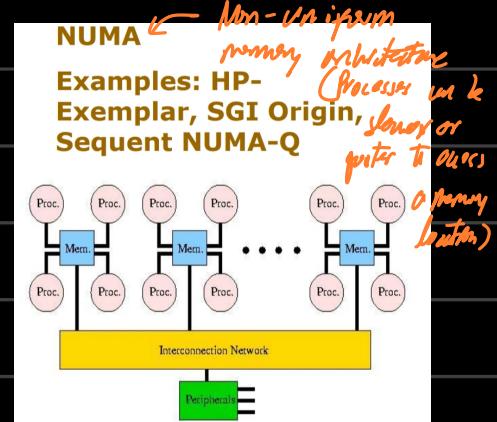
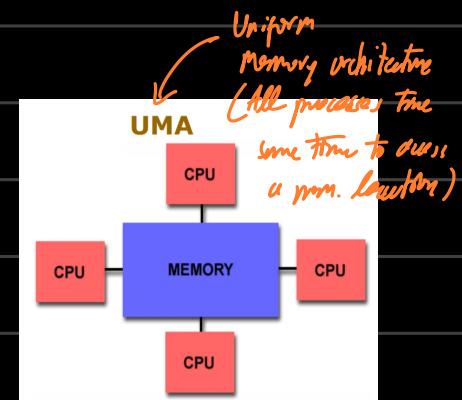
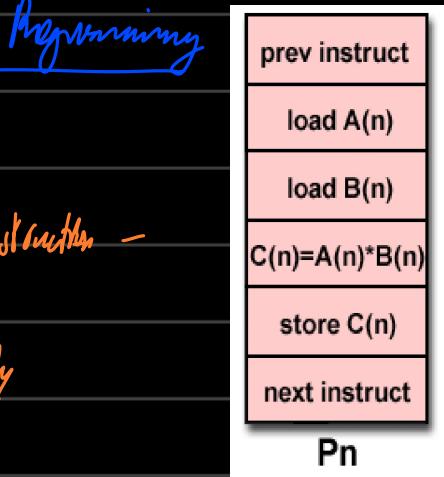
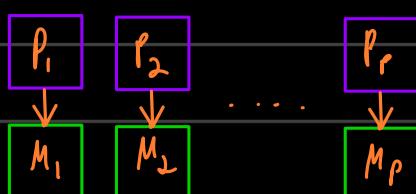


Introduction to Scalable Systems - Parallel Programming

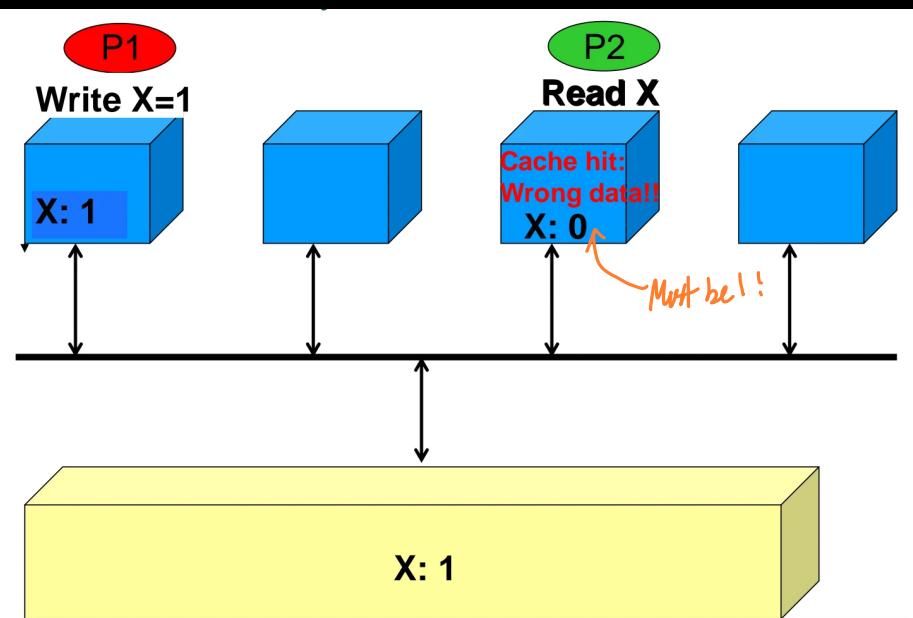
Classifications of Architectures

- Flynn's classification
- In terms of parallelism & data stream
- SISD: Single instruction single data
 - Serial programming
- SIMD: Single instruction multiple data
 - Vector processes & processor arrays
 - Issue & specialized SIMD instruction, and a instruction shall be carried out simultaneously.
- MIMD: Multiple instruction multiple data
 - Most popular for supercomputers etc.
- Classification based on memory:
 - Shared memory : UMA & NUMA
 - Shared memory implies that address space is shared, it does not mean that a single memory is shared.
- Message passing architecture
 - A process explicitly sends messages to another process.



- Shared memory architecture
- The shared memory itself could be distributed among processor "nodes"
- If all processors need to access a memory element x in M_2 , P_1 will quickly get it because it is closer. However, P_2 will be slower because it is further away. This is in NUMA.

Shared memory & Coherency



- If X has been updated by P_1 to 1, P_2 will still be reading the old value.
- Hence, we need to synchronise the writes across processors.

- Solutions:
- Write update: Propagate cache line to all other processors on cache write.
- Writes are costlier.
- Cache line is sent to all other processors, thereby leading to waste of time & resources.
- Write invalidate: Inform other processors that they have 'stale' data, and to read from memory next time.
 - Reads are costlier.
 - Followed in all systems.
 - Dirty bit needs to be maintained.

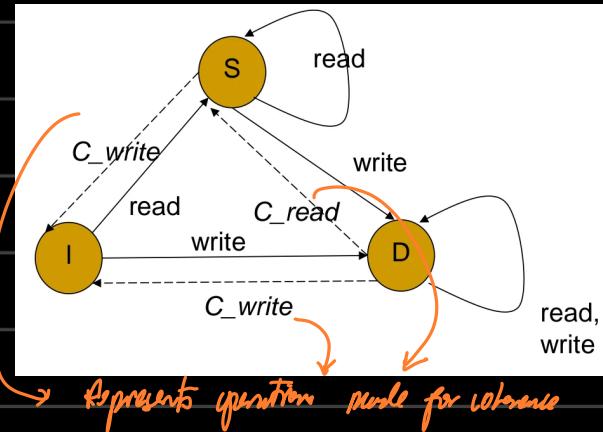
Cache Coherence by Write Invalidate

→ 3 states possible :

→ Shared : This memory is shared

→ Invalid : Some other processor has modified this

→ Dirty : I have modified this cache line.



Implementations of Cache Coherence Protocol

→ Snoopy

→ Used with bus-based architectures

→ Shared bus interconnect where all cache controllers monitor all bus activity

→ There is only one bus operation at a time, cache controllers can be built to take corrective actions & enforce coherence in caches.

→ Memory operations are performed

→ Directory based

→ A small part of shared memory is used to maintain cache states.

→ Instead of broadcasting memory ops. to all processors, propagate it only to the relevant processor.

→ Maintain cache states and a list of "relevant processors" in a central directory.

→ This will increase the time for memory access.

