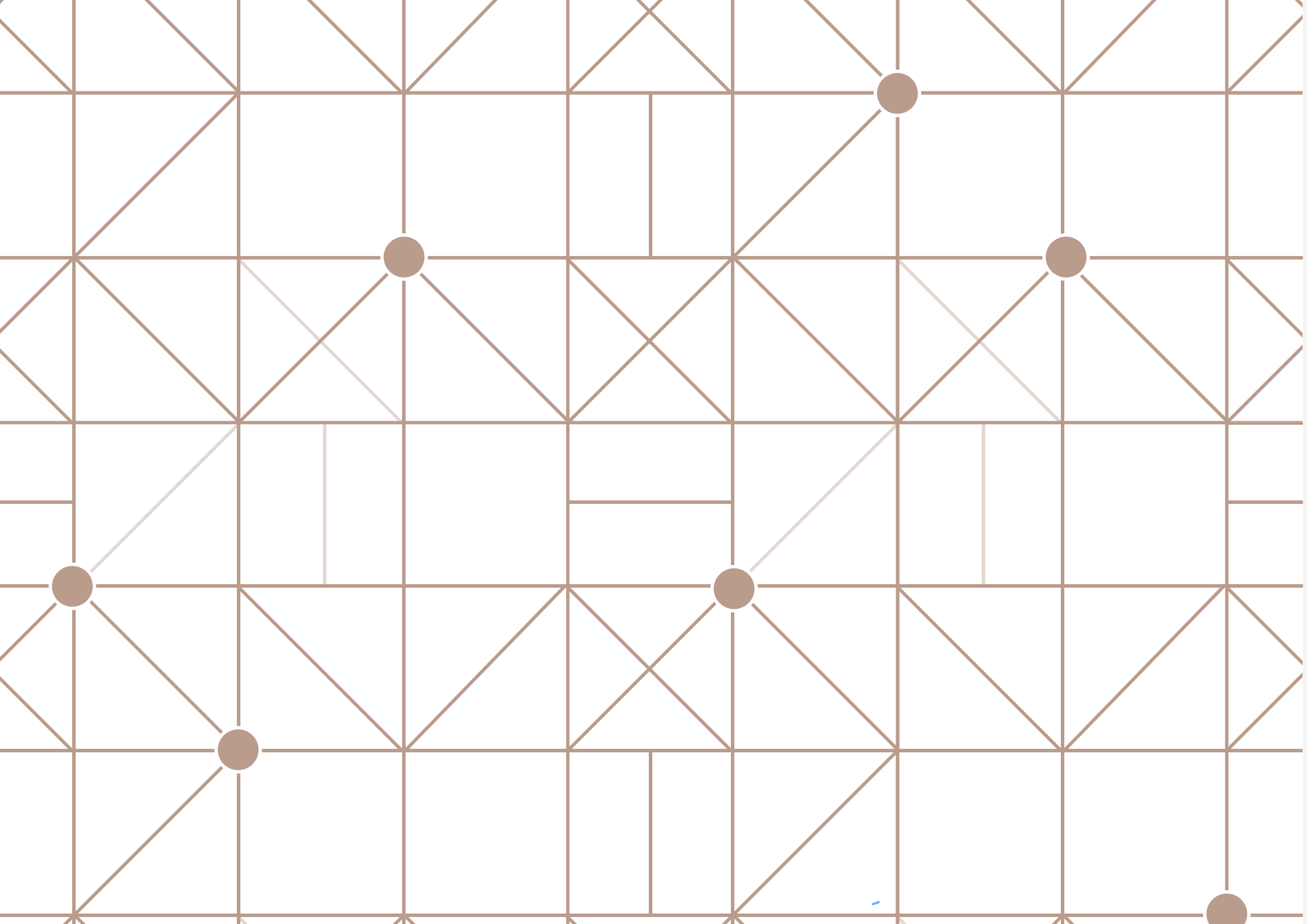
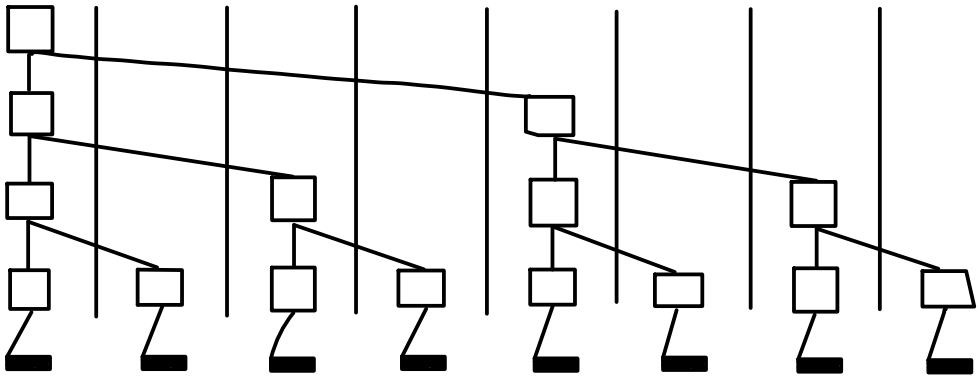


