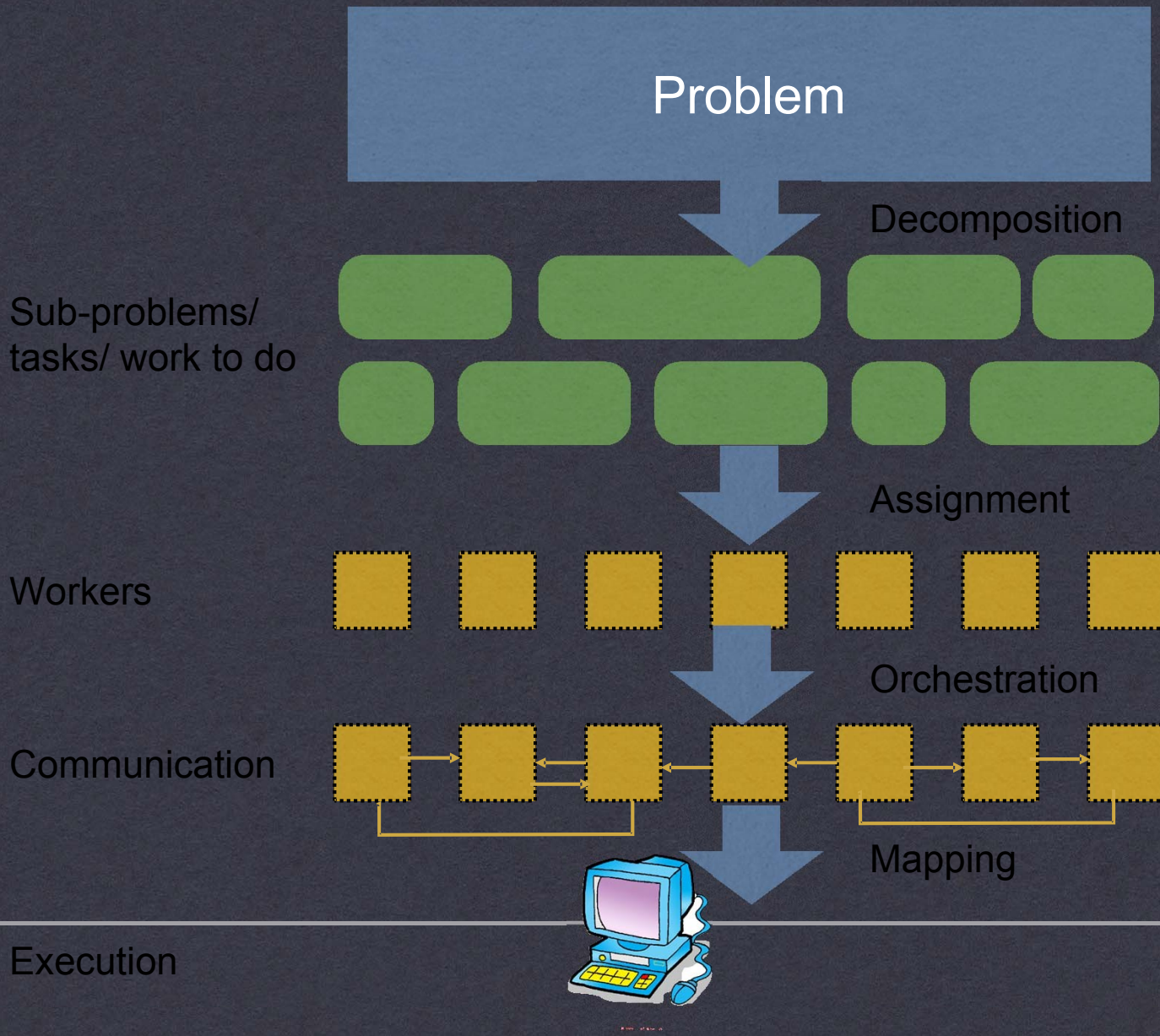


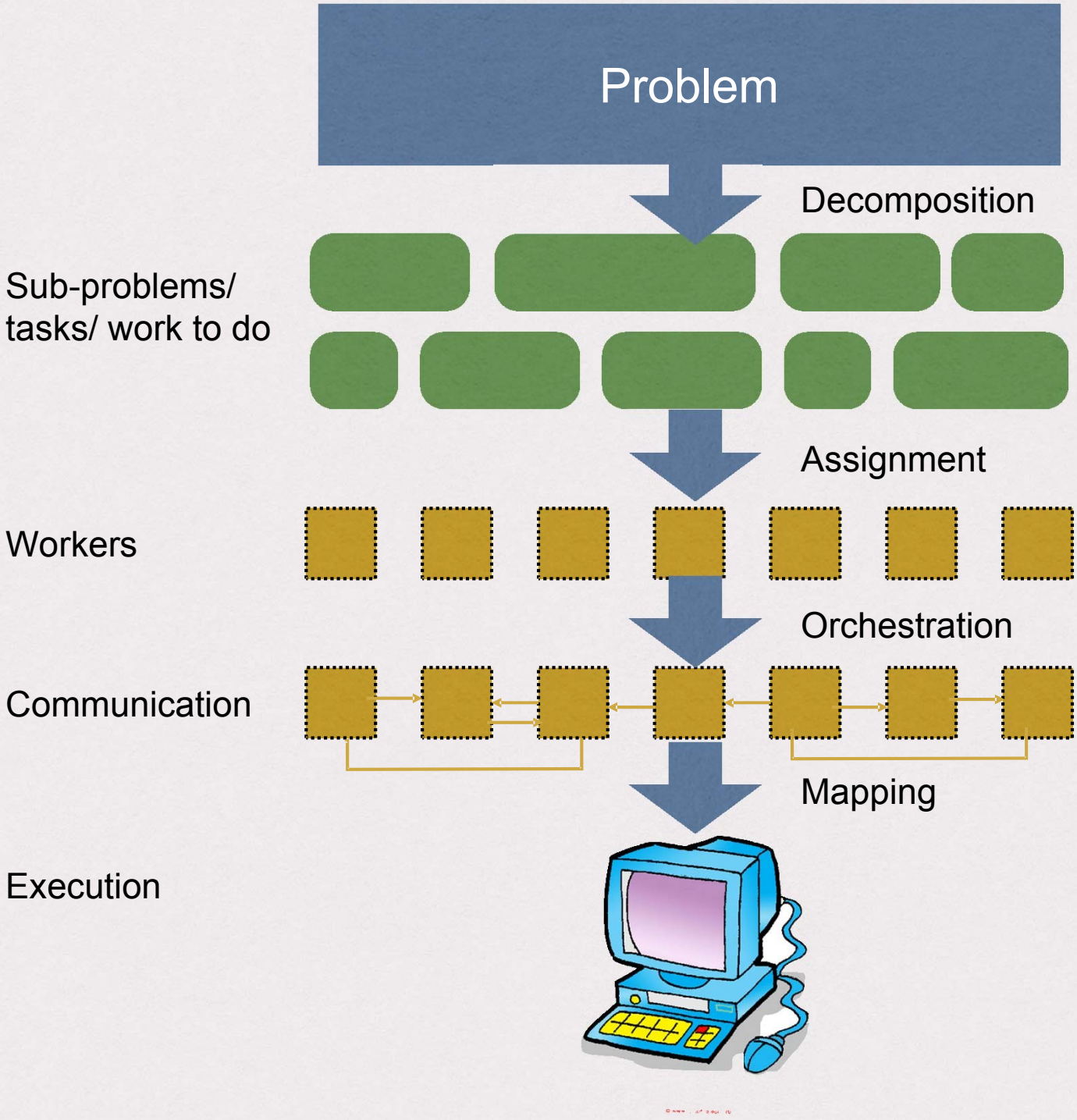
PARALLEL AND DISTRIBUTED COMPUTING

PERFORMANCE OPTIMISATION

Refresh Amdahl's law

- Let S=the fraction of sequential execution that is inherently sequential
- The maximum speedup on P processors is given by:

$$speedup \leq \frac{1}{S + \frac{1-S}{P}}$$


Problem
Decomposition
Sub-problems/
tasks/ work to do
Assignment
Workers
Orchestration
Communication
Mapping
Execution

High performance

- Optimizing the performance of parallel programs is an iterative process of refining choices for decomposition, assignment, and orchestration...
- Key goals:
  - Balance workload onto available execution resources
  - Reduce communication (to avoid stalls)
  - Reduce extra work (overhead) performed to increase parallelism, manage assignment