Implementation of Directory Based Protocols

- Using presence bits for all owner processes.
- True schemes:
 - Full bit vector scheme
 - $O(M \times P)$ storage for P processors & M cache lines.
 - This much storage is not necessary, so we will be a very sparse matrix.
 - Sparse / Tagged directory scheme
 - Used by modern day processors
 - Limited cache lines & limited processor bits.
 - Instead of storing bits for each processor, we store pointers to the processor no. (like linked list)

total cache lines across
all processors

Falshe Sharing

- Cache coherence occurs at the granularity of cache lines.
- Because a cache line can contain multiple data elements.
- If multiple processes share the cache line, but not the individual data elements.
- Hence, even if two processes have nothing to do with each other, they may still be considered to be sharing.
- E.g. Two processes, on working on two cols. of the same matrix (2-D array).
- This can be solved by better programming.
- Solutions:
 - Reorganize the code s.t. each processor work on individual rows rather than on individual columns. This may still lead to false sharing as two rows may share a cache line (if row size < cache block size)

→ This can be further solved by padding (adding dummy elements to ensure that row size = cache block size).

Used to connect processors
to memory

Interconnect

Used to connect

different processors

14/10/25

network of connections

→ Used in both shared & distributed memory architectures.

→ Interface (PCI / PCI-e): For connecting a processor to a network link.

→ Communication network

→ Consists of switching elements connected with each other using a pattern of connections.

→ Pattern defines network topology.

→ In shared memory systems, memory units are also connected to this network.

→ Network topologies: can be static / dynamic

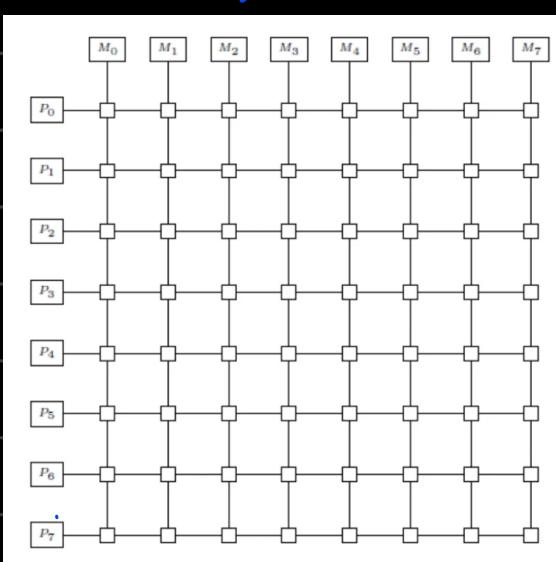
Network Topologies

→ Bus / ring : Used in small-scale shared memory systems.

→ Crossbar switch : Used in small-scale shared memory / medium-scale distributed memory.

→ Consists of 2-D grid of switching elements, each switching element has 2 ips & 2 oips

→ P^2 switches shall be used.

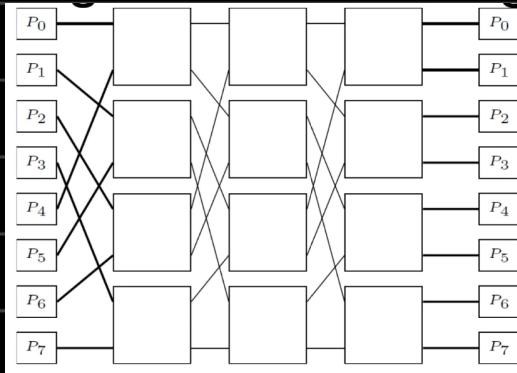


→ Multi-stage network (Omega)

→ To reduce switching complexity

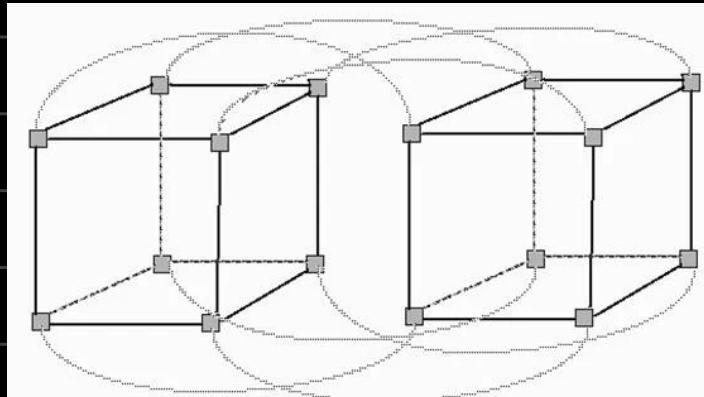
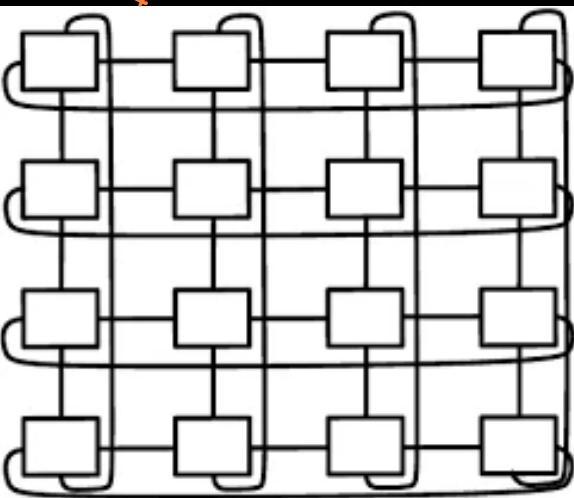
→ Consists of log P stages, and each consists of $P^{\frac{1}{2}}$ switching elements

→ Crossbar is non-blocking whereas Omega is blocking



→ Mesh, torus, hypercubes & fat tree are commonly used network topologies
is distributed memory architectures.

→ Hyper cubes are networks with dimensions.



Four-Dimensional Hypercube

→ Fat-tree network

→ Binary tree, with

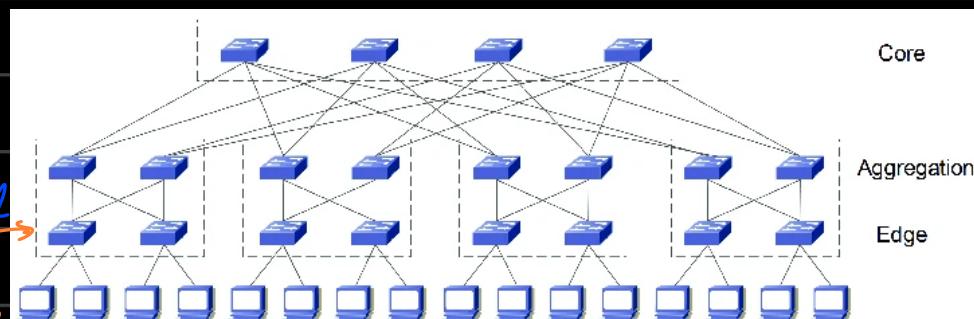
processors arranged
in leaves

→ Other nodes

correspond to switches.

→ Edges become fatter as we go up the tree.