UNIT 3

SHWARTZ' ALGO

- An algo for tree operations — +- reduce
- His observation - tree structure should be used to connect the processes rather than the items.
- For a fixed no. of processes P & no of values n, P<n there are 2 approaches for computing +- reduce:
  (1) Introduce logical threads that implement a combining tree that literally encodes the soln. to summing in pairs w/ n/2 logical concurrency.
  (2) Have the processes add n/P of the items locally, & then combine the P intermediate sums w/ a P-leaf tree that connects the processes

- This algo illustrates the point — Unlimited Parallelism approach is inferior to the Scalable Parallelism approach.



All processes operate locally before using a tree to combine their results.
During the tree combination phase, the parallelism is uneven, as seen in the diagram

BASIC STRUCTURE OF SCAN & REDUCE

- As we need a scalable soln., we assume that the DS upon which the scan/reduce operates has been allocated in pieces to an unknown no. of processes.
- We use a Schwartz-like algo that computes locally & then uses a tree to complete the computation.
- Also assume the existence of a local variable in each process, tally, which is used to store intermediate results of the reduce & scan operations.
- with these assumptions, all reduce & scan ops can be implemented by defining variations of the following functions:

init() — Initializes the tally in prep. for the local computation.

accum() — Performs local accum. of the operand element into the tally.

combine() — composes interm. tally results from its 2 subtrees & passes the result to the parent.

x-gen() — Takes the global result & generates the final answer.

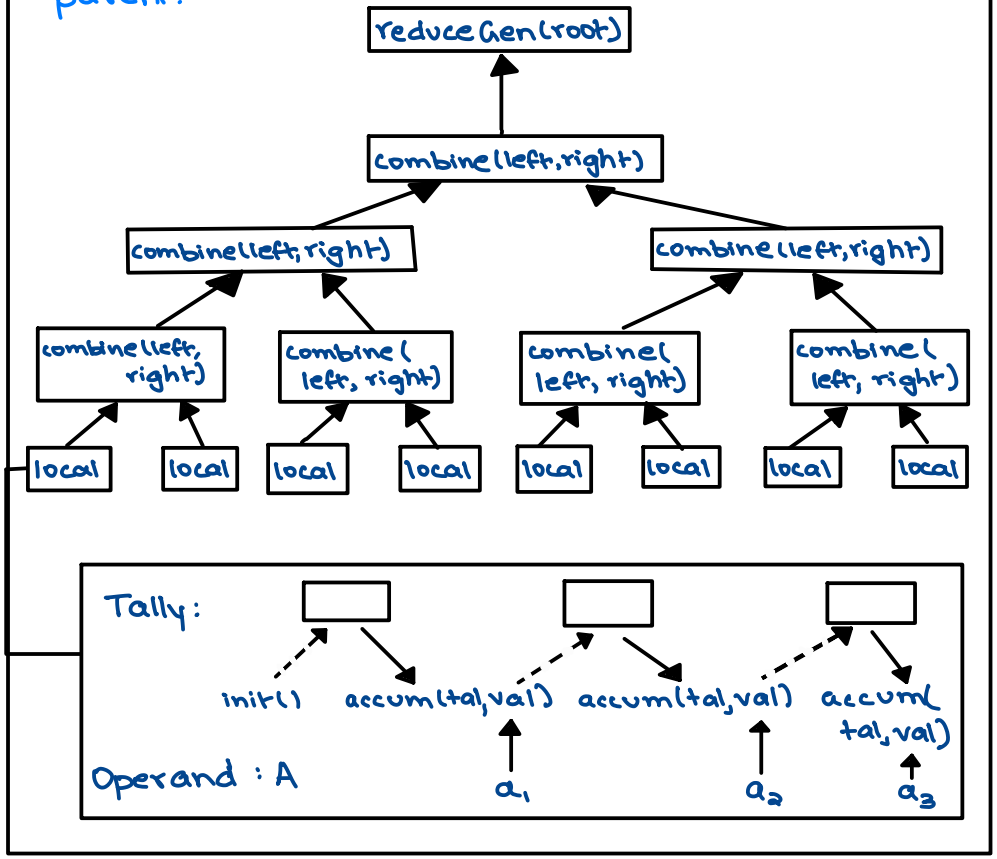
STRUCTURE FOR GENERALIZED REDUCE

init( )
sets up an integer (temp.) tally & initializes it to 0

accum(tally, val)
adds val, which is A[i] for some i to tally, & stores the result in tally.

combine(left, right)
adds left & right tally values & passes it to the parent.

