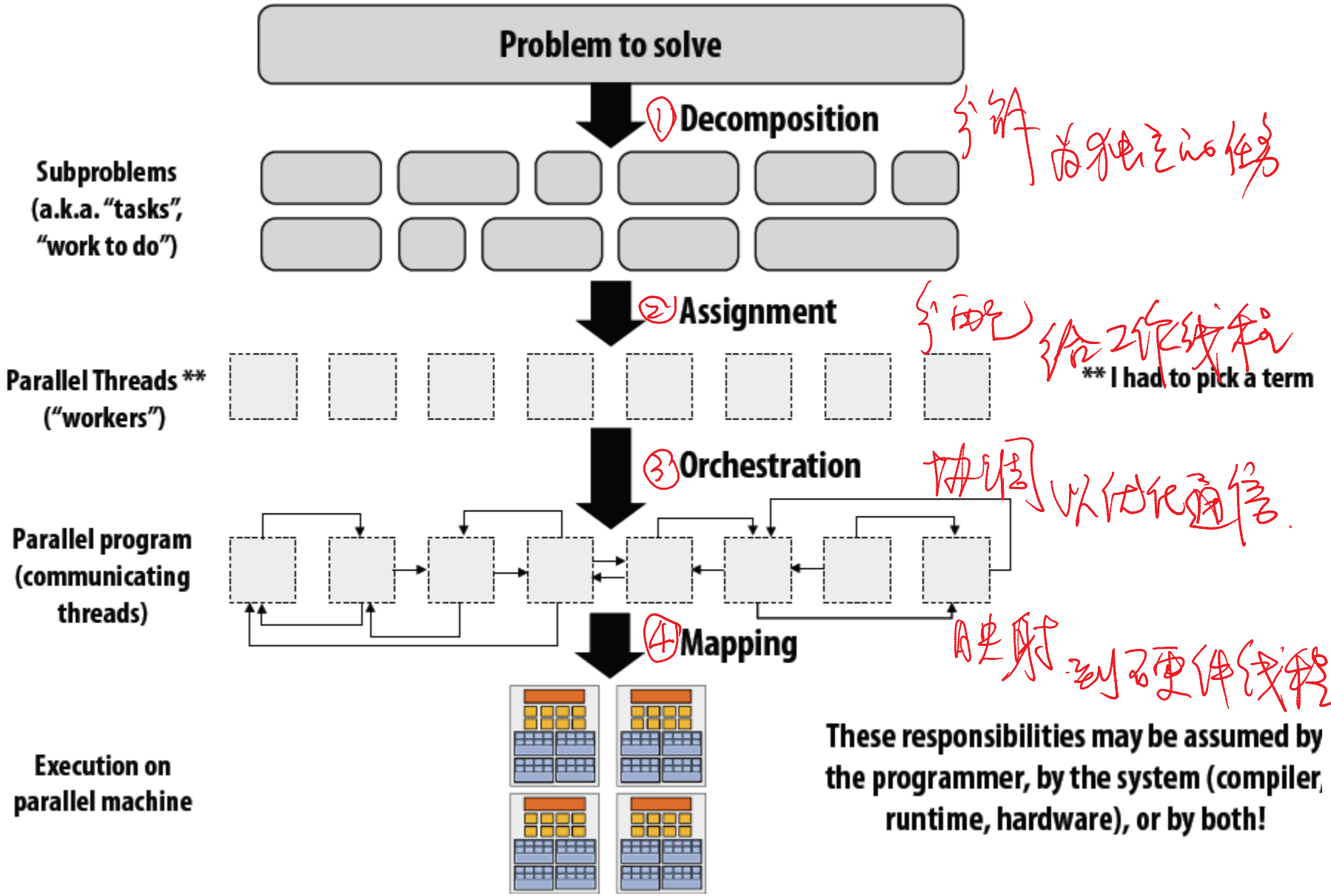
尽量提高计算强度以充分利用现代处理器

计算强度与"计算与通信比"类似

# Lecture14.Programming Basics

- Decomposition
- Assignment
- Orchestration
- Mapping

# Creating a parallel program

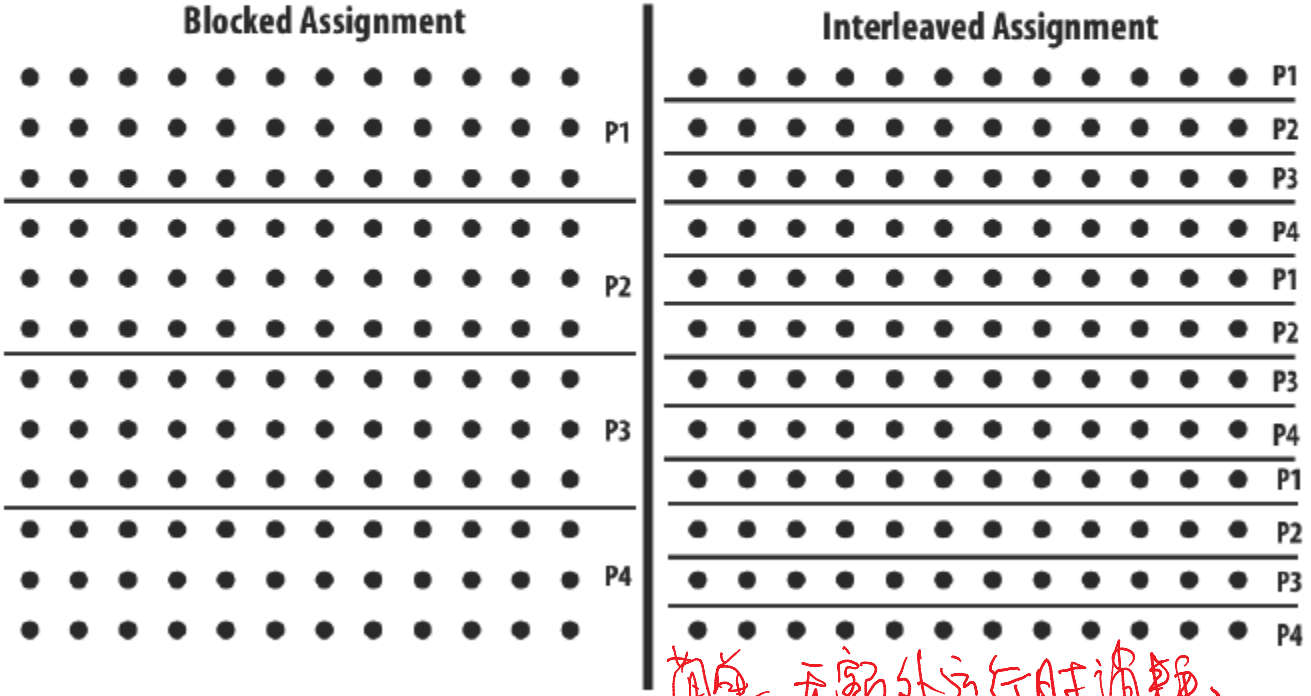


# Lecture15.Workload Balancing

- Task Assignment and Scheduling

# Static assignment

- Assignment of work to threads is pre-determined
  - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells (work) to each thread (worker)
  - We discussed two static assignments of work to workers (blocked and interleaved)




- Good properties of static assignment: simple, essentially zero runtime overhead (in this example: extra work to implement assignment is a little bit of indexing math)

# Dynamic assignment 动态分配

**Program determines assignment dynamically at runtime to ensure a well distributed load. (The execution time of tasks, or the total number of tasks, is unpredictable.)**

Sequential program
(independent loop iterations)