→ Any pair of processors can communicate with each other w/o blocking



- Evaluation metrics for interconnection topologies
- Diameter: max. distance b/w any two processing nodes.
 - Connectivity: multiplicity of paths b/w 2 nodes
 - Min. no. of links to be removed to disconnect the network
 - Bisection width: Min. no. of links to be removed to split the network into two equal halves
 - Channel width: No. of bits that can be simultaneously communicated over a link, i.e. no. of physical wires between 2 nodes
 - " rate : performance of a single physical wire
 - bandwidth: Channel width * channel rate
 - Bisection " : min. volume of communication b/w two halves of the network, i.e. bisection width * channel bandwidth
- most important metric of a supercomputer.*

Type of network	Diameter	Type of Network	Connectivity
Full-connected	1	Linear array	1
Star	2	Ring	2
Ring	$P/2$	2-D mesh	2
Hypercube	$\log_2 P$	2-D mesh (with wraparound)	4
		D-dimension hypercube	10

Type of network	Bisection width
Ring	2
P-node 2-D mesh	\sqrt{P}
Tree	1
Star	1
Completely connected	$P^2/4$
Hypercubes	$P/2$

Many core architectures

- GPU & CPU
 - Less computationally intensive part runs on CPU, and more intensive part run on GPU. (fine-grained parallelism)
 - Basically CPU outsources this work to GPU.
- CUDA (Compute Unified Device Architecture): NVIDIA's GPU architecture, used with CUDA C language



NVIDIA A100

- 7 Graphics Processing Clusters (GPCs), each GPC has 7/8 texture processing clusters (TPCs) and up to 64 streaming multiprocessors (SMs) upto 68SMs

→ Each SM has:

→ 64 FP-32 GPU cores

→ 4 tensor cores

→ 192 KB of unshared shared memory & L1 data cache

Specialized cores for matrix ops.
(matrix multiplication, accumulation - MMA). Used extensively for AI & HPC applications.

coarse-grained parallelism



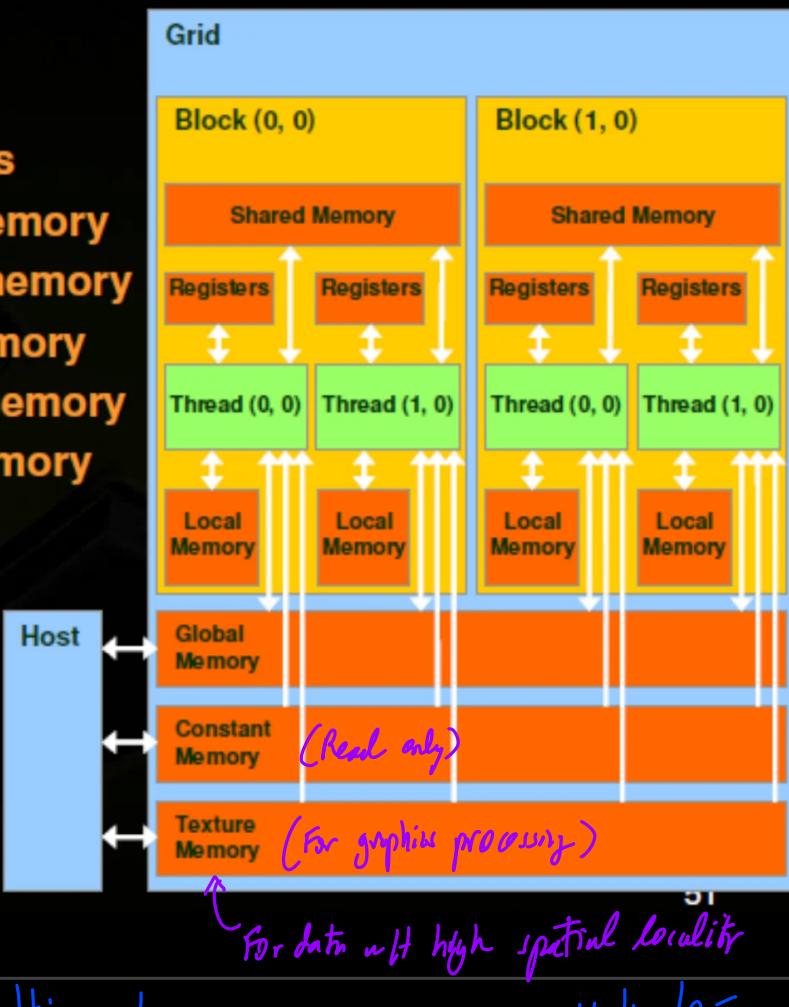
CUDA Memory Spaces



Each thread can:

- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

The host can read/write global, constant, and texture memory (stored in DRAM)



Memory Hierarchy

16/10/25

→ Global / device memory

→ Can be accessed by all the threads executing in the SMX.

→ " " " " the CPU host

→ Kepler K-40: 12 GB, A100: 80 GB

→ Shared memory :

→ In each SMX

→ Shared by all threads executing in an SMX .

→ Kepler K-40: 64 KB (can be configured as 16/32/48 KB for shared mem., rest for L1 cache)

→ A100 - 192 KB

- Latency of data access :
 - Device memory : 200 - 400 clock cycles
 - Shared " : 20 - 40 " "
- Differences with CPU threads
 - Fast context switching : The state of a thread stored in shared memory & registers stay till execution completion (zero overhead context switching)
 - Cache explicitly managed : User program has to explicitly bring the frequently accessed data from device to the shared memory. (Most of the space is usually reserved for shared memory)

- Triple parallelisation
 - Parallelisation across nodes
 - " in a single node
- A combination of CPU & GPU

Parallelisation Principles

- Parallel programs incur overheads over sequential programs :
 - Communication delay → latency → Synchronisation.

↗
to prevent this, ensure that the parallel program is load - balanced

Evaluation metrics

→ Speedup

Let T_p = execution time ,

$$\text{Speedup } S(p, n) = \frac{T(1, n)}{T(p, n)}$$

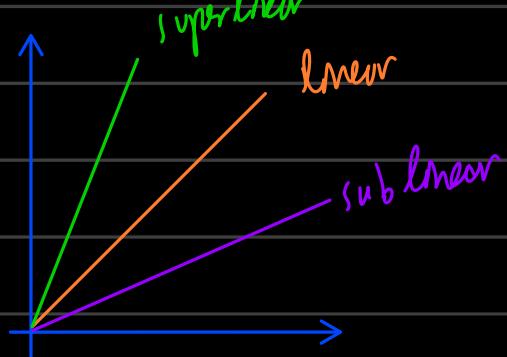
↑ no. of processes
↓ no. of instructions

→ Ideally, $S(p, n) = p$, i.e. time is equally split between all processors.

→ Usually, $S(p, n) < p$, due to overheads of parallel programming

→ Sometimes, $S(p, n) > p$! Superlinear speedup!

→ Happens when we get additional benefits due to parallelism:
→ Better cache locality.



→ Efficiency, E: $E(p, n) = \frac{S(p, n)}{p}$

→ Scalability: Limitations in parallel computing, reln. to $n \in p$

→ If a program works well for 2 processor, what is the guarantee that the program will work well for 100 processors also?

→ Limitations on speedup - Amdahl's law:

→ If the performance improvement to be gained from using some faster mode of execution is limited by the fraction of time the faster mode can be used:

→ Places a limit on speedup:

$$S(p, n) = \frac{1}{f_S + \frac{f_P}{P}}$$

→ This is because there will always be some fraction of the code that cannot be parallelised, and must have to be performed sequentially.