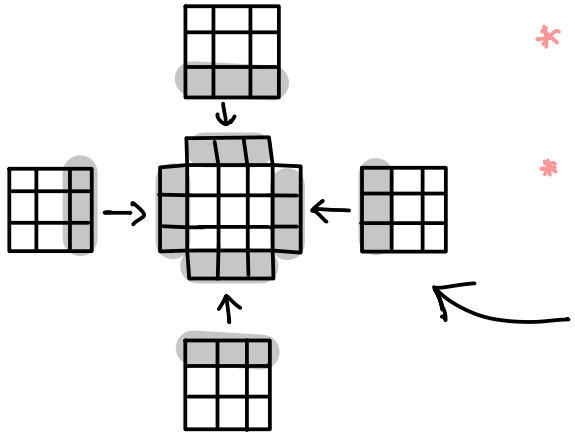
reduce-gen(root)
No-op for this simple operation.



ASSIGNING WORK TO PROCESSES STATICALLY

BLOCK ALLOCATIONS

- To exploit locality, for most computations contiguous portions of a data structure should be allocated on the same process
- 1D arrays are assigned to processes in blocks of consecutive indices.
- For 2D arrays — 2D blocks - consecutive indices in both dimensions.
- Blocks make more sense than allocating whole rows as they reduce communication.



OVERLAP REGIONS

- Type of software cache.
- Example → In many array-based computations, values are computed by referencing a fixed set of neighboring values — producing stencil computations — as their memory reference pattern is a stencil applied at each point.

  They're referenced as follows:

  * When allocating storage for the local portion, allocate extra space to hold the non-local values to be referenced. This extra space — overlap region.
  * Obtain these non-local values & save them in the overlap region.
  * Perform the computation on what are now all local data values.

Contd.

ADVANTAGES OF OVERLAP REGIONS

- All references in the computation are local, yielding large blocks of independent computation.
- Reduces commun. costs
- Merges multiple cross-process dependences into b dependences. (b → no. of neighboring processes)

CYCLIC & BLOCK ALLOCATIONS

- **Cyclic allocations** address the issue of load imabalance (amount of work is not proportional to the amount of data) by allocating elements to different processes in a round-robin fashion.
- However, cyclic distribution can increase communication costs by increasing the no. of cross-process dependences
- **Block cyclic allocations:**
  - → A block's dimensions doesn't have to divide the matrix's; the block is truncated where necessary.
  - → Each process receives blocks from throughout the matrix, so as the computation proceeds, there will be complete as well as incomplete portions on the processes.

IRREGULAR ALLOCATIONS

- Many algos use data structures other than arrays, like — unstructured grids & irregularly shaped grids typically made up of triangles.
- Assuming that interactions occur at grid boundaries, comm. b/w processes can be minimized by identifying large portions of the grid that have a large surface area to volume ratio. Techniques for such partitioning can be
  - based on geometric partitioning
  - " " graph theoretic "
- As the data references are irregular & often not known until runtime, it is inefficient to fetch the non-local values.
- To avoid this- **inspector/executor technique**

  ↓

  * Before executing an iteration or segment of code, the data references for its irregular data structures are "inspected", i.e, analyzed to identify those that are non-local & the process to which they are assigned.
  * All non-local references are batched & fetched.
  * Then, w/ all the data local, the executor performs the computation.

ASSIGNING WORK TO PROCESSES DYNAMICALLY

- In many cases, its impossible to adopt a fixed work assignment as new work is created during the computation.
- In other cases, static allocation leads to poor load balance

WORK QUEUES

→ It is a data structure for dynamically assigning work to processes.
→ They are particularly used when the work is dynamically generated during the computation.
→ The simplest work queue - FIFO list of task descriptors - newly created tasks are added to 1 end of the queue & tasks to be processed are removed from the other end.
→ Example - producer/consumer paradigm.

VARIATIONS OF WORK QUEUES

- Elements of the queue can be processed in FIFO, LIFO, randomized, or priority order.
- Specific uses of these queues can vary based on granularity of work.
  - L> A small grain-size increases queue manipulation overhead & increases the likelihood of contention; a large grain-size increases the chances of load imbalance
  - L> Use of variable grain size that is proportional to the no. of elements in the queue finds a balance b/w overhead & load imbalance
- Using multiple queues is a scalable approach that reduces contention but can increase latency.
- Multiple queues causes load imbalance. Soln. → Work Stealing - ler threads retrieve work from queues to which it wasn't assigned.