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

执行时间无法事先知

因为不同大小的元素，素数判断

的执行时间相差太大.

Parallel program
(SPMD execution by multiple threads,
shared address space model)

```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;     // shared variable

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

atomic_incr(counter);

① 如何分配的？

② 为什么要 lock？

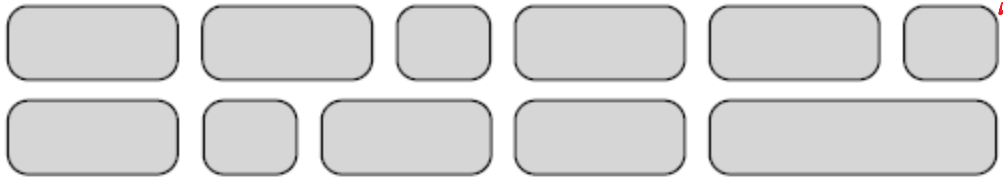
③ 为什么动态分配比较好？

④ 这种分配可能的缺点是什么？(不均衡)

# Dynamic assignment using a work queue

用工作队列进行动态分配.

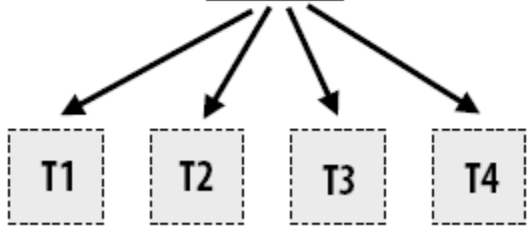
Sub-problems
(a.k.a. "tasks", "work")



分割的子任务(工作)

工作队列

Shared work queue: a list of work to do
(for now, let's assume each piece of work is independent)




Worker threads:
Pull data from shared work queue
Push new work to queue as it is created

执行线程从队列取工作包进行执行并且可以将新工作入队

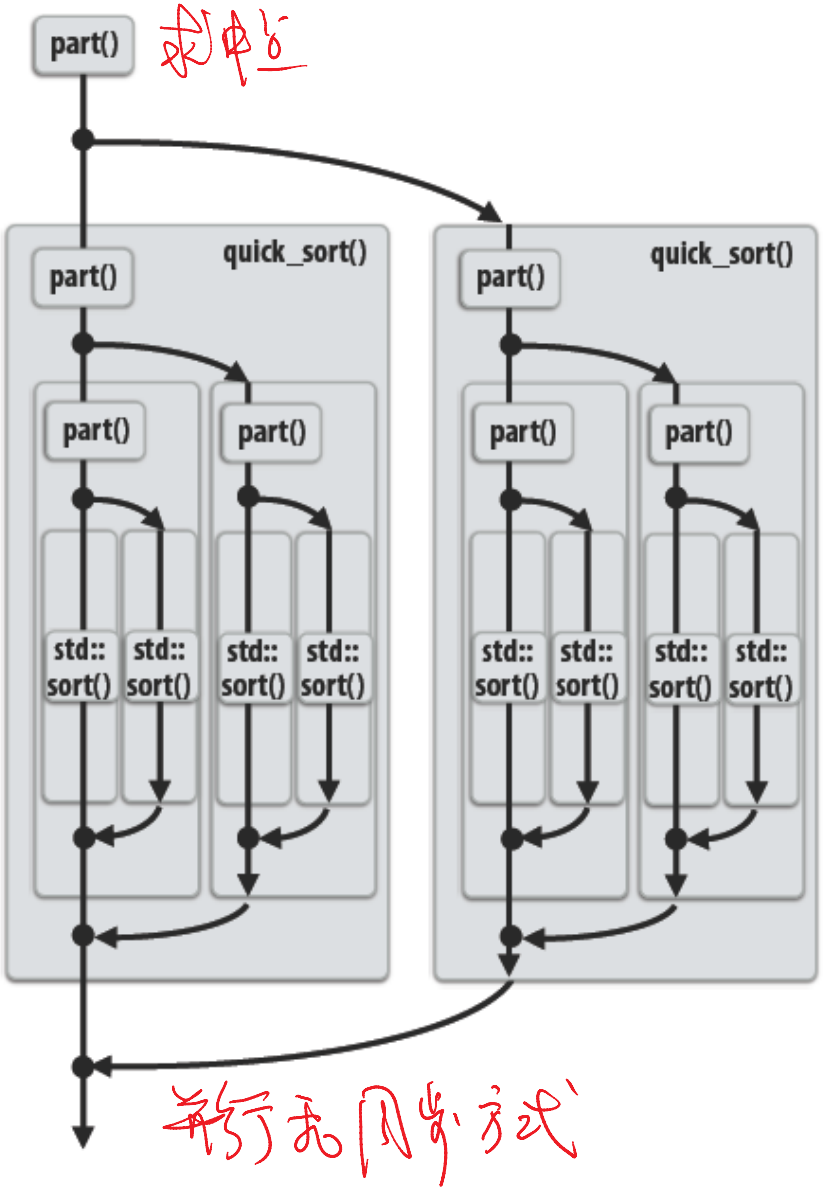
# Lecture16.Scheduling fork-join parallelism

# Parallel quicksort in Cilk Plus (Fork-Join Abstraction)