Sequential processing :-

Sequential

Parallelizable

Parallel

"

:

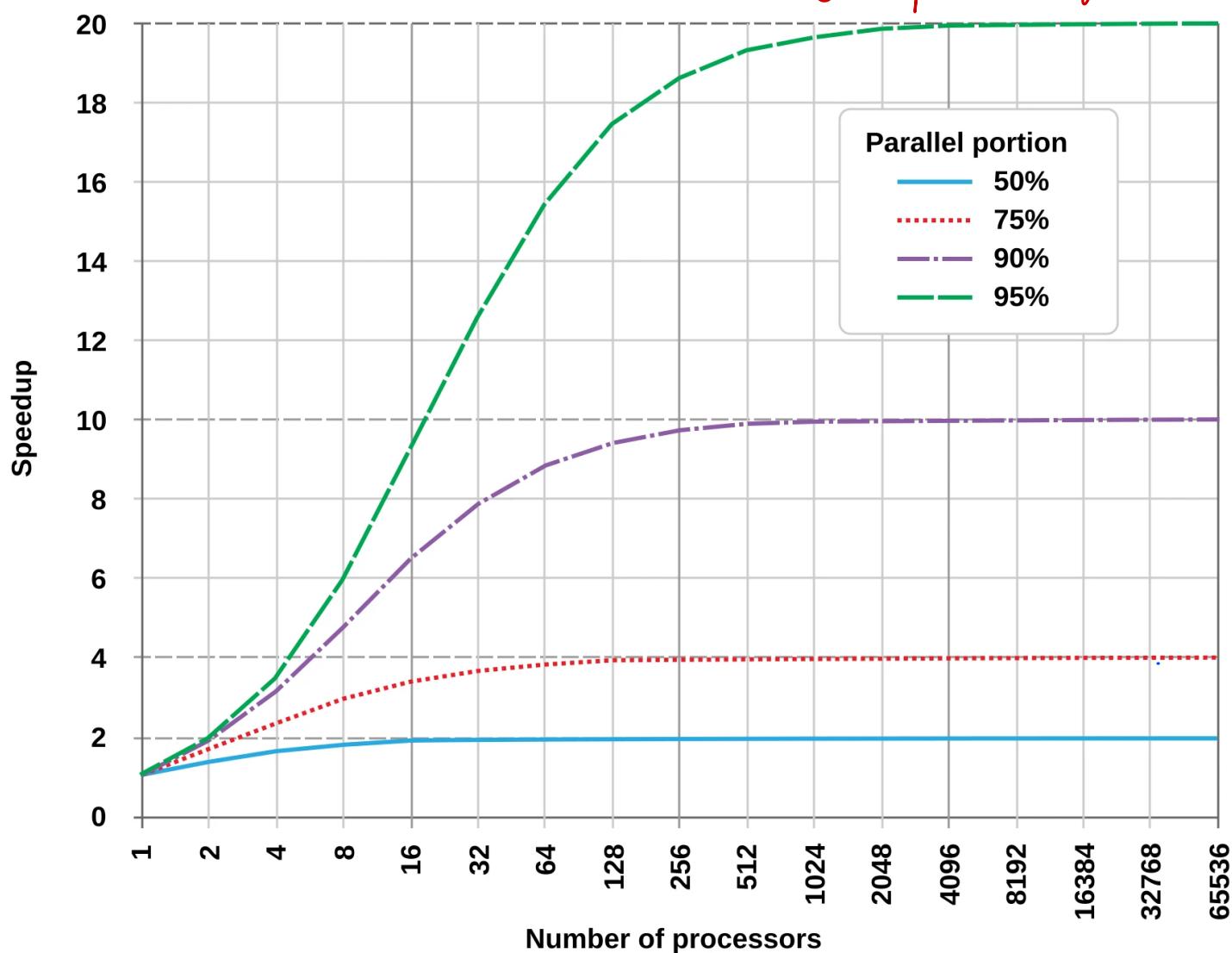
Sequential

y

Parallelizable

Amdahl's Law

(Law of diminishing returns)



→ Gustafson's law:

→ Increase the problem size proportionally so as to keep the overall time constant.

→ The scaling keeping the problem size constant is called strong scaling
→ " due to increasing problem size is called weak scaling

Amdahl's law

Scalability & Iso efficiency

→ Efficiency decreases as P increases, but increases with increasing N:

$$E \propto \frac{N}{P} \uparrow$$

- How effectively run the parallel algo. we can increasing no. of processors
- How the no. of computations must scale with P to keep E constant. This is called an iso efficiency function: expressed in terms of P
- No. of computations we need to add, to keep the efficiency constant.
- The lower the iso efficiency function, the better. Having a small iso efficiency function basically implies not many computations need to be added to keep efficiency constant.

Parallel Program Models

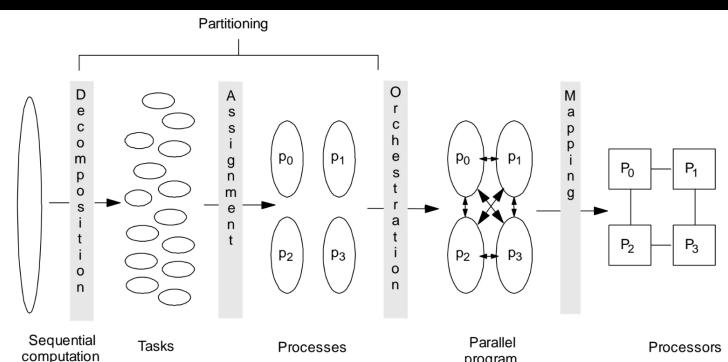
21/10/25

- Classification 1
 - Single program multiple data (SPMD)
 - Multiple " " (MPMD)
- Mostly used
- **Fortran**
- Classification 2
 - Shared memory model : Threads, open MP, CUDA
 - Message passing MPI (Message passing interface)
- Widely used
- Used in GPUs

Parallelism in Program

→ How to produce a parallel version of a sequential program?

- Step 1: Decomposition - Identify Tasks that can be performed parallelly.
- " 2: Assignment - Grouping the tasks into processes with best load balancing.
- " 3: Orchestration - Reducing synchronization & overhead
- " 4: Mapping - Map processes onto processors (if reqd.)

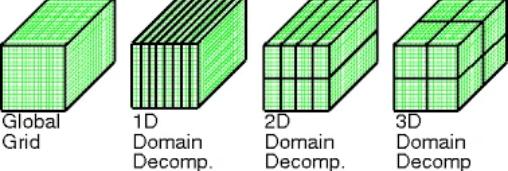


→ Decomposition & Assignment
 → Specifies how to group tasks of a process s.t. workload is balanced & overhead is reduced
 → Can be done using structured approach.
 → Both decomposition & assignment are usually combined together, and are independent of architecture

Parallelism
 is dictated by the data

Owner-computer rule

Data Parallelism & Domain decomposition



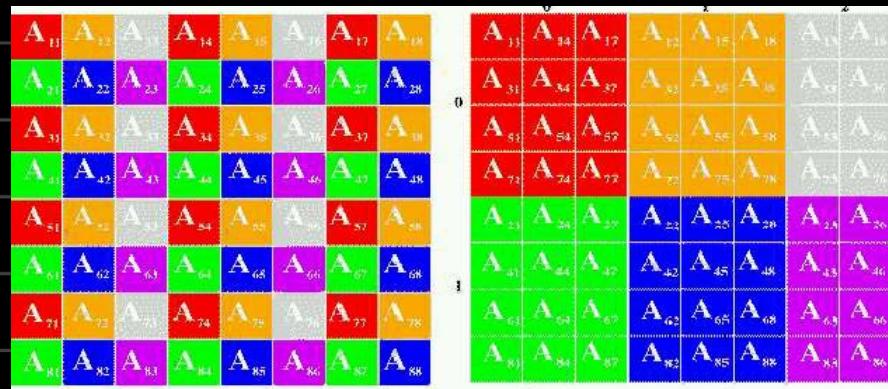
→ Domain decomposition: When data is divided across the processing entities, and each process owns & computes a portion of the data. A matrix grid is used to specify domain decomposition.