TREES

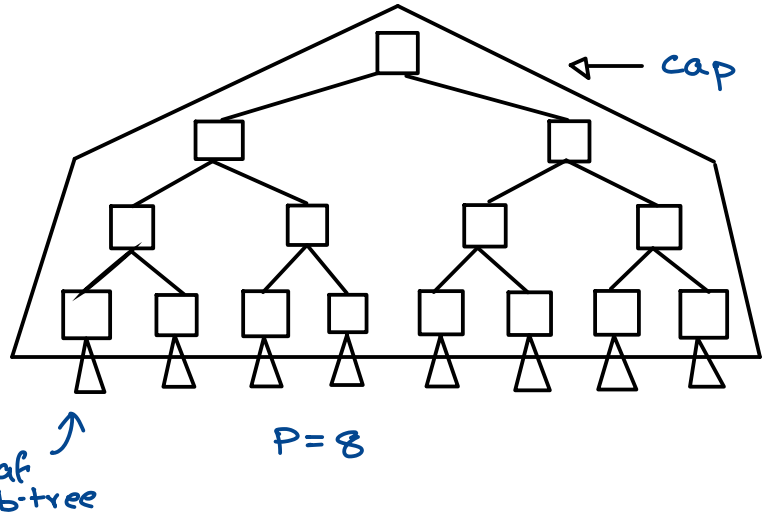
* Data structures that represent hierarchical organizations.
* They present challenges in parallel computing:
  - Trees are constructed using pointers & in languages that don't provide shared memory, pointers are local only to 1 process.
  - We use trees for their dynamic behavior, but this behavior often implies extensive performance-limiting communication.
  - The irregular structure of some trees can make it difficult to reason about comm. & load balance

ALLOCATION BY SUBTREE

* We could replicate the top of the tree, the "cap", & assign a copy of it & a sub-tree to each of the processes.
* Cap allocation is effective as it is small relative to the rest of the tree.

contd

contd.



- As the root & its immediate descendents are available on all processes, interactions that propagate through the root can use local data to identify the correct destination subtree.
- As computation proceeds, changes in the cap must be maintained coherently across processes.

DYNAMIC ALLOCATIONS

- Some trees are allocated dynamically in an unpredictable fashion, other trees are explored to uneven depths.
- Example → A search w/ alpha-beta pruning (in game searches) will prune away portions of the hierarchical search space based on previously seen results, leading to a search tree w/ leaves at different depths. In such cases a work queue can be used.

UNIT 4

POSIX THREAD CREATION

```
int pthread_create(
      pthread_t *tid,
      const pthread_attr_t *attr,
      void *(*start_routine)(void*)
      void *arg
   );
```

ARGUMENTS:

→ A thread ID of the successfully created thread

→ The thread's attributes; the NULL value is default

→ The function that the thread will execute once it is created.

→ An argument to the start_routine

Return value - 0 if successful, error code from <errno.h> otherwise.

Notes: use a structure to pass multiple arguments to the start routine.

MUTEX CREATION & DESTRUCTION

```
int pthread_mutex_init(
   pthread_mutex_t *mutex);

int pthread_mutex_destroy(
   pthread_mutex_t *mutex);
```

POSIX THREAD TERMINATION

```
pthread_exit(
   void *status
);
```

ARGUMENTS:

→ The completion status of the thread that has exited. This pointer value will be available to other threads.

Return value: none

Notes: When the thread exits by returning from the start_routine, the thread's completion status is set to the start_routine's return value.

SYNCHRONIZATION (mutex)

- A condition variable allows threads to wait until some condition becomes true, at which point one of the waiting threads is nondeterministically chosen to stop waiting.

BOUNDED BUFFER PROBLEM

→ w/ Producers & Consumers

→ when the buffer is empty, the consumers must wait. when " full, producers " ".

MUTUAL EXCLUSION

- To allow threads to interact constructively, we need methods for coordinating their interaction.
- In particular, when 2 threads share access to memory, its useful to employ a lock, mutex, to provide mutually exclusive access to the variable.
- To use the mutex, its address is passed to the lock & unlock routines
- Bracket all critical sections - i.e; that code must be executed atomically by only a thread at a time - by the locking of the mutex upon entrance & unlocking upon exit.
- When a mutex is unlocked, 1 of the threads that was blocked while attempting to acquire the lock will become unlocked & will be granted the mutex.

```
int pthread_mutex_lock(          // Lock a
   pthread_mutex_t *mutex);      mutex
int pthread_unlock(              // Unlock a
   pthread_mutex_t *mutex);      mutex
```

SAFETY ISSUES (MUTEX)

- Double-locking: A problem which occurs when a thread attempts to acquire a lock that it already holds.
- Problems also arise if a thread accesses some shared variable wlout locking it first, or if a thread doesn't relinquish a lock.
- Important problem - DEADLOCK.

DEADLOCK:

The necessary conditions for deadlock:

* Mutual Exclusion - a resource can be assigned to at most 1 thread
* Hold & wait - threads hold some resources & request other resources.
* No preemption - a resource that is assigned to a thread can only be released by the thread that holds it.
* Circular wait - a cycle exists in which each thread waits for a resource that is assigned to another thread.