```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
    int* middle = partition(begin, end);
    cilk_spawn quick_sort(begin, middle);
    quick_sort(middle+1, last);
  }
}
```

Sort sequentially if problem size is sufficiently small (overhead of spawn trumps benefits of potential parallelization)

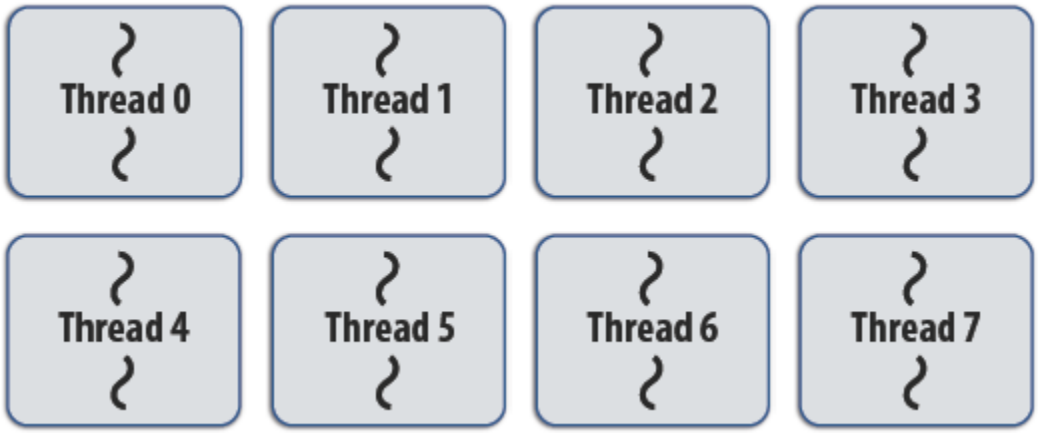
当 Size 太小时，用串行 sort.



# Pool of worker threads

- **The Cilk Plus runtime maintains pool of worker threads**
  - **Think: all threads created at application launch ***

    *** It's perfectly fine to think about it this way, but in reality, runtimes tend to be lazy and initialize worker threads on the first Cilk spawn. (This is a common implementation strategy, ISPC does the same with worker threads that run ISPC tasks.)**
  - **Exactly as many worker threads as execution contexts in the machine**




**Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading**