Data Distributions

- For dividing the data in a dimension among processes in a dimension, data distribution schemes are followed:
- Some common data distributions are:

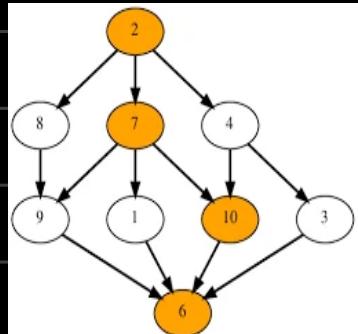
- Block: for regular computations
- Cyclic: when there is load imbalance across space.

→ These two distributions can be mixed-and-matched



Tern Parallelism

- Independent terms are identified, where the independent terms may or may not process different data.
- Objectives:
 - Balance the groups
 - Minimise inter-group dependencies
- Represented by task dependency graph
 - NP-hard problem
- Can be done using auto-parallelisation compiler, but such compilers do not do such a good job!



Orchestration

- Goals:
 - "Pack" the data structures
 - Structuring communication
- Synchronisation

- Solutions:
 - Minimise data locality
 - Minimise volume of data exchange
 - "frequency" " " "

→ Latency: How much time is reqd.
(L) for first byte of data
to be received by the receiver

\rightarrow Pausing can be used here \rightarrow Bandwidth: speed of data communication in bytes/second
 \rightarrow Total time = $l + \frac{X}{b}$, (b)

where X bytes needs to be sent.

\rightarrow Pausing optimizes the latency term.

\rightarrow Minimizing contention / hotspots

\rightarrow One particular variable is accessed by multiple processes, or one particular process is bombarded with data.

\rightarrow Do not use the same communication pattern in all processes

leads to sequential contention

E.g.

for ($i=0$ to n)

if ($p \neq me$)
communicate with p

	Processor 0	1	2	3
0	-	1	2	3
1	1	-	2	3
2	1	2	-	3
3	1	2	3	-

i We can use randomized message passing to solve this

processor 3 will become hot-spot @ time = 3 units

\rightarrow Overlapping computations with interactions

\rightarrow split communications into two phases:

\rightarrow Depends on communicated data (type 1)

\rightarrow Does not depend on communicated data (type 2)

\rightarrow Soln: initiate type 1 communication, during this, perform type 2 also.

\rightarrow Replicating data / computations: If doing some computation can save on communication cost, it is better to do so.

- Mapping
- Map logical processes to the actual processors, depends on network topology and communication pattern
 - Processor speeds can also be considered in case of heterogeneous systems.
 - All data & task parallel strategies follow static mapping.
 - Dynamic mapping (self-scheduling / work-stealing)
 - A process holds a set of tasks, it distributes these tasks among the processors.
 - Once a process completes its task, it should assign the workload to another process for another task.

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

Sample program:

```

procedure Solve (A) /*solve the equation system*/
  float **A;           /*A is an (n + 2)-by-(n + 2) array*/
begin
  int i, j, pid, done = 0;
  float temp;           /*to indicate which row to start from*/
  mybegin = 1 + (n/nprocs)*pid;    /*to indicate which row to end at*/
  myend = mybegin + (n/nprocs);
  while (!done) do /*outermost loop over sweeps*/
    diff = 0;          /*initialize difference to 0*/
    Barriers (barrier1, nprocs); /*tries to achieve synchronisation between threads, a
                                   process can move from the barrier to the next step only if all
                                   other threads have also called barrier; it will
                                   keep waiting until then.*/
    for i ← mybeg to myend do/*sweep for all points of grid*/
      for j ← 1 to n do
        temp = A[i,j];           /*save old value of element*/
        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                         A[i,j+1] + A[i+1,j]); /*compute average*/
        diff += abs(A[i,j] - temp); /*mutual exclusion is reqd. to update shared variable diff*/
      end for
    end for
    if (diff/(n*n) < TOL) then done = 1;
  end while
end procedure

```

Issues with the program:

- diff & the array A are shared variables, but they are not protected by mutex.
- Too much communication w.r.t. shared variable diff.

Solution :

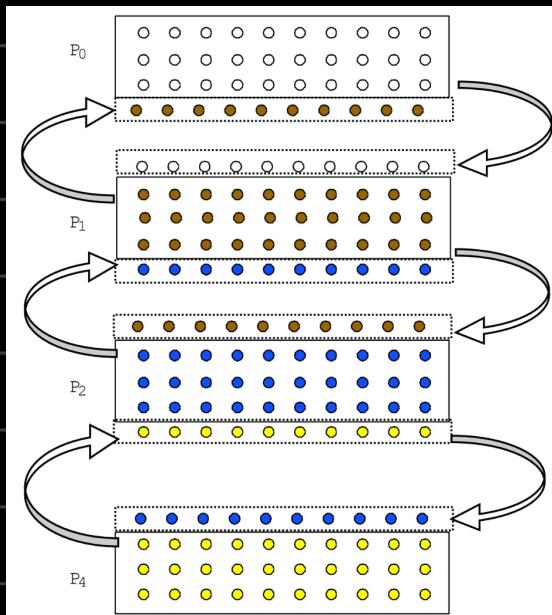
- Add proper mutex statements
- Rather than diff being updated for every iteration of the inner loop, we can just maintain a local diff variable and update the global diff for every iteration of the outer loop.

Final program:

```
procedure Solve (A) /*solve the equation system*/
    float **A; /*A is an (n + 2)-by-(n + 2) array*/
begin
    int i, j, pid, done = 0;
    float mydiff, temp;
    mybegin = 1 + (n/nprocs)*pid;
    myend = mybegin + (n/nprocs);
    while (!done) do /*outermost loop over sweeps*/
        mydiff = diff = 0; /*initialize local difference to 0*/
        Barriers (barrier1, nprocs);
        for i ← mybeg to myend do/*sweep for all points of grid*/
            for j ← 1 to n do
                temp = A[i,j]; /*save old value of element*/
                A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                  A[i,j+1] + A[i+1,j]); /*compute average*/
                mydiff += abs(A[i,j] - temp);
            end for
        end for
        lock (diff-lock);
        diff += mydiff;
        unlock (diff-lock)
        barrier (barrier1, nprocs);
        if (diff/(n*n) < TOL) then done = 1;
        Barrier (barrier1, nprocs);
    end while
end procedure
```

Message Passing Version

- Used when we cannot declare A as a global array.
- What now: Used to accommodate data from other process.
- RECEIVE does not transfer my data, SEND does.
Hence, the data transfer is sender-initiated
- Mutual exclusion is implicitly provided.