- Tip #1: Always implement the easiest/simplest solution first, then:
  - measure
  - then decide if you need to do performance improvement:
    - will you have large workloads?
    - will you have large input data?

Balancing workload

1. Identifying enough concurrency in decomposition, and overcoming Amdahl's Law.
2. Deciding how to manage the concurrency (statically or dynamically).
3. Determining the granularity at which to exploit the concurrency.
4. Reducing serialization and synchronization cost.

1. Identifying enough concurrency

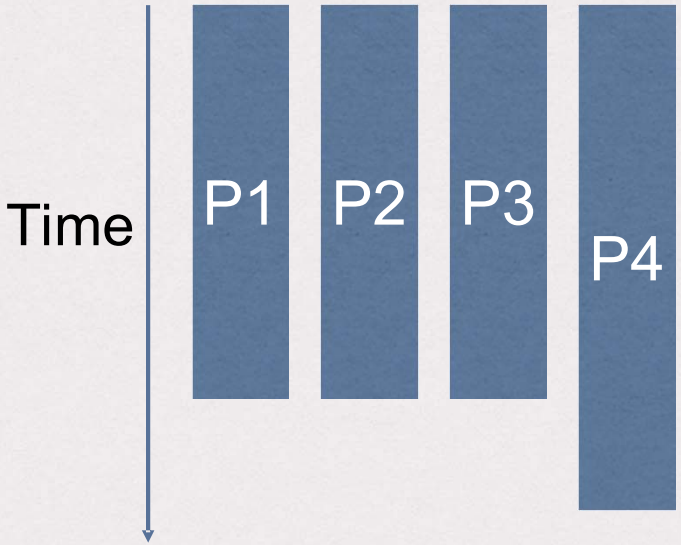
- Look to see if problem has data parallelism and function parallelism
- Data parallelism: same function performed on all the data
- Function parallelism: entirely different calculations are performed concurrently on either the same or different data.
  - eg. pipelining