```
while (work_exists()) {
    work = get_new_work();
    work.run();
}
```

# Implementing work stealing: dequeue per worker

用双向队列实现盗借.

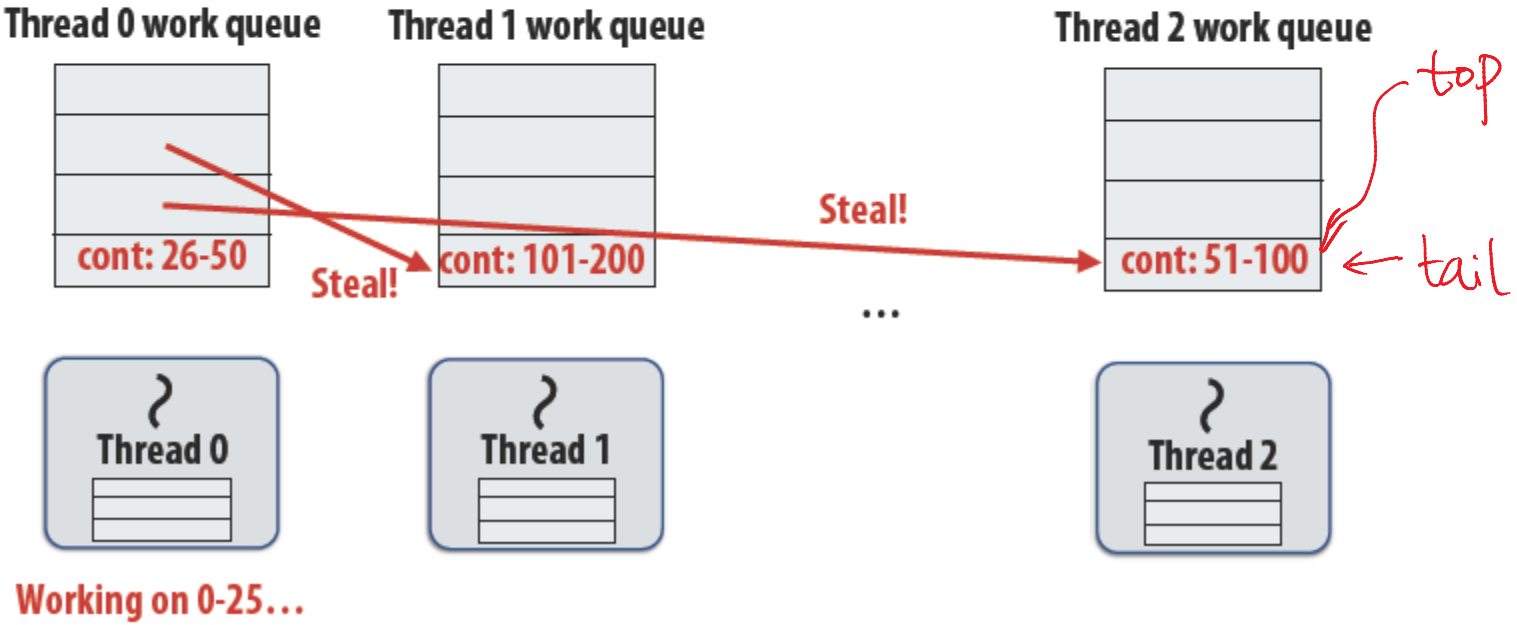
**Work queue implemented as a dequeue (double ended queue)**

- Local thread pushes/pops from the "tail" (bottom)
- Remote threads steal from "head" (top)

本线程在队尾操作(取,放)

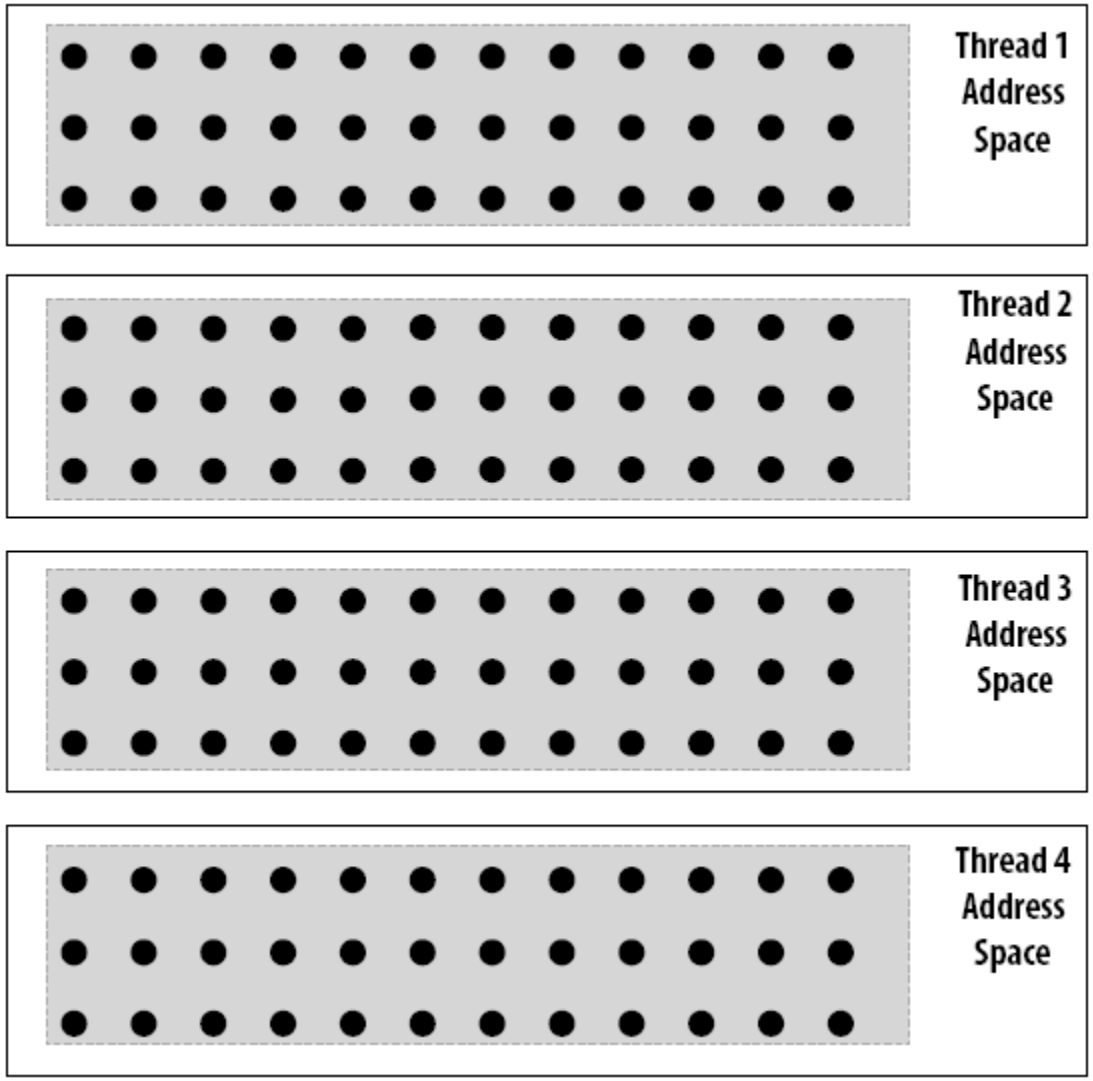
其它线程在队头操作(取)

为什么要这样操作?



# Lecture17.Communication Cost

# Message passing model: each thread operates in its own address space



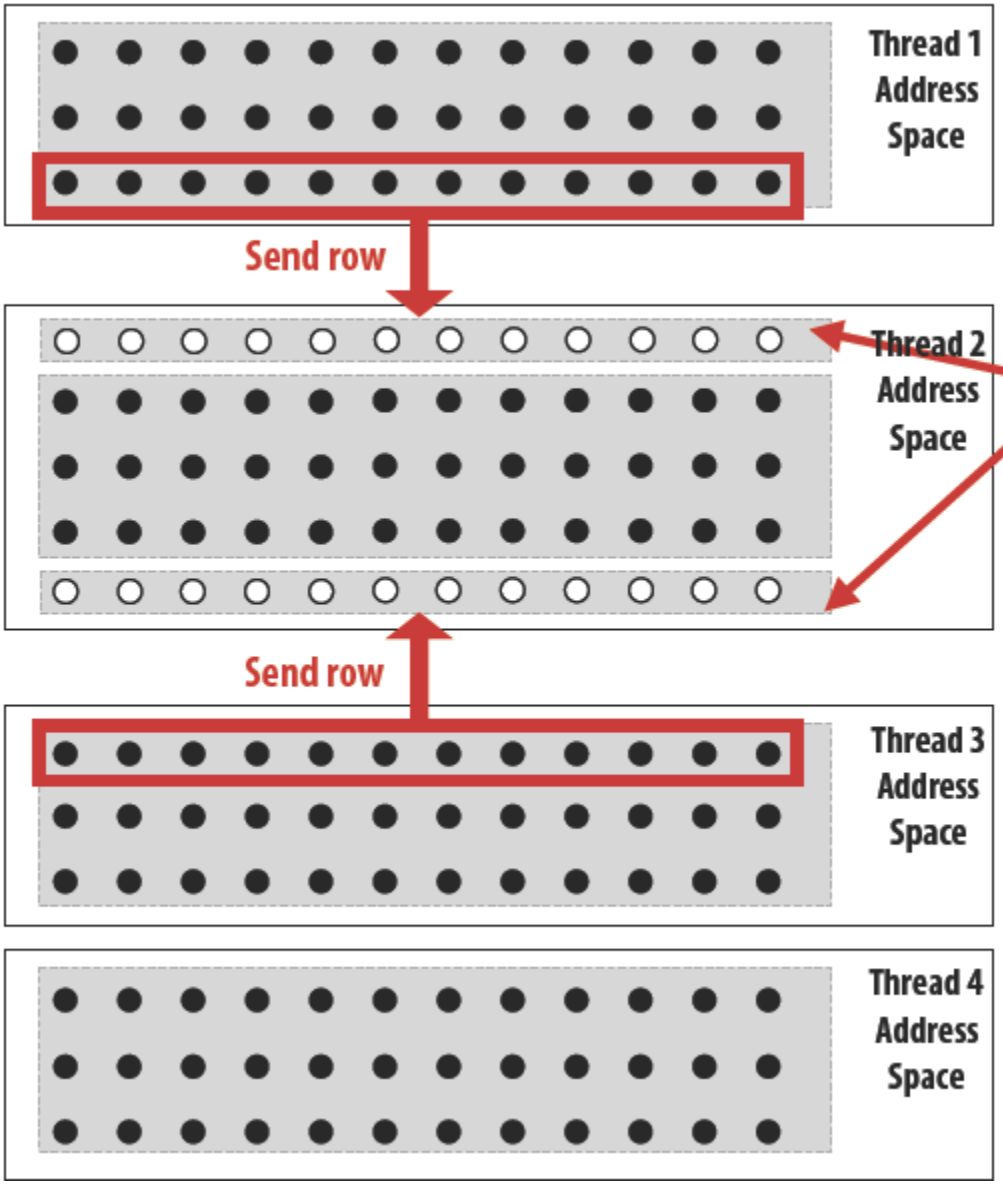

In this figure: four threads