- Instead of Po sending individual messages to all processes, it can do a broadcast.
- Similarly a REDUCE function can be used to add the messages of individual processes.
- Deadlock possibility
 - Let us assume that a process can send only after receiving. In such a case, deadlock shall occur when all processes are sending.
 - We can use semiotic flavours of SEND and RECEIVE to solve this

	SAS	Msg-Passing	Send/Receive	
Explicit global data structure?	Yes	No		
Communication	Implicit	Explicit	Synchronous	Asynchronous
Synchronization	Explicit	Implicit		
Explicit replication of border rows?	No	Yes	Blocking asynch.	Nonblocking asynch.

```

graph TD
    SAS[SAS] --> EGDS[Explicit global data structure?]
    SAS --> IM[Implicit]
    EGDS --> Comm[Communication]
    EGDS --> Sync[Synchronization]
    IM --> Sync
    IM --> ERB[Explicit replication of border rows?]
    Comm --> Implicit[Implicit]
    Sync --> Explicit[Explicit]
    ERB --> No[No]
    MP[Msg-Passing] --> SR[Send/Receive]
    SR --> S[Sync]
    SR --> AS[Asynchronous]
    S --> BA[Blocking asynch.]
    S --> NA[Nonblocking asynch.]
  
```

Open MP

- A portable programming model and standard for shared memory using compiler directives. Easy to convert sequential code to parallel code.

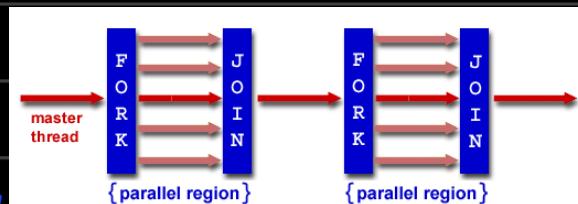
→ For n-join model

→ Begin as a single thread (master)

→ For: When parallel construct is encountered, threads are created.

→ Join: At the end of the parallel segments the threads are joined together.

→ Provides fine-level parallelism



→ Supports loop-level parallelism

→ "dynamic", where no. of threads vary from one parallel segment to another.

→ Follows Amdahl's law

- Work sharing constructs:
 - 3 types: loop, section, single
- for construct:
 - For distributing iterations among threads
 - schedule clause: Used to specify which iteration goes to which threads.
 - schedule (static, chunk-size)
 - Chunks of size chunk-size are distributed among the threads in a round-robin order.
 - schedule (dynamic, chunk-size)
 - Chunks of size chunk-size are distributed among the threads in a dynamic manner, depending on which thread is free.
 - schedule (runtime)
 - chunk size, thread, etc., are all determined by open MP at runtime.

```

include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
int i, chunk; float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i) {
#pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
} /* end of parallel section */
}

```

These are shared variables
 ↗
 This is private
 because it is constantly updating
 ↗
 by default, a thread will wait
 for all other threads. specifying
 no wait shall 'free up' the
 thread

Synchronization Directives

- `#pragma omp master` || This statement can only be executed by master
- `#pragma omp critical` || Depict this is critical section; only one thread can access critical section at a time.
- `#pragma omp atomic` || This statement must be executed atomically; the thread must not be context-switched out
- `#pragma omp flush (variable list)` || flush the updates made to those variables so that the updates are visible to all other threads immediately.
- `#pragma omp ordered` || The threads should execute in order of thread number.

Data Attribute (clause)

- Variables are shared by default. Data scopes are explicitly specified by data scope attribute clauses.

threadprivate

- Global variable list is made private to a thread, i.e. each thread gets its own copy of the variables.
- Persist between different parallel regions.

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
main () {
    /* Explicitly turn off dynamic threads */
    omp_set_dynamic(0);
    /* First parallel region */
    #pragma omp parallel private(i,beta)
    for (i=0; i < 10; i++) alpha[i] = beta[i] = i;
    /* Second parallel region */
    #pragma omp parallel
    printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);}
```

private (list)

→ Variables in the list are private to each thread

firstprivate (list)

→ Variables in the list are initialized with the value of the old object

- last private (list)
 - The value of the private object corresponding to the last iteration is assigned to the object.
- locus'
 - Simple : cannot be located more than once
 - Negotiable : The thread which has the locus can place multiple locus
 - Used with third - party Open MP libraries .
 - A subfunction may attempt to lock a variable again . This ensures it won't be blurred .

Message Passing Interface (MPI)

- Enlist communication & synchronisation , giving more control to the user .
- Point - to - point communications - send & recv :

MPI-SEND (buf, count, datatype, dest, tag, comm)

Message

Destination

Msg. identifier

Communication Context (MPI communicator) , used to indicate the group where the msg . must be sent . MPI_COMM_WORLD is the "wild card"

The same node may have multiple ranks in multiple groups .

MPI-RECV (buf, count, datatype, source, tag, comm, status)

Message

source rank

sender & receiver
tag should
match

receiver can query the MPI interface know the status of the message .

comm . context must also be same as the sender's comm . context

MPI - Comm - rank : Can be used by a process to get its rank .

- Receiver source & tag field may also be wild carded using MPI_ANY_SOURCE and MPI_ANY_TAG.
- Can be used when order of message receiving does n't matter.

What happens to status if we use wild cards??

- Utility functions :
 - MPI_Init : Initialise MPI environment
 - MPI_Finalize : (close " ")
 - MPI_Comm_size (comm_size) : Total no. of processes in the group pointed to by the communicator.
 - MPI_Wtime ()
- MPI_SEND & MPI_RECV are blocking; they will keep waiting for that message. We can use buffering to save time (so that the processor can continue to work instead of waiting)

↑ Hint for immediate because this function returns immediately.

MPI_ISEND (buf, count, datatype, dest, tag, comm, request)

MPI_IRecv (buf, count, datatype, dest, tag, comm, request)

MPI_WAIT (request, status)

a sort of 'token' for checking completion of communication

blocks until communication is complete

MPI - TEST (request, flag, status) // checks if communication has completed

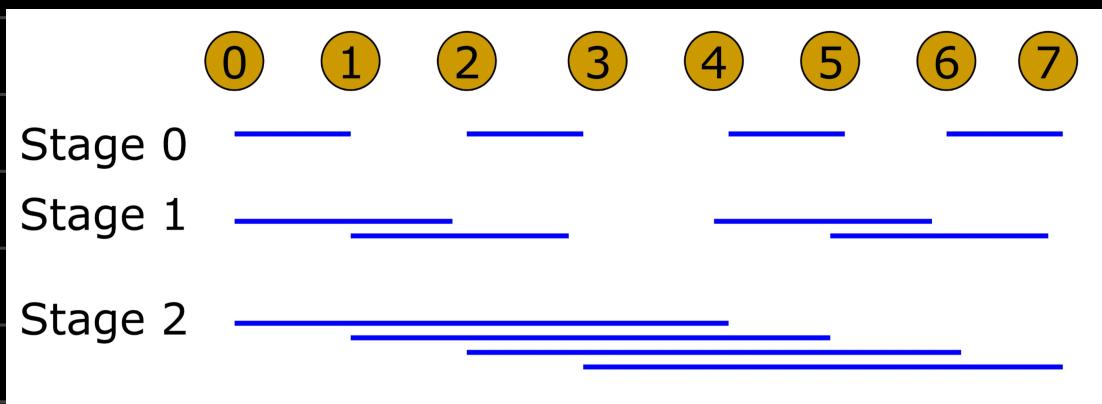
MPI - REQUEST - FREE (request)

The prev. blocking has been split into posting & completion

MPI - SEND \Leftrightarrow MPI - ISEND + (immediately) MPI - WAIT

Butterfly algorithm

- In the n^{th} round, process i synchronizes with $i \oplus 2^k$ pairwise.
- Will require $2 \log P$ pairwise synchronizations



Broadcasting

- If all processes have to broadcast to everyone else, we can use the naive algo:

for all processes 0-n

send to process

recv from process

- This can result in blocking. It is much better to use a method similar to butterfly:

```

for all procs. i in order{
    dest = (my_proc+i)modP
    src = (myproc-i+P)modP
        send to dest and recv from src
}

```

CUDA Programming

→ Hierarchical Parallelism

- Parallel computations are managed as grids, one grid executes after another
- Grid consists of blocks, Blocks are assigned to SM. ↪ *streaming multiprocessor*
- Blocks consist of elements, elements are computed by threads
- Up to 1024 threads per thread block

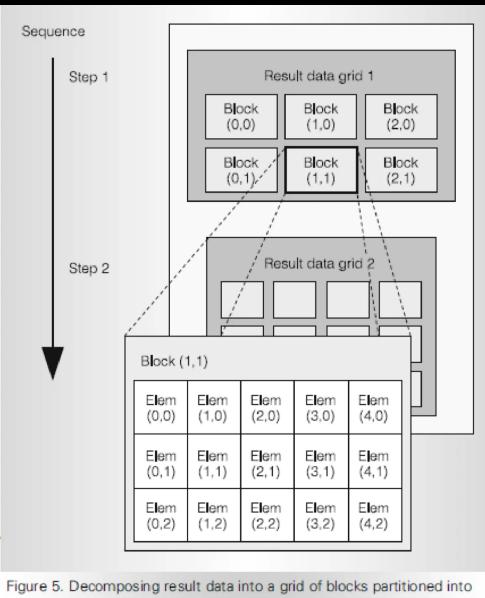
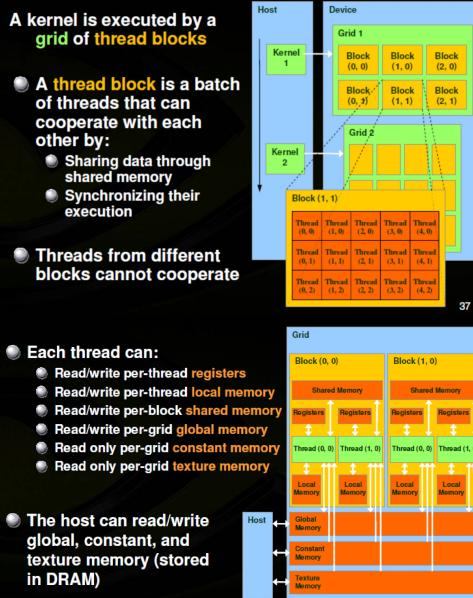


Figure 5. Decomposing result data into a grid of blocks partitioned into elements to be computed in parallel.



CUDA Memory Spaces

- Global and Shared Memory introduced before**
 - Most important, commonly used
- Local, Constant, and Texture for convenience/performance**
 - Local: automatic array variables allocated there by compiler
 - Constant: useful for uniformly-accessed read-only data
 - Cached (see programming guide)
 - Texture: useful for spatially coherent random-access read-only data
 - Cached (see programming guide)
 - Provides address clamping and wrapping

Memory	Location	Cached	Access	Scope ("Who?")
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

- Parallel portions of an application are executed on the device as kernels**
 - One **kernel** is executed at a time
 - Many threads execute each **kernel**

- Differences between CUDA and CPU threads**
 - CUDA threads are extremely lightweight
 - Very little creation overhead
 - Instant switching
 - CUDA uses 1000s of threads to achieve efficiency
 - Multi-core CPUs can use only a few

→ CUDA Programming Language

- Programming language for threaded parallelism using minimal extension of C.

→ Built-in variables

→ `threadIdx. {x,y,z}` : thread ID within a block

→ `blockIdx. {x,y,z}` : block ID within a grid

→ `blockDim. {x,y,z}` : No. of threads in a block

→ grid dim. $\{x, y, z\}$: No. of blocks within a grid

→ general CUDA tips

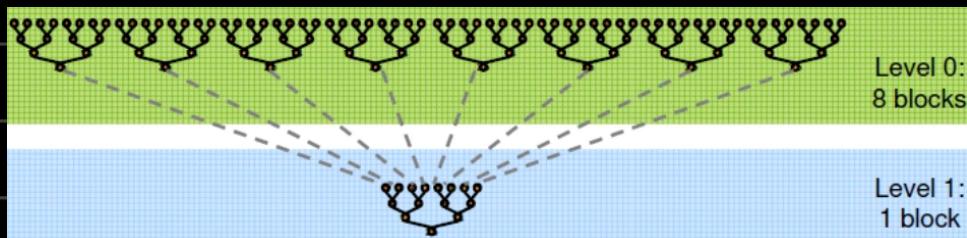
- copy data from CPU to GPU → By default execution on host does not wait kernel to finish
- Compute on GPU
- copy data back from GPU to CPU → Minimize data transfer bet. CPU & GPU
- Maximize no. of threads on GPU.

→ CUDA Elements:

- CudaMalloc : for allocating memory on GPU
 - CudaMemcpy : for copying data from CPU to GPU
 - CudaFree : to free allocated memory ↪ similar to is
 - synchronize : to synchronize all threads in a block
- ↳ short notes (Completed 9/11)

→ CUDA synchronization

→ Engineered to build global synchronization logic, hence we use tree structure



→ Soln: Decompose the problem into multiple GPU kernel invocations

PRAM (Parallel Random Access Memory)

- Helps to write generic parallel algorithm without any architecture constraints.
- Allows programmers to treat processing power as unlimited
- Ignores complexity of inter-process communication

→ Benefits:

- can be suitable as a 'benchmark' for a parallel program. Can be used to establish tight V.B / L.B for practical implementations
- Suitable for modern day architectures, i.e. GPUs ↗ because GPUs have very high no. of threads

→ PRAM architecture model

- consists of control unit, global memory, and an unbounded set of processors, each with its own private memory.
- Executed in SIMD model.

→ PRAM models may deal with read/write conflicts in different ways:

→ EREW: Exclusive read exclusive write (most restrictive)

→ CREW: Concurrent read exclusive write

→ Similar to SE X locus

→ CRCW: Concurrent read concurrent write (most flexible)

→ COMMON: All threads can write simultaneously if they are writing the same value.

→ ARBITRARY: values written may be different, but one of the values will be chosen as final.

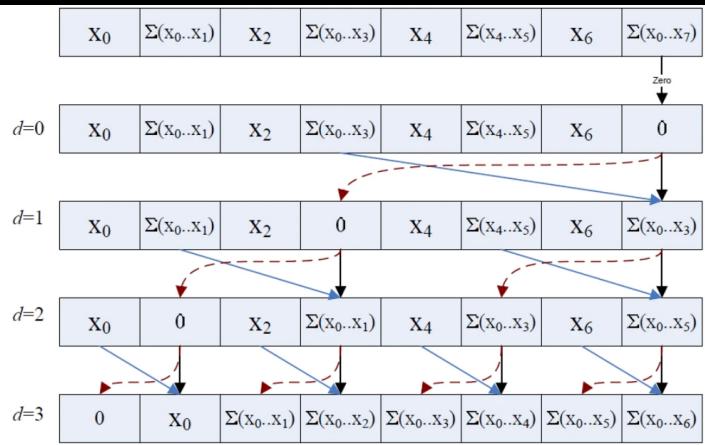
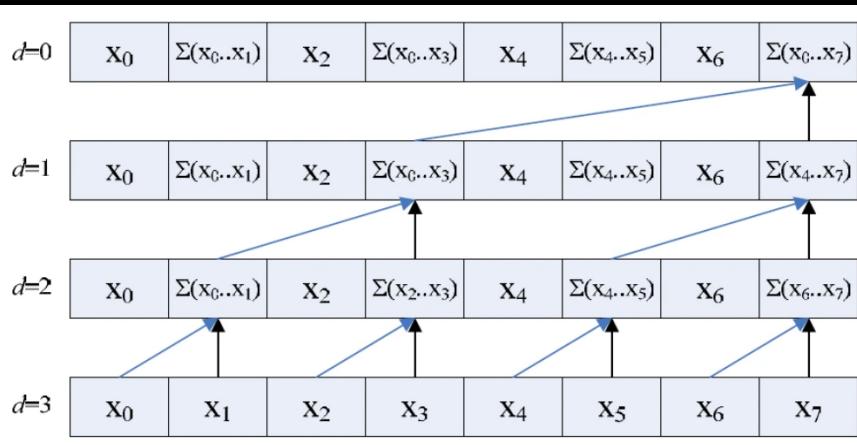