1. Identifying enough concurrency

- If both data and function parallelism are available in the application, we need to choose which to parallelise
- Function parallelism does not usually grow with the size of the problem being solved - why is that?
- Data parallelism grows with the size of the problem

2. Deciding how to manage concurrency

- Best case: all processors are computing all the time during program execution
  - they are computing simultaneously, and they finish their portion of the work at the same time



- executes for 20% longer ==> 20% of program runtime is essentially serial execution
- not much speedup!

Static assignment

- One of the most basic techniques, where work is allocated to the threads by the programmer
- Programmer needs to know the cost (execution time) and the amount of work (how many tasks)

Near-static assignment

- Do an assignment once, then re-adjust if needed after profiling the application
- Use prediction based on the execution time of previous tasks

Dynamic assignment

- Assignment determined at runtime to ensure well balanced load
  - Execution time of tasks and total number of tasks is unpredictable

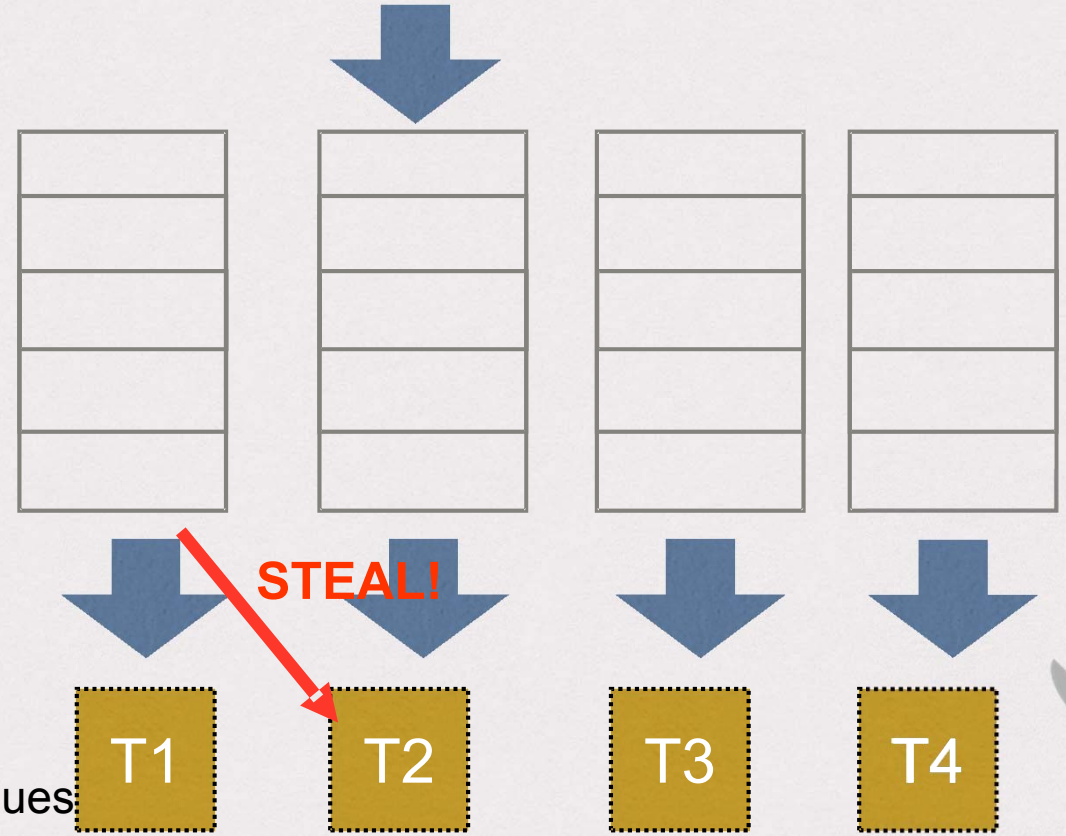
Some rules

- Useful to have many more tasks than processors (many small tasks enables good workload balance via dynamic assignment)
  - Motivates small granularity tasks
- But want as few tasks as possible to minimize overhead of managing the assignment
  - Motivates large granularity tasks
- Ideal granularity depends on many factors:
  - you must know your machine and your workload

Smarter schedulers

- Might schedule longer tasks first
- Might schedule based on priorities etc

Sub-problems/
tasks/ work to do



Worker threads
Pull data from own queue
Push work to own queue
When own queue empty, steal from other threads' queues

Distributed work queues

- Costly synchronization/communication occurs during stealing
  - But not every time a thread takes on new work
  - Stealing occurs only when necessary to ensure good load balance
- Leads to increased locality
- Common case: threads work on tasks they create (producer-consumer locality)

Challenges

- Who to steal from?
- How much to steal?
- How to detect program termination?
- Ensuring local queue access is fast (while preserving mutual exclusion)

Intro to distributed schedulers in clusters

- Users submit their jobs to a cluster's scheduler
- Jobs are queued
- Jobs in queue considered for allocation whenever state of a machine changes (submission of a new job, exit of a running job)
- Allocation – which job in the queue?, which machine?