The grid data is partitioned into four allocations, each residing in one of the four unique thread address spaces

(four per-thread private arrays)

不同的地址空间
(按线程分配的私有空间)
(有可能在同一个物理地址空间)

# Data replication is now required to correctly execute the program




**Example:**

**After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2 (thread 2 requires up-to-date red cell information to update black cells in the next phase)**

**"Ghost cells" are grid cells replicated from a remote address space. It's common to say that information in ghost cells is "owned" by other threads.**

**Thread 2 logic:**

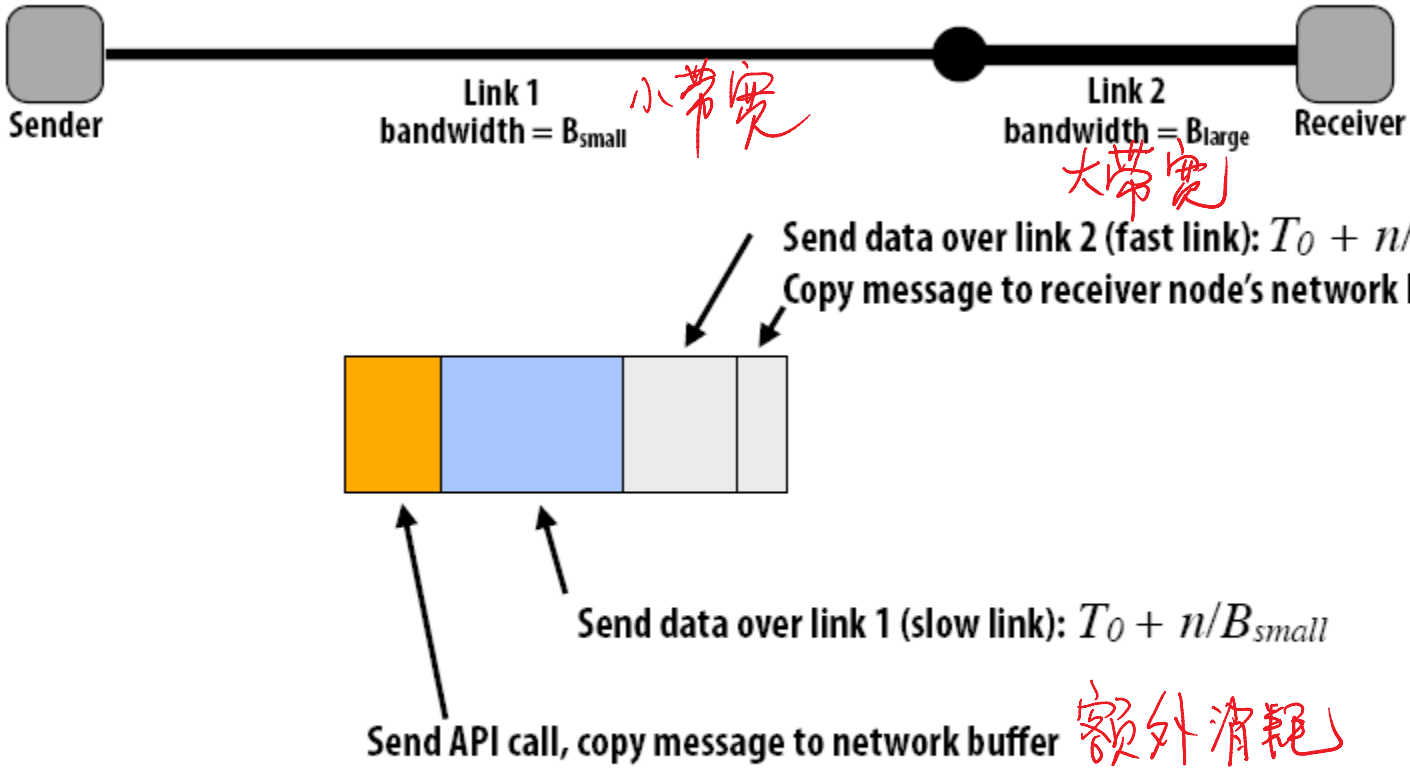
```
float* local_data = allocate(N+2,rows_per_thread+2);