→ PRIORITY: processor with lowest index will be prioritized

* Any PRAM model can be executed with any other model (using open modifications, of course)

→ Steps in PRAM:

- Phase 1: Sufficient no. of processes are activated
- ' 2: Activated processes compute in parallel

→ E.g. Prefix sum calculations: converge upto $\frac{n}{2}$ processors, taking $O(\log n)$ time



Up sweep

→ E.g. merging two sorted list

not operationally efficient

Down sweep

→ Most PRAM algorithms are faster by performing more operations than an optimal (sequential) algorithm.

→ For this problem, a sequential program requires $(n-1)$ comparisons to merge two sorted $\frac{n}{2}$ lists.

→ If we implement a CREW PRAM model for this problem, the T.C. rises to $O(n \log n)$, which makes it not cost-optimal.

* PRAM just solves the problem faster by using more processors, but using more operations

Example: Enumeration sort

- Computes the final position of each element by comparing it with the other elements and counting the number of elements having smaller value
- A special CRCW PRAM can perform the sort in $O(1)$ time
- Spawn n^2 processors corresponding to n^2 comparisons
- Special CRCW PRAM - If multiple processors simultaneously write values to a single memory location, the sum of the values is assigned to that location

Example: Enumeration sort

- So, each processor compares $a[i]$ and $a[j]$. If $a[i] > a[j]$, writes $\text{position}[i] = 1$, else writes $\text{position}[i]=0$
- So the summation of all positions will give the final position - constant time algorithm
- But not cost-optimal - takes $O(n^2)$ comparisons, but a sequential algorithm does $O(n \log n)$ comparisons

Parallel Algorithms (Sorting)

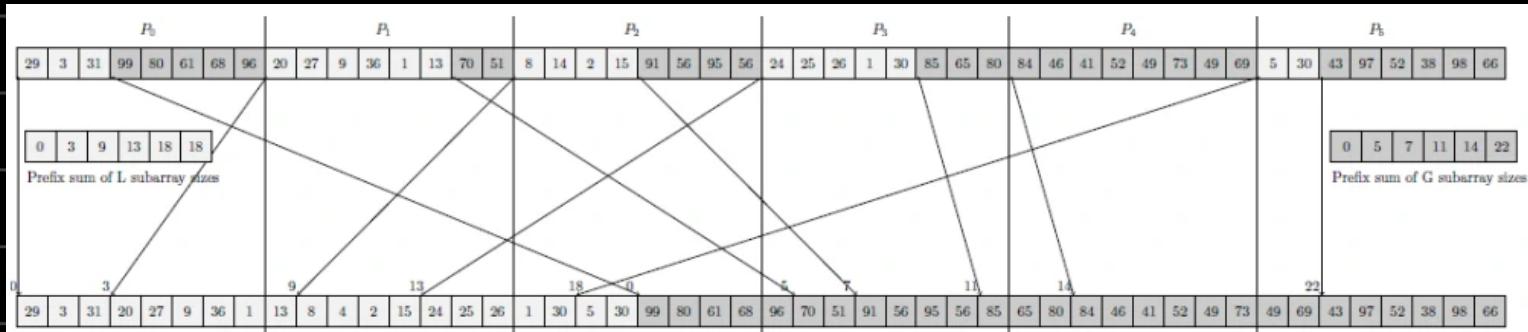
→ Parallel Quicksort

- Start with a single process and divide array into sub-arrays.
- Both processes will perform their iteration first (using pivot). The sub-arrays to the left & right of pivot element are then sorted in the next iterations. We distribute these sub-arrays to two processes.
- This, however, leads to inefficient usage of processors.

→ Another algorithm : Involves all processor in all iterations

→ Parallel quick sort:

- Processors are divided into two groups. First group will process L_i 's and second group will process G_i 's.
- * The sizes of processor groups should be in the ratio of L_i 's and G_i 's to ensure a balanced load.
- Shared memory implementation
 - All L_i 's are formed in the first part of the array, and all G_i 's are formed in the second part.
 - Each processor needs to know the locations in the shared memory where it has to write its L_i & G_i .
 - Prefix sums the size of subarrays can be used
 - Can be done in $O(\log P)$



→ Message passing version

→ A processor should know which elements in its L.E.G. to send to which processor. We can use distributed prefixsum.

→ A processor calculates the destination processor using:

→ Total no. of elements in L subarrays

→ Prefix sum of sizes

→ size of the processor group that will be responsible for L subarray

→ This process is repeated with subgroups, until no. of subgroups = no. of processors.

→ Complexity analysis:

→ In each iteration : $O(\log P)$ iterations

→ Broadcast: $O(\log P)$

→ Allreduce: $O(\log P)$

→ Prefixsum & all-to-all : $O(\log P + \frac{N}{P})$

→ local quick sort: $O\left(\frac{N}{P} \log\left(\frac{N}{P}\right)\right)$

→ Total = $O\left(\frac{N}{P} \log\left(\frac{N}{P}\right)\right) + O(\log P \cdot (\log P + \log P + \log P + \frac{N}{P}))$

$$= \underbrace{O\left(\frac{N}{P} \log\left(\frac{N}{P}\right)\right)}_{\text{Computational cost}} + \underbrace{O(3 \log^2 P + \frac{N}{P} \log P)}_{\text{Communication cost}}$$

* Ideally, we want computation cost = communication cost