Approaches to dealing w/ deadlock:
(1) prevent deadlocks
(2) allow them to occur, but detect & break them.

POSIX threads don't provide a mechanism for breaking locks, ∴ there's only deadlock avoidance here.

- The bounded buffer problem can be solved using 2 condition variables — nonfull & nonempty.
- Since multiple threads will be updating these condition variables, we protect their access w/ a mutex.

① **LOCK HIERARCHIES**:
→ Preventing cycles in the resource allocation graph can prevent deadlocks.
→ We can prevent cycles by imposing an ordering on the locks & by requiring that all threads acquire their locks in the same order
→ A lock hierarchy requires programmers to know a priori what locks a thread needs to acquire.

② **Monitors**:
→ A monitor encapsulates code & data & ensures mutual exclusion.
→ A monitor has a set of well-defined entry points, its data can only be accessed by code that resides inside the monitor, & only one thread can execute the monitor's code at any time.
→ The limited no. of entry points facilitates the preservation of invariants.
They (↑) are properties that are assumed to be true upon entry & that must be restored upon exit.

③ **Re-Entrant Monitors**:
→ while monitors help enforce a locking discipline, they don't solve all concurrency problems.
→ Eg: if a procedure in a monitor attempts to re-enter the monitor by calling an entry procedure, deadlock will occur.

PERFORMANCE ISSUES

READERS & WRITERS EXAMPLE: GRANULARITY ISSUES

- There are different granularities for using condition variables.
- Consider a resource that can be shared by multiple readers or accessed exclusively by 1 writer.
  To coordinate access to such a resource, we can implement 4 routines that readers & writers can invoke — AcquireExclusive(), ReleaseExclusive(), AcquireShared(), & ReleaseShared(). These routines are each protected by a single mutex, & they collectively use 2 condition variables. Condition variable — wBusy & rBusy

- Only 1 condition variable could be used but the code would suffer from spurious wakeups in which writers can be awakened only to go back to sleep immediately.

THREAD SCHEDULING

- A kernel thread is a unit of scheduling w/in the kernel or OS.
  Threads can be mapped to kernel threads in 2 primary ways:

① We could map multiple threads to 1 kernel thread. Such threads can be created & destro-yed w/out OS intervention. Problem — if 1 thread blocks, the entire kernel thread blocks, & none of the other threads can execute. unbound threads

② Map each thread to its own kernel head. Here, the OS is aware of all threads, so when 1 thread blocks, another thread can be scheduled to execute in its place. They're called bound threads.

PRIORITY INVERSION

- Occurs when a low-priority thread holds a lock that a high-priority thread wishes to acquire.
- A more serious case occurs if a 3rd thread is created w/ a priority that is b/w the low- & high-priority threads. This medium-priority thread will preempt the low-priority thread. Thus, the high-priority thread must block as long as the medium-priority thread executes.

SOLUTIONS:

① Priority Ceiling - highest possible thread priority. Any thread that acquires a lock executes at this priority, which prevents some medium-level thread from preempting it.

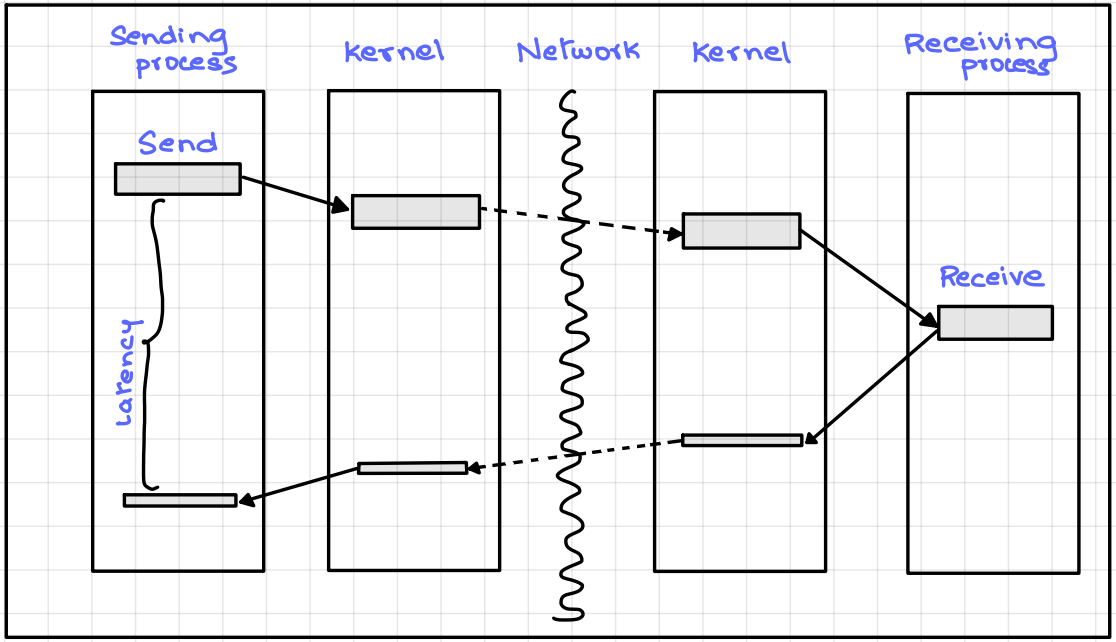
② Priority Inheritance - when a thread acquires a mutex, it temporarily inherits the priority of the highest thread that is blocked waiting for that mutex.