int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);

// receive ghost row cells (white dots)
recv(&local_data[0,0], bytes, tid-1);
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);

// Thread 2 now has data necessary to perform
// future computation
```

# A more general model of communication

## Example: sending a n-bit message

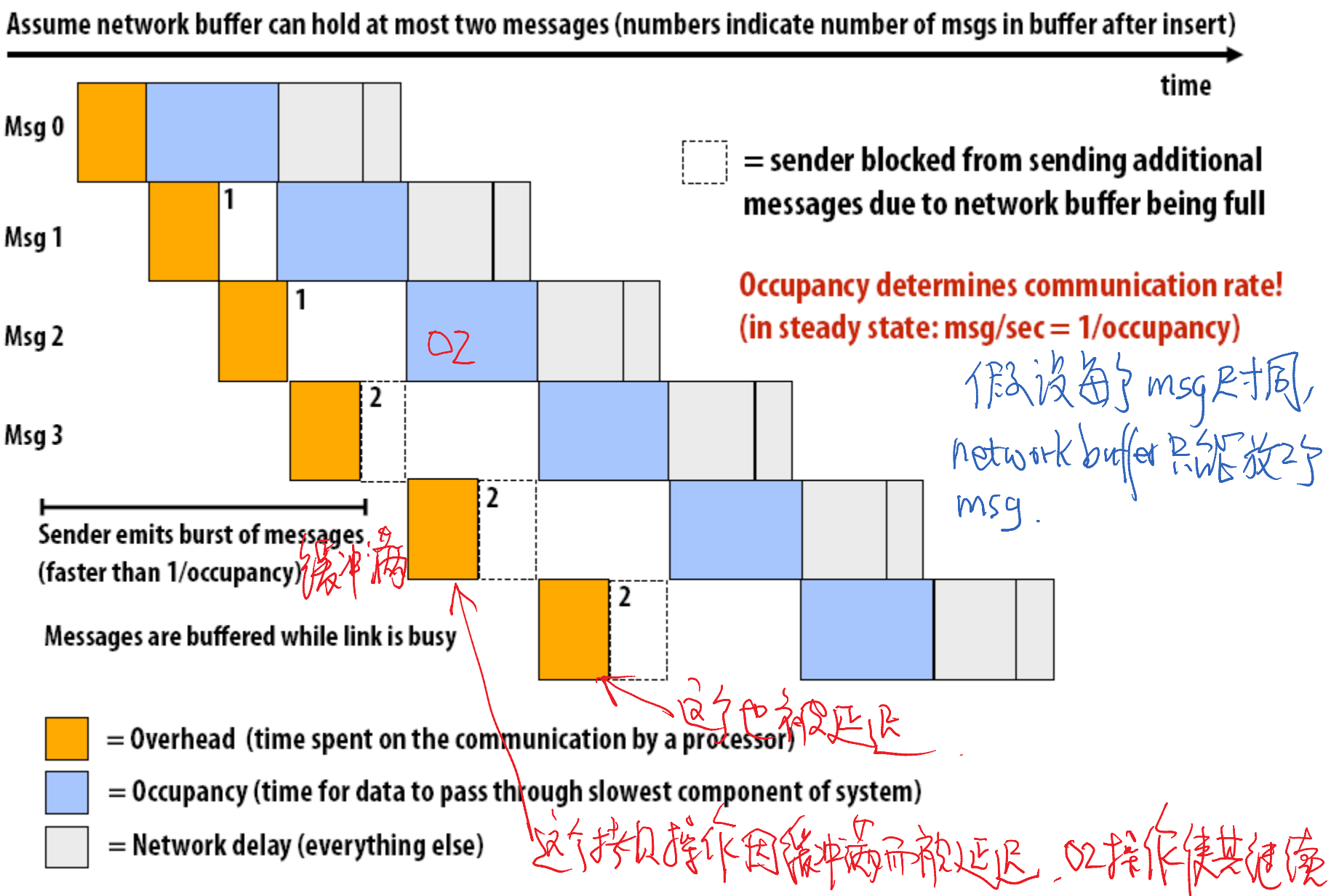
更一般的通信模型

**Total communication time = overhead + occupancy + network delay**



# Pipelined communication

流水式通信.



# Think of a parallel system as an extended memory hierarchy

拓展的存储层次体系

I want you to think of "communication" very generally:

- Communication between a processor and its cache
- Communication between processor and memory (e.g., memory on same machine)
- Communication between processor and a remote memory
  (e.g., memory on another node in the cluster, accessed by sending a network message)

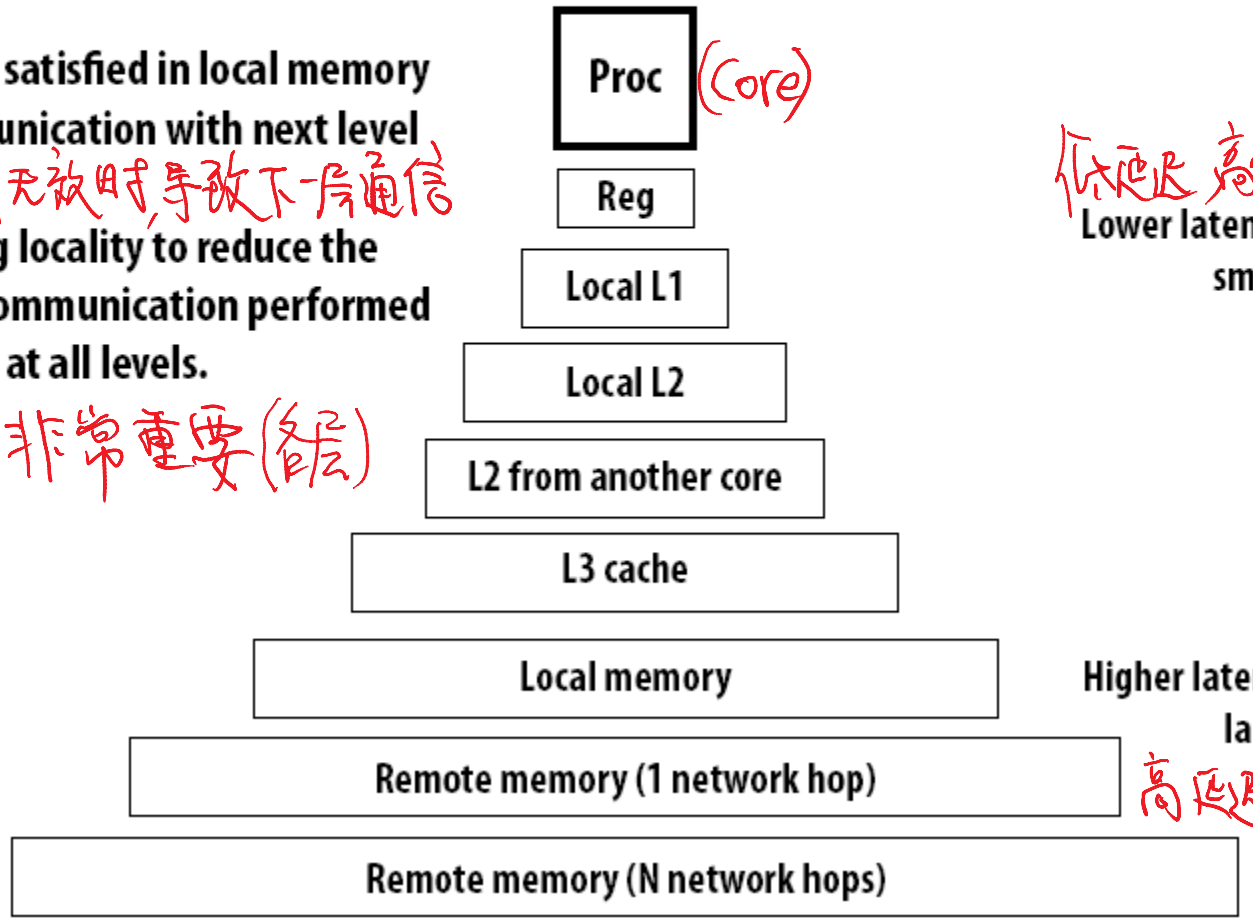
更一般地理解并行计算系统中的通信

Accesses not satisfied in local memory cause communication with next level

局部存储无法时，导致下一层通信

So managing locality to reduce the amount of communication performed is important at all levels.

局域性非常重要(各层)




低延迟 高带宽 小容量

高延迟 低带宽 大容量

# Communication-to-computation ratio

通信计算比率

$$\frac{\text{amount of communication (e.g., bytes)}}{\text{amount of computation (e.g., instructions)}}$$

- If denominator is the execution time of computation, ratio gives average bandwidth requirement of code 表示计算所需的平均带宽
- "Arithmetic intensity" = 1 / communication-to-computation ratio
  - I find arithmetic intensity a more intuitive quantity, since higher is better.
  - It also sounds cooler

算术强度（越高越好，说明程序更多的在做计算）