Schedulers

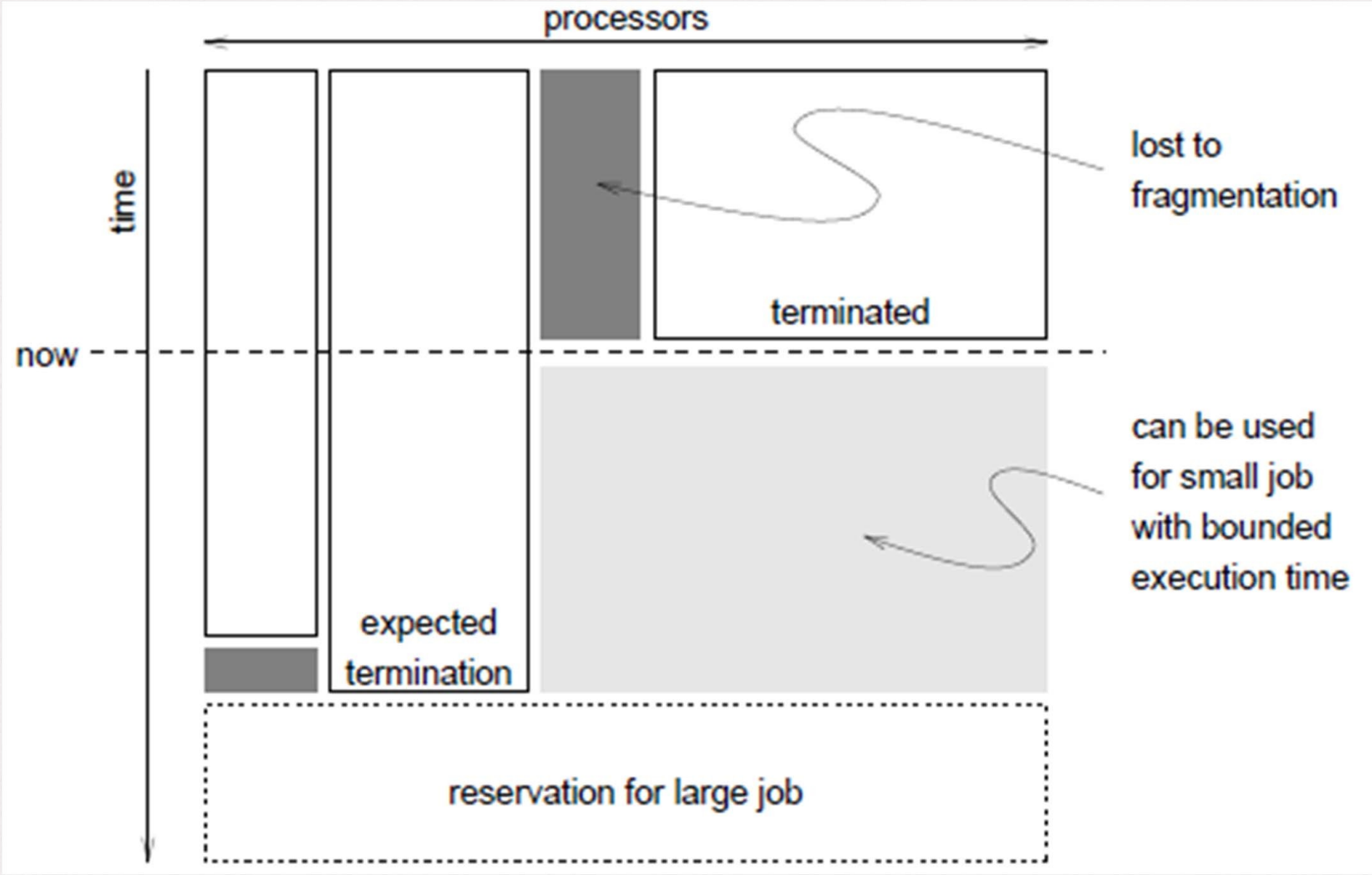
- Packing jobs to the processors
- Additional Goal – to increase processor utilization across cluster
- Lack of knowledge of future jobs and job execution times. Hence simple heuristics to perform packing at each scheduling event
- Current scheduling decisions impact future job allocations, and utilisation

FCFS

- If the machine's free capacity cannot accommodate the first job, it will not attempt to start any subsequent job
- No starvation; But poor utilization
- Processing power is wasted if the first job cannot run

Backfiling

- Allows small jobs from the back of the queue to execute before larger jobs that arrived earlier
- Requires job runtimes to be known in advance – often specified as runtime upper-bound


processors
time
now
lost to
fragmentation
terminated
can be used
for small job
with bounded
execution time
expected
termination
reservation for large job

3. Reducing communication

- Tradeoff with load balance is reducing interprocessor communication
- Decomposing a problem into multiple tasks usually means that there will be communication among tasks
- If these tasks are assigned to different processes, we incur communication among processes and hence processors

4. Reducing unnecessary work

- Sometimes the sequential program is better than the parallel program
- Tip 2: If needed, compute your own data values rather than have one process compute them and communicate them to the others
  - may be a good tradeoff when the cost of communication is high
- Examples:
  - all processes computing their own copy of the same shading table in computer graphics applications
  - trigonometric tables in scientific computations.
- If the redundant computation can be performed while the processor is otherwise idle due to load imbalance, its cost can be hidden.