UNIT 5

POINT TO POINT COMMUNICATION

- In MPI, point-to-point communication is specified redundantly by the sending & receiving process.
- Messages are matched based on the operations' specified source/destination process & the message tag, which is a non-negative integer specified by the user.
- Thus, tags can be used to distinguish logically different messages b/w the same pairs of processes.
- In some cases, the process may not know which process it wishes to communicate with, in which case it can specify MPI_ANY as the source/destination.
- MPI guarantees that messages b/w the same source & destination processes will be delivered in order.

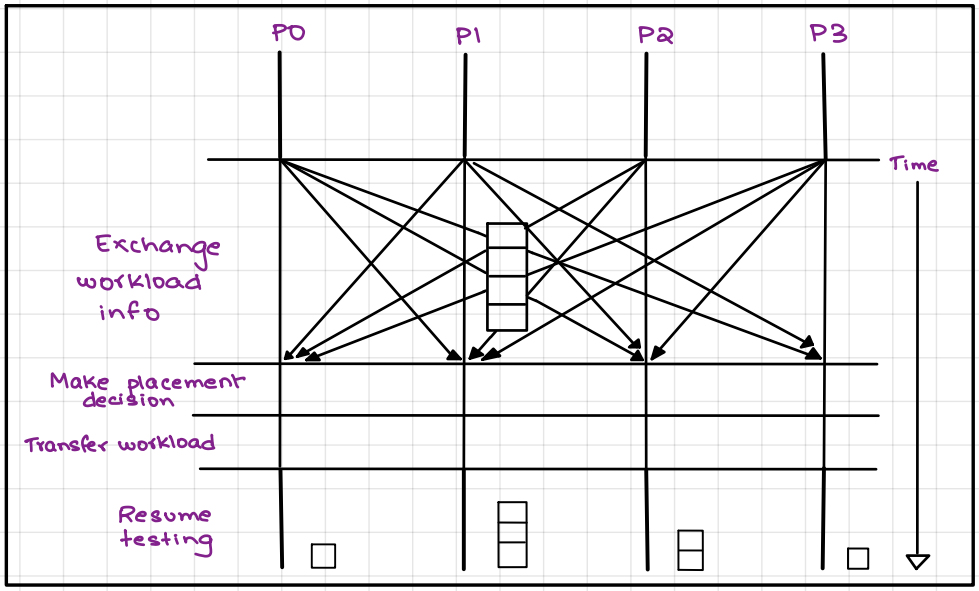
- MPI provides many flavors of point-to-point communication.
  To understand why, realize that there is significant synchronization & data copying that must occur; for each message, the data must be copied across 4 address spaces.



- This figure focuses on the most basic data transfer operation; it ignores implementation-specific details that might require additional buffering & handshaking.

PERFORMANCE ISSUES

- Regardless of the latency of the underlying hardware communication infrastructure, MPI messages incur large overheads for each message sent.
- Thus, we typically want to reduce the no. of messages sent by:
  (1) choosing algos that produce few cross-process dependences, &
  (2) combining multiple messages into a single larger message.
- If we wish to send the elements of an arbitrary array, it gets complex; how do the sender & receiver agree on the message length when only the sender knows the length?
  One solution is to use a protocol in which the sender 1st sends the length to the receiver, but this doubles the no. of messages.
- Because of the large startup cost of each message, it is often useful to overlap communication w/ computation



- The large overhead of messages & the fixed set set of MPI processes imply that MPI programs are static in nature.
- However, as the figure shows, MPI programs can perform dynamic work distribution.
- The figure shows a set of 4 processes that periodically exchange info about the state of each process' workload, use this info to locally compute the desired work distribution for the next set of iterations, transfer work accordingly, & then continue w/ the computation.
- Such work distribution is only feasible if the benefit of the improved load balance outweighs the cost of this expensive redistribution protocol.

SAFETY ISSUES

- In MPI, there is no shared data, so there is no need to provide explicit mutual exclusion.
- Nevertheless, MPI programs can be difficult to write because so many low-level details are left to the programmer.
  Eg → Programmers must specify messages redundantly at both the sending & receiving process.
- Thus programmers must ensure that sends & receives are properly matched, which includes tags, types, ordering of messages & details of the message length.
- Moreover, the more efficient variants of messages – such as non-blocking msgs – are typically the more difficult to use. They make additional assumptions about the timing & buffering of messages, assumptions that must be enforced by the programmer.