- High arithmetic intensity (low communication-to-computation ratio) is required to efficiently utilize modern parallel processors since the ratio of compute capability to available bandwidth is high (recall element-wise vector multiple from previous lecture)

为了有效利用现代处理器的功能，尽量高的算术强度是必要的！

# Lecture18.Techniques for reducing communication

- Reduce overhead of communication to sender/receiver
  - Send fewer messages, make messages larger (amortize overhead)
  - Coalesce many small messages into large ones
- Reduce latency of communication
  - Application writer: restructure code to exploit locality
  - Hardware implementor: improve communication architecture
- Reduce contention
  - Replicate contended resources (e.g., local copies, fine-grained locks)
  - Stagger access to contended resources
- Increase communication/computation overlap
  - Application writer: use asynchronous communication (e.g., async messages)
  - HW implementor: pipelining, multi-threading, pre-fetching, out-of-order exec
  - Requires additional concurrency in application (more concurrency than number of execution units)

# Improving temporal locality by fusing loops

合并循环，提高时间局部性。

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op
(arithmetic intensity = 1/3)

每一个"乘"2个读和1个写。

Two loads, one store per math op
(arithmetic intensity = 1/3)

同上。

Overall arithmetic intensity = 1/3

总计算强度 1/3

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

Code on top is more modular (e.g, array-based math library like numarray/numPy in Python)

Code on bottom performs much better. Why? 算术强度提高。

# Improve arithmetic intensity by sharing data

共享数据提高计算能力

- **Exploit sharing: co-locate tasks that operate on the same data**

把操作相同数据的任务调度到一起

  - **Schedule threads working on the same data structure at the same time on the same processor**