Quick question

- You have been given a parallel program whose performance is lacking and have been asked to improve it
- What questions do you ask?

PARALLEL AND DISTRIBUTED COMPUTING

Terminology

- ***Latency***: The amount of time needed for an operation to complete.
  - Example: A memory load that misses the cache has a latency of 200 cycles.
  - A packet takes 20 ms to be sent from my computer to Google
- ***Bandwidth:*** The rate at which operations are performed.
  - Example: Memory can provide data to the processor at 25 GB/sec.
  - A communication link can send 10 million messages per second.

PARALLEL AND DISTRIBUTED COMPUTING

Communication - model 1

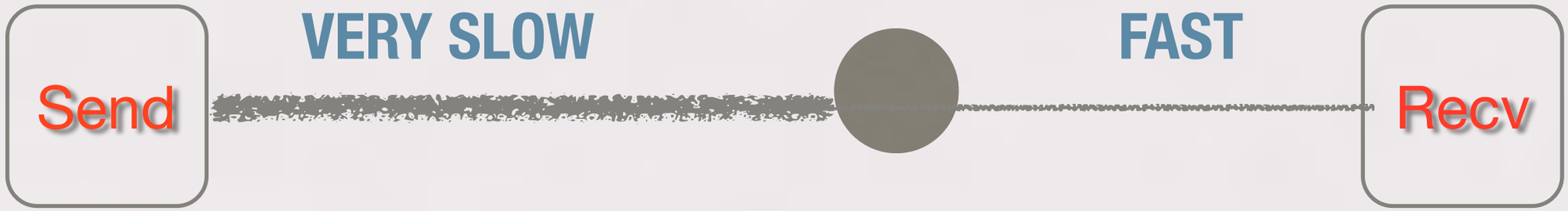
$$T(n) = T_0 + \frac{n}{B}$$

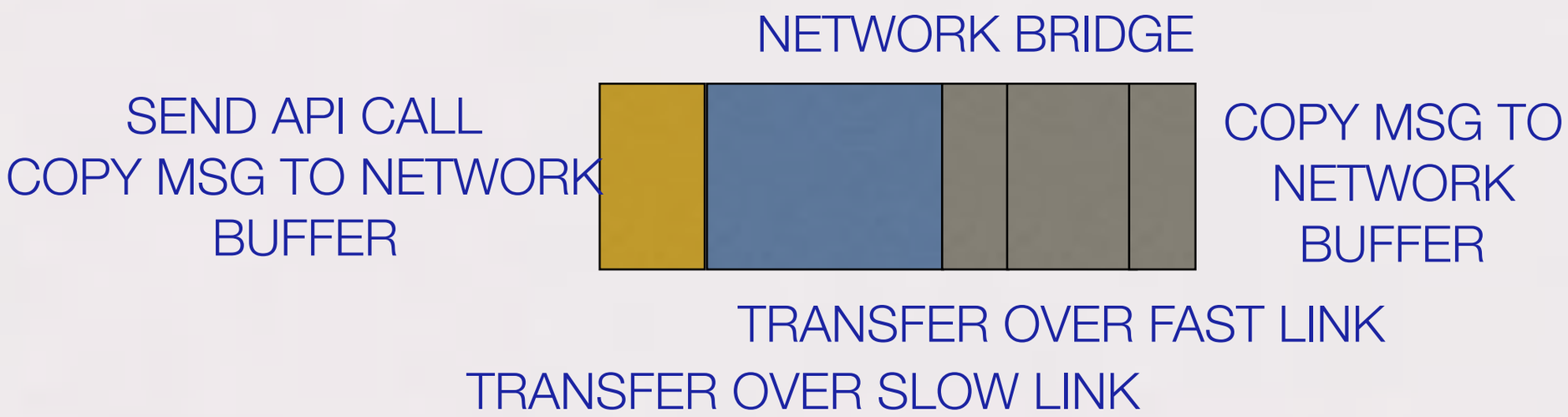
- T(n) = transfer time (overall latency of the operation)
- T0 = start-up latency (e.g., time until first bit arrives)
- n = bytes transferred in operation
- B = transfer rate (bandwidth of the link)
- Assumption: processor does no other work while waiting for transfer to complete ...
- Effective bandwidth = n / T(n)
- Effective bandwidth depends on transfer size