UNIFIED PARALLEL C

- It gives the programmer a global view of the address space.
- When UPC array variables are declared "shared" they are distributed across the memories of program instances such that the linearly ordered array elements are distributed in a cyclic or block cyclic arrangement
- Cyclic & block cyclic distributions can be advantageous when load balancing is a concern, but they generally reduce locality.
- The UPC programmer can regain locality by directly allocating portions of the array to processes, thereby ensuring that the dense neighborhoods are allocated.
- Since UPC extends C, it supports pointers.

UPC pointers:

		Property of the pointer	
		Private	Shared
Property of the refer-ence	Private	Private-Private, p1	Private-Shared, p2
	Shared	Shared-Private p3	Shared-Shared, p4

- These properties are associated w/ the language's type system, as their declarations indicate.

```
int *p1                    ; private ptr, pointing locally
shared int *p2             ;    "      "  , "  to a shared space
int *shared p3             ; shared ptr, "  locally
shared int *shared p4 ;  "    ",  "shared space
```

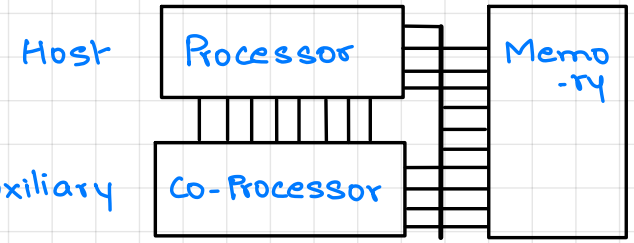
- The upc_forall statement is a global operation, whereas most UPC code executes locally within a process

TITANIUM

- The Ti language is an extension of Java that executes on distributed memory parallel computers.
- It has a memory model similar to UPC, & like the other PGAS languages, it generates communication code based on a one-sided communication library.
- The 1st obvious quality of Ti that distinguishes it from its predecessors is its object-oriented nature.
- It also distinguishes itself from Java by adding regions, which support safe performance-oriented memory management as an alternative to garbage collection.
- Several other Java features have been restricted or limited, but more typically features have been added.
  Eg→ 2D arrays have been added to make Ti more suitable for scientific computation.
- An important feature of Ti is its unordered iteration foreach, which simplifies both the programmer's & the compiler's job.
  The key feature is a point – a tuple of integers than can range over the domain of a variable.
- Compared to forall, the foreach allows concurrency over multiple indices within a single block.
- Ti enforces global synchronization using barriers & the concept of a shared variable called single that is shared across all processes. The barrier ensures that all processes stop computing at the same time to update & read memory.

ATTACHED PROCESSORS

→ Special-purpose processors can be much more power & space efficient than general-purpose processors, so it makes sense to offload as much work as possible to special purpose attached processors.
→ The attached processor is performing work on behalf of the host processor.



→ The fact that the host processor runs with the attached processor is typically not a significant source of parallelism. Instead, there is generally significant parallelism embedded in the attached processor.

① GPUs

→ The idea of performing general purpose computing on graphical processing units has gained recent popularity due to the tremendous performance & price of GPUs.
→ Improvements in GPUs have been increasing at a faster rate than improvements in general-purpose microprocessors.
Reasons why GPUs offer such good performance:

- Economically, their performance is driven by the large video game industry.
- Technically, a GPU is specialized for graphics rendering- a compute-intensive appⁿ w/ large amounts of data parallelism.

TRANSACTIONAL MEMORY

→ A database transaction modifies data to ensure 4 properties — ACID.

Atomicity: All operations in the transaction complete, or none complete.

Consistency: Storage is updated as if the operations were completed in some serial order.

Isolation: A transactions' operations are unaffected by any other.

Durability: Changes made by a transaction persist

→ In a TM system, the programmer identifies transactions, & the system — implemented in either hardware or software — enforces the semantics of a transaction by tracking their loads & stores to memory & detecting any conflicts that would violate the transac-tional properties.

→ If a transaction completes successfully, it is said to have committed; otherwise the transaction aborts & none of its side effects are visible.

→ TM simplifies programming because a transaction gives the illusion of serial execution wlout the possibility of interactions from other threads.

→ TM provides many benefits — transactions are scalable, composable, deadlock free, & easy to use.

- GPUs can largely ignore many issues that general purpose processors must deal with, like the need to keep pipelines full for sequential codes.

→ GPUs implement hardware support for the graphics pipeline, which takes as input a list of vertices that define geometry & emits as output an image in a framebuffer.

(2) Cell Processors

→ The goal of the Cell Processor is to produce a powerful device to support interactive video games.

→ The Cell architecture contains a std PowerPC, including L1 & L2 caches, & 8 SPEs (synergistic processing elements).