  - **Reduces inherent communication**

这样减少固有通信

- **Example: CUDA thread block**

线程块

  - **Abstraction used to localize related processing in a CUDA program**
  - **Threads in block often cooperate to perform an operation (leverage fast access to / synchronization via CUDA shared memory)**
  - **So GPU implementations always schedule threads from the same thread block on the same GPU core**

在同一个GPU核上调度相同线程块内的线程

# Exploiting spatial locality

数据空间局域性.

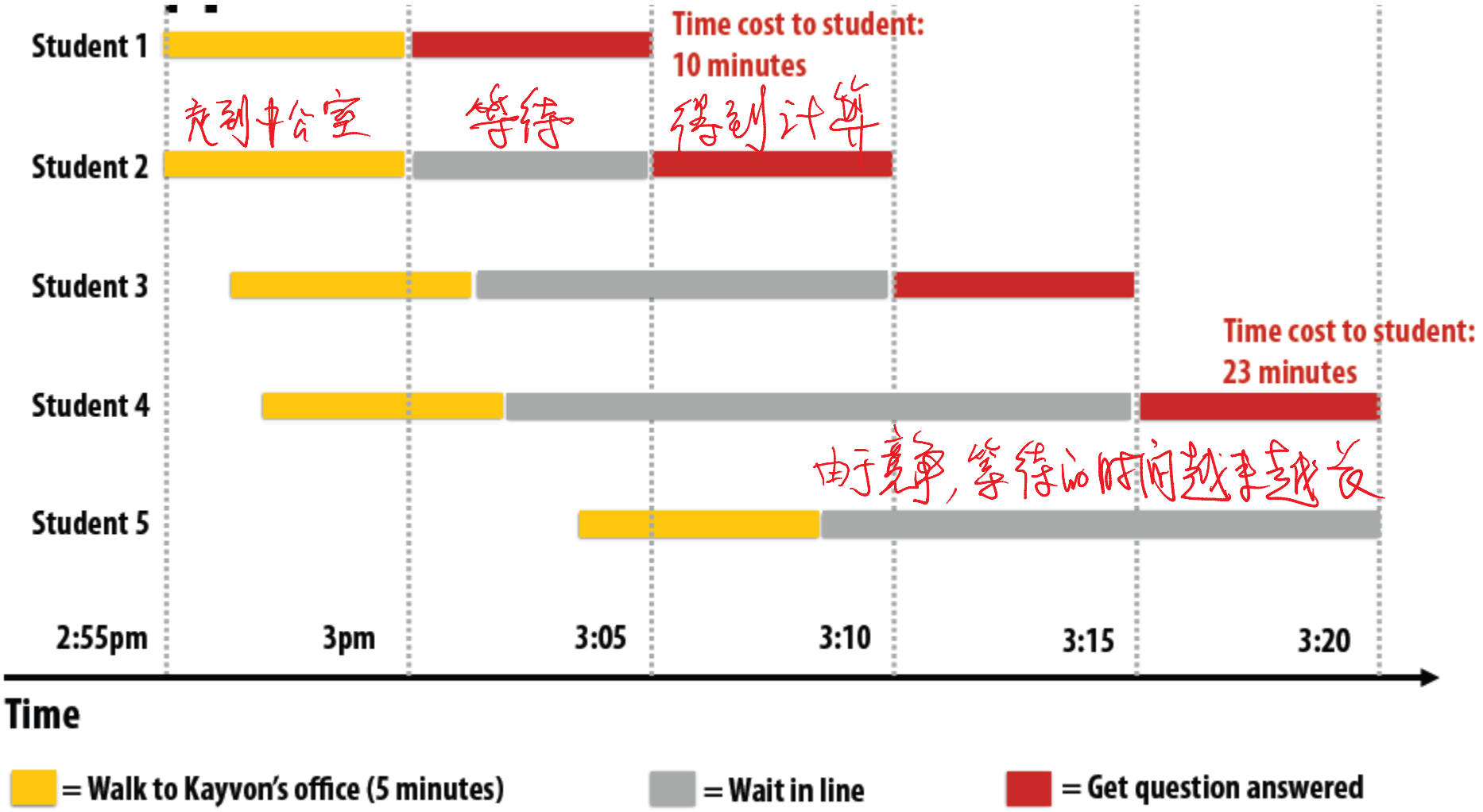
- **Granularity of communication can be important because it may introduce artifactual communication**
  - **Granularity of communication / data transfer**
  - **Granularity of cache coherence**

通信粒度可能导致人为通信.

通信粒度
数据传送粒度

缓存一致的粒度

共享计算资源

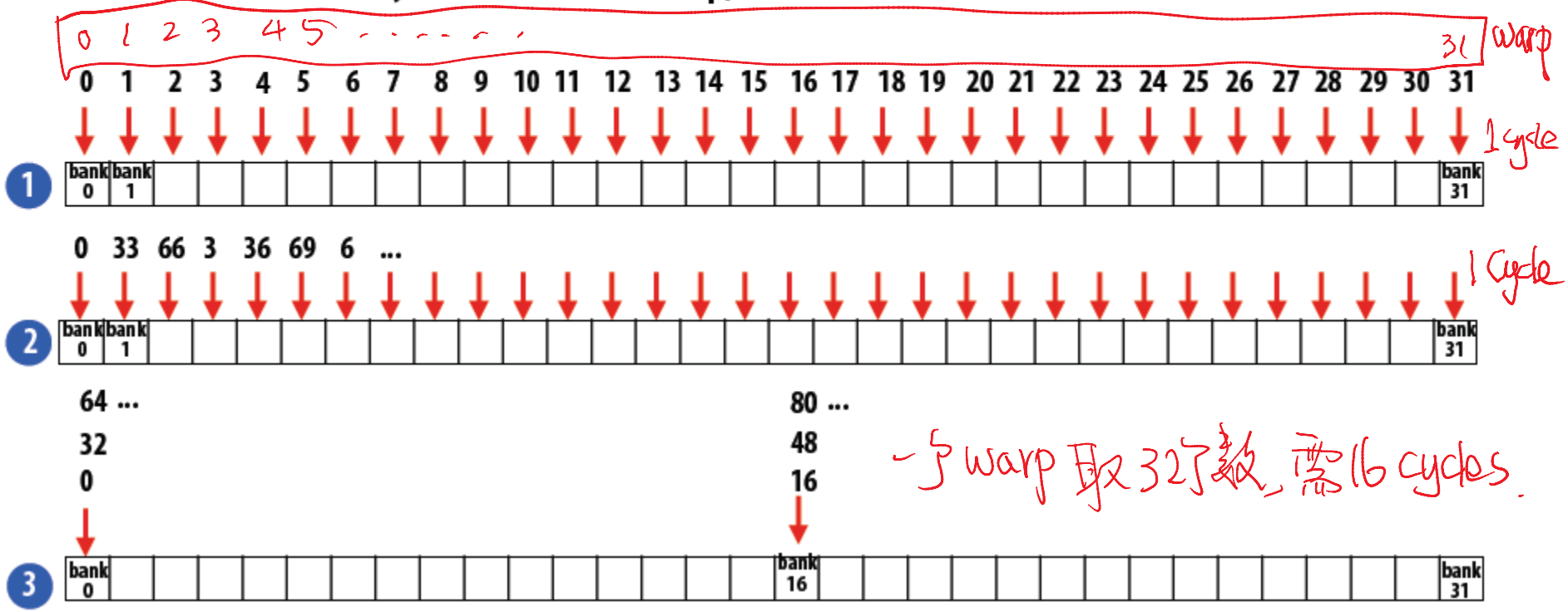



**Problem: contention for shared resource results in longer overall operation times (and likely higher cost to students)**

# Example: Accessing NVIDIA GTX 480 shared memory

- Shared memory implementation
  - On-chip storage, physically partitioned into 32 SRAM banks
  - Address X is stored in bank B, where B = X % 32
  - Each bank can provide one word of data to warp per clock
- Figure shows memory addresses requested from each bank as a result of a shared memory load instruction (keep in mind this instruction is executed by all 32 threads in a warp)

```
__shared__ float A[512];

int index = threadIdx.x;

1 float x2 = A[index];       // single cycle
2 float x3 = A[3*index];     // single cycle
3 float x4 = A[16 * index]; // 16 cycles
```



# Examination

- Open Book
- Briefly Answer Questions
- Code Parallel programs in a certain programming Model in a certain language with performance Consideration

Good Luck!