PARALLEL AND DISTRIBUTED COMPUTING

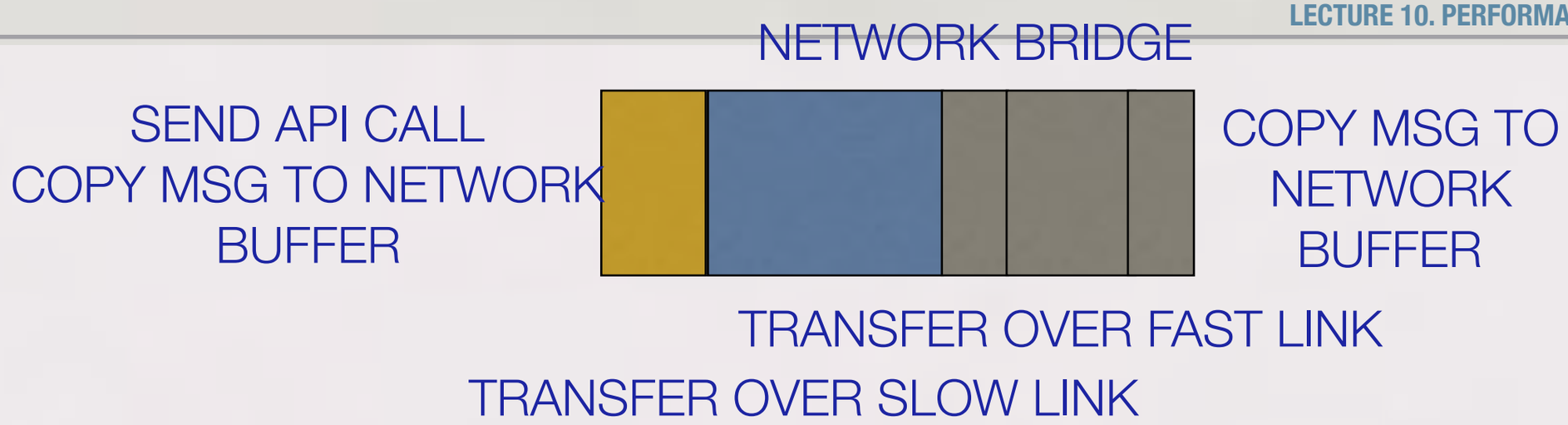
Communication: model 2

- Communication time = overhead + occupancy + network delay

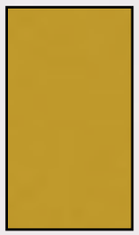




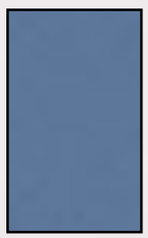
PARALLEL AND DISTRIBUTED COMPUTING



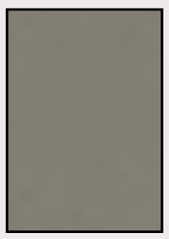
- Occupancy determines communication rate (effective bandwidth)



Overhead (time spent on the communication by a processor)



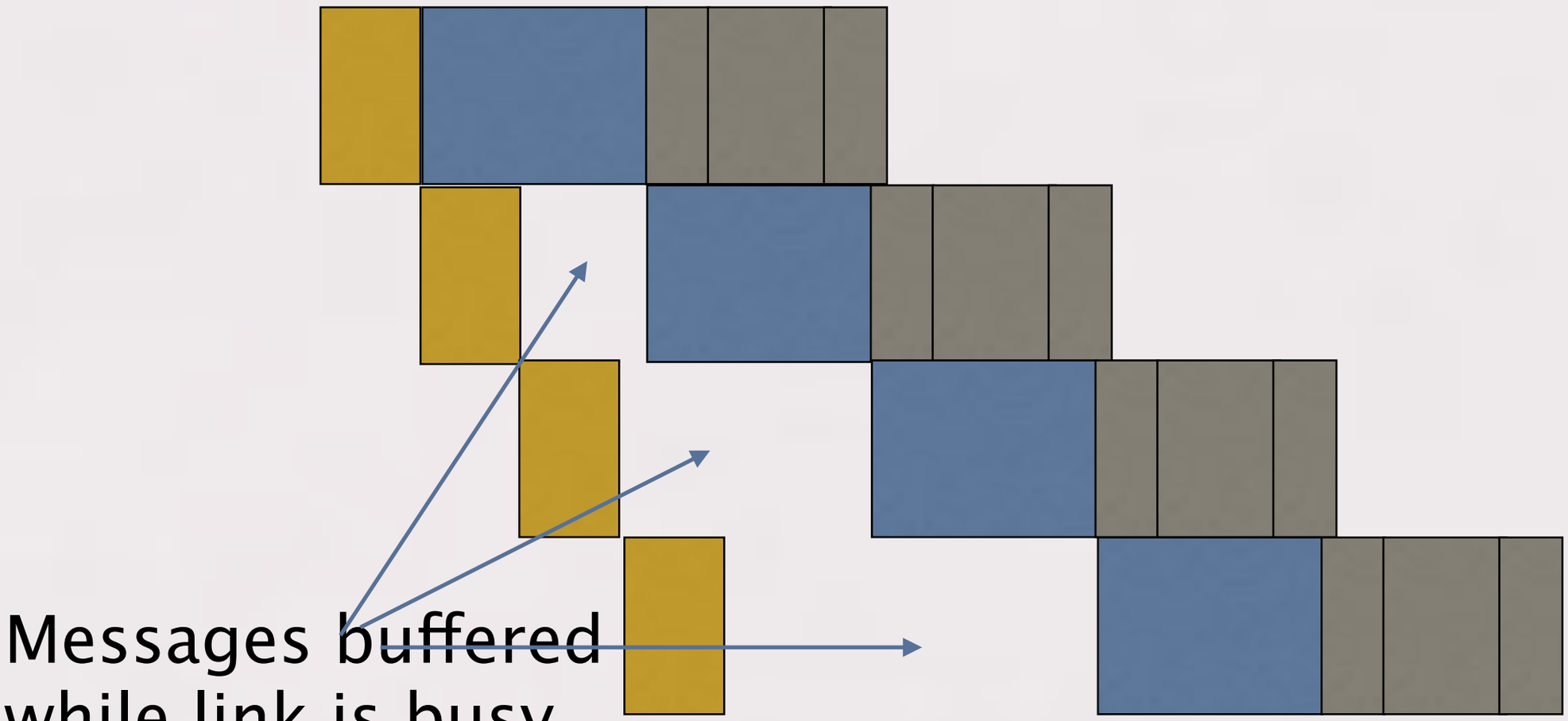
Occupancy (time for data to pass through slowest component of system)