→ The SPEs are powerful SIMD machines capable of executing 4 single-precision floating-point operations per cycle.

→ The EIB (element interconnect bus) supports an enormous amount of on-chip communication.

→ The Cell is programmed using C/C++ under Linux.

→ The Cell system has excellent bandwidth to main memory.

IMPLEMENTATION ISSUES OF TM

- A TM system must perform 2 basic operations — it must detect when memory accesses from different threads conflict (conflict detection), & it must provide a mechanism for isolating the effects of a transaction until it either commits or aborts (data versioning).
- Conflict detection checks whether 2 threads access the same memory location, with at least 1 of the operations being a modification of that location.
- To detect such conflicts, the TM system must keep track of the read set & write set for each transaction.

Conflict Detection

Pessimistic Conflict D

- This scheme detects conflicts as early as possible, to avoid
- wasted work.

Upon detecting a conflict, it can either abort a transaction immediately, or it can stall 1 of the transactions.

Optimistic Conflict D

- This scheme assumes that conflicts are rare, so it delays conflict checks until the end of a transaction, at which point the transaction either aborts or commits.

- Data versioning maintains multiple versions of data so that transactions can be either rolled back or committed.

Data Versioning

Eager versioning

- The system saves copies of the old version of data so that it can write the new values to main memory.
- Aborts are slower as they must restore the transactions' state by walking over the list of saved values.

Lazy versioning

- New values are saved to a write buffer while the old version is left untouched.
- Aborts are fast, while commits must copy values from the write buffer

4 IMPORTANT PROPERTIES OF PARALLEL LANGUAGES

① CORRECTNESS

* Though it is significant to write correct programs of any type, the problem is worse for parallel programs, which are often sensitive to timing features of the program execution.
* Sequential programs are generally reproducible, so 2 executions w/ the same i/p produce the same output; but parallel programs can vary wildly in their behaviour depending on timing variations.
* A parallel program is **P-independent** if & only if it always produces the same output on the same input regardless of the number or arrangement of processes on which it is run; otherwise a program is **P-dependent.**
* The notion of P-independence can be translated to languages by classifying programming abstractions into 2 groups:

Global view abstractions - A language construct that preserves P-independent program behavior presents a Global view abstraction.

local view abstractions - A language construct that doesn't preserve P-independent program behavior presents a local view abstraction.

② PERFORMANCE

* How much performance is enough? The answer depends on the context.
* Eg, on a 2-processor multi-core system, there are different situations where programmers might be content w/ speedups of 1.2, 1.9, or 2.1:
  - → A modest speedup of 1.2 may be satisfactory in cases when the appⁿ in question has little inherent parallelism, & the goal was to exploit an otherwise idle processor.
  - → A speedup closer to 2 is reasonable when a programmer has carefully structured the program to eliminate most parallel parallel overheads & to exploit available concurrency.
  - → Superlinear speedup is possible when the computation exhibits good locality of reference.
* In evaluating performance, we need to understand the situation & have reasonable performance expectations.

③ SCALABILITY

* Performance is not only appⁿ-dependent, but also hardware-dependent. In particular, as we increase the no. of processors, it is harder to sustain the same level of speedup.
* Thus, in many cases scalable parallelism may seem to be an unnecessary goal, particularly when the short-term objective is to run on a specific hardware platform.

contd. ↓

HIDDEN PARALLELISM

-> Parallelism has long been successful as it has been underground & hidden from programmers.

-> The prime example is the parallelism exploited by the micro-architecture of contemporary processors. Consider the following features that deliver hidden parallelism:

- multiple functional units
- Pipelined execution
- Out of order execution
- DMA controllers
- Prefetch units
- Vector processors
- Chip multi-processors
- Co-processors

-> The underlined items expose parallelism to the software. All others are largely hidden from the programmer.

-> Over time the trend has been toward ever-larger units of parallelism, to the point that we are no longer able to hide parallelism w/in the hardware.

-> Hidden parallelism is important as it hides complexity, which is a powerful tool in system design.
The trick is to hide complexity wlout obscuring performance costs.

④ PORTABILITY

* Performance portability - the ability of a program to obtain good performance, wl some modest amount of tuning, on a wide range of parallel computers.
* Performane portability can be achieved by desig-ning algos based on a realistic abstract machine model (CTA).
* The abstract model limits assumptions to universal concepts & suggests costs that are realistically achievable on machines that scale.
* By programming to a realistic abstract machine model, programmers can move their code to different architectures wl the expectation that the program's core properties will be compatible wl the new hardware; tuning may be needed, but the algos do not need to be completely rethought.

TRANSPARENT PERFORMANCE

- It is important to allow programmers to reason about performance as they develop their algos & programs.
- Languages of any level can hinder this ability to reason.
- The ability to reason about performance can be supported by either low-level or high-level approaches.

LESSONS FOR THE FUTURE