Network delay (everything else)

PARALLEL AND DISTRIBUTED COMPUTING

Pipelined communication



- Occupancy determines communication rate (effective bandwidth)

PARALLEL AND DISTRIBUTED COMPUTING

Communication cost

- The effect operations have on program execution time(or some other metric, e.g., power…)
  - “That function has very high cost” (cost of having to perform the instructions)
  - “My slow program sends most of its time waiting on memory.” (cost of waiting on memory)
- Total communication cost = num_messages x (communication time - overlap)
  - Overlap: portion of communication performed concurrently with other work
  - “Other work” can be computation or other communication (as in the previous example)

PARALLEL AND DISTRIBUTED COMPUTING

Communication …

- 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 — e.g. by sending a network message)

PARALLEL AND DISTRIBUTED COMPUTING

A very simple memory hierarchy

Low latency,
high bandwidth,
small capacity

High latency,
low bandwidth,
larger capacity

PARALLEL AND DISTRIBUTED COMPUTING

Communication to computation ratio (C2CR)

- Amount of communication / amount of computation
- Need low ratio to efficiently utilize modern parallel processors since the ratio of compute capability to available bandwidth is high

PARALLEL AND DISTRIBUTED COMPUTING

Artifactual communication

- Inherent communication: information that fundamentally must be moved between processors to carry out the algorithm given the specified assignment (assumes unlimited capacity caches, minimum granularity transfers, etc.)

- Artifactual communication: all other communication (artifactual communication results from practical details of system implementation)

PARALLEL AND DISTRIBUTED COMPUTING

Examples

- System might have a minimum granularity of transfer (result: system must communicate more data than what is needed)
  - Program loads one 4-byte float value but entire 64-byte cache line must be transferred from memory (16x more communication than necessary)
- System might have rules of operation that result in unnecessary communication:
  - Program stores 16 consecutive 4-byte foat values, so entire 64-byte cache line is loaded from memory, and then subsequently stored to memory (2x overhead)
- Poor allocation of data in distributed memories (data doesn’t reside near processor that accesses it most)
- Same data communicated to processor multiple times because cache is too small to retain it between accesses
- Data may be communicated multiple times, for example, every time the value changes, but only the last value may actually be used. On the other hand, data may be communicated to a process that already has the latest values, again because it was too difficult to determine this.

PARALLEL AND DISTRIBUTED COMPUTING

Improving locality (temporal)

- Temporal locality: A program is said to exhibit temporal locality if tends to access the same memory location repeatedly in a short time-frame
- Goal: structure an algorithm so that its working sets map well to the sizes of the different levels of the hierarchy
  - keeping working sets small without losing performance for other reasons

PARALLEL AND DISTRIBUTED COMPUTING

Keeping working sets small

- Assign tasks that tend to access the same data to the same process

- Once assignment is done, a process's assigned computation can be organized so that tasks that access the same data are scheduled close to one another in time

  - we reuse a set of data as much as possible, before moving to other data

PARALLEL AND DISTRIBUTED COMPUTING

Fusing loops

```
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,   *tmp;
// assume   arrays   are   allocated   here
// compute  D  =  (A +  B) *  C
add(n,   A, B, tmp);
mult(n,  tmp,  C, D);
```

- Two loads, one store per math op => C2CR = 3/1 = 3

PARALLEL AND DISTRIBUTED COMPUTING

Fusing loops

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

}
// compute  D  =  (A +  B) *  C
fused_muladd(n,  A, B, C, D);
```

- Three loads, one store, 2 math ops => C2CR = 4/2 = 2

PARALLEL AND DISTRIBUTED COMPUTING

Improve C2CR

- Exploit sharing: co-locate tasks that operate on the same data
  - Schedule threads working on the same data structure at the same time on the same processor
  - Reduces inherent communication
- Example: OpenCL working group
  - Abstraction used to localize related processing
  - Threads in block often cooperate to perform an operation (leverage fast access to / synchronization via shared memory)

PARALLEL AND DISTRIBUTED COMPUTING

Exploit spatial locality

- Granularity of communication / data transfer

PARALLEL AND DISTRIBUTED COMPUTING

Total communication cost = num_messages x (message communication time – overlap)

Total communication cost = num_messages x (overhead + occupancy + network delay – overlap)

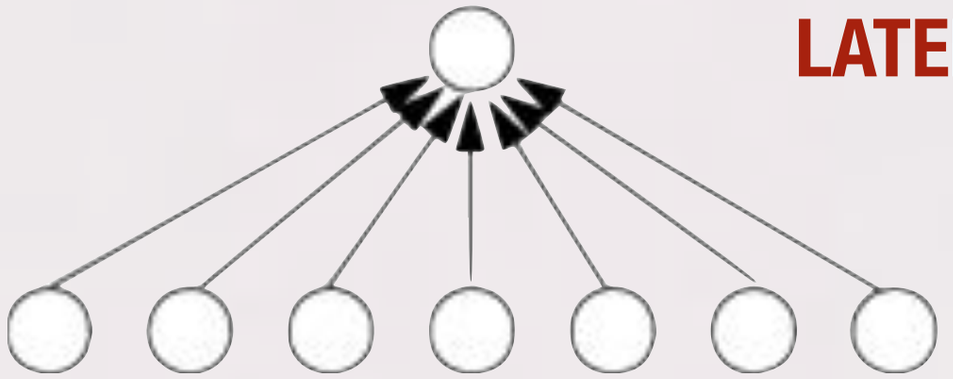
Total communication cost = num_messages x (overhead + **(T0 + n/B + contention)** + network delay – overlap)

Occupancy consists of the time it takes to transfer a message (n bytes) over slowest link + delays due to contention for link

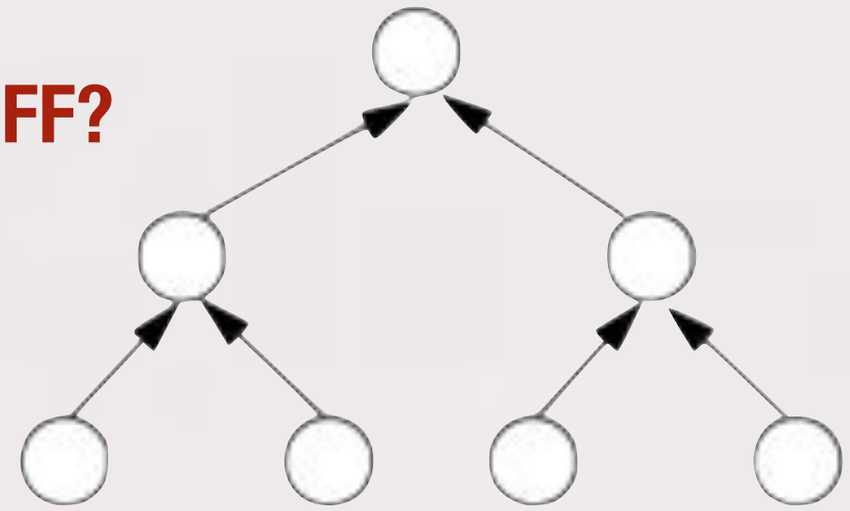
PARALLEL AND DISTRIBUTED COMPUTING

Contention

- A resource can perform operations at a given throughput (number of transactions per unit time)
  - Memory, communication links, servers, etc.
- Contention occurs when many requests to a resource are made within a small window of time (the resource is a “hot spot”)



LATENCY TRADE-OFF?



FLAT UPDATING OF SHARED VARIABLE

TREE STRUCTURE

PARALLEL AND DISTRIBUTED COMPUTING

Reducing communication cost

- Reduce contention
  - Replicate contended resources (e.g., local copies, fine-grained locks)
  - Stagger access to contended resources
- Increase 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)

PARALLEL AND DISTRIBUTED COMPUTING

Reducing communication cost (2)

- Reduce overhead to sender/receiver
  - Send fewer messages, make messages larger (reduce overhead)
  - Coalesce many small messages into large ones
- Reduce delay
  - Application writer: restructure code to exploit locality
  - HW implementor: improve communication architecture

PARALLEL AND DISTRIBUTED COMPUTING

In short …

- Inherent vs. artifactual communication
  - Inherent communication is fundamental: how the problem is decomposed and how work is assigned
  - Artifactual communication depends on the machine (often as important to performance as inherent communication)
- Improving program performance:
  - Identify and exploit locality: communicate less (decrease C2CR)
  - Reduce overhead (fewer, large messages)
  - Reduce contention
  - Maximize overlap of communication and processing (hide latency so as to not incur cost)

PARALLEL AND DISTRIBUTED COMPUTING