Introduction to Scalable Systems

Data Structures & Algorithms

→ Saving storage for sparse matrices

→ Each non-zero element is added as a tuple in a list.
→ This tuple contains the row, coln., & value.
→ Storing row & coln. overheads is better than storing all 0s.

→ Compressed Sparse Row (CSR) representation (Stores as three arrays):

→ V[nnz]: non-zero values in row-major order
→ C[nnz]: coln. offset within row group for a non-zero value
→ R[M+1]: stores cumulative count of non-zero elements till $(i-1)^{th}$ row

(nnz: no. of non-zero elements; M: dimension of the matrix)

→ Relational, cache friendly

Hashing Analysis

→ Using chain hashing: Expected no. of keys in unsuccessful search

$$= \frac{1}{n} \sum_{j=1}^{n} \left[1 + E\left(\sum_{j=i+1}^{n} X_{ij}\right)\right]$$

$$= 1 + \frac{n-1}{2b}$$

$$= O(1+\alpha), \text{ where } \alpha = n/b$$

(n: no. of elements stored; b: no. of blocks)

B-Trees

→ Depth of the tree = $\left\lceil \log_{\lceil \frac{m}{2} \rceil} (\text{size}) \right\rceil + 1$

Graphs

→ Diameter = Distance of the longest shortest path!
= max $(d(u,v))$ $\forall u, v \in V, u \neq v$

→ Clique : A subgraph (which is a connected graph
→ Maximal clique : clique with most no. of nodes possible.

→ Graph traversals
→ BFS
→ Start at source vertex (use a queue
→ $O(|V|)$ for adjacency matrix (better for dense graphs due to cache locality
→ $O(d)$ for " list (better for sparse graph
(edge degree)

→ Total time
→ $O(w|V|)$ for adjacency list
→ $O(w+f)$ " " matrix
(w = no. of vertices in connected component; f = no. of edges in connected component.)

→ DFS (Use stack instead of queue)
→ Requires $O(h)$ space (for recursion stack)
→ Same TC as BFS

Algorithms

→ Dijkstra's algorithm

Method of storage	TC
Array of weights	$O(\|V\|^2 + \|E\|)$
Using min. heap to store edge weights	$O((\|V\|+\|E\|)\log\|V\|)$
Using Fibonacci heap	$O(\|E\|+\|V\|\log\|V\|)$

Worse than array implementation when $|E| >> |V|$, i.e for an almost connected graph

Parallel Programming

→ Flynn's classification: In terms of parallelism & data stream

→ SISD: Single instruction single data (parallel programming)

→ SIMD: " " multiple " (vector processors & processor arrays)

→ MIMD: Multiple " " "

→ Used in supercomputers etc.

→ Classification based on memory

→ Shared memory: UMA & NUMA

* Address space is shared, not a single memory

→ NUMA (Non-Uniform memory architecture)

→ Message passing

→ UMA (Uniform memory architecture)

→ Shared memory is distributed among processor nodes

Shared Memory & Cache

→ If a mem. location has been updated by a processor, how will other processors know? They may use the old value (in their cache)!

→ Solutions:

→ Write update: Propagate cache line to all processors on write write.

→ Writes shall be costlier

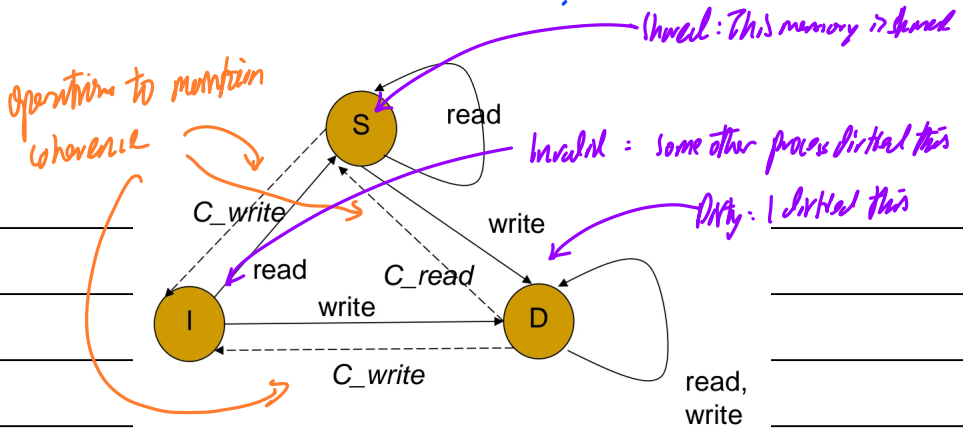
→ Cache line is sent to all processors, even those who do not need it, thereby wasting resources.

followed in all systems

→ Write invalidate: Inform all other processors that they have "stale" data, and to read from memory next time.

→ Reads are costlier, dirty bit is used.

→ Three states are possible:



Implementations

→ Snoopy: Used with bus-based architecture

→ Directory based:

→ A small part of shared memory is used to cache states, so that any changes can be relayed to relevant processors only.

→ Contains cache states and the list of relevant processors for a cache line

→ Increases memory access time.

→ Implementation:

→ Use presence bits for owner processors:

→ Full-bit vector scheme: $O(M \times P)$ storage for P processors & M cache lines. (Not required)

→ Sparse (tagged) directory: Limited cache lines & presence bits

→ False sharing

→ A cache block may be shared by two processors, but not the actual memory location.

→ Solution: restructure code s.t. it is non-intrusive

→ Use padding to ensure a cache line only contains its own data.

Inter connects

→ PCI or PCI-e is used to connect a processor to a network line.

→ Communication network: Consists of switching elements connected with each other

→ Topologies:

→ Bus / ring

→ Cross bar switch: 2-D grid of switching elements

→ P^2 switching elements

→ Multi-stage network (omega)
  → consists of ($\log_2 P$) switches, each with $P/2$ switching elements
  → Can be blocking

→ Mesh, torus, hypercube, and fat-tree

→ Evaluation metrics for interconnection topologies:
  → Diameter: max dist. b/t any two nodes
  → Connectivity: multiplicity of paths b/t 2 nodes (min no. of links to be removed to disconnect the topology)
  → Bisection: min. no. of links to be removed to split the network to two equal halves.
  → Channel width: no. of bits that can be simultaneously communicated over a link.
  → " rate: perfor mance of a single physical wire
  → " bandwidth: channel width × channel rate
  → Bisection " : Max. volume of communication b/t two halves of the network
  → bisection width × channel bandwidth

Type of network	Diameter
Full-connected	1
Star	2
Ring	$P/2$
Hypercube	$\log_2 P$

Type of network	Connectivity
Linear array	1
Ring	2
2-D mesh	2
2-D mesh (w/t wraparound)	4
D-dimension hypercube	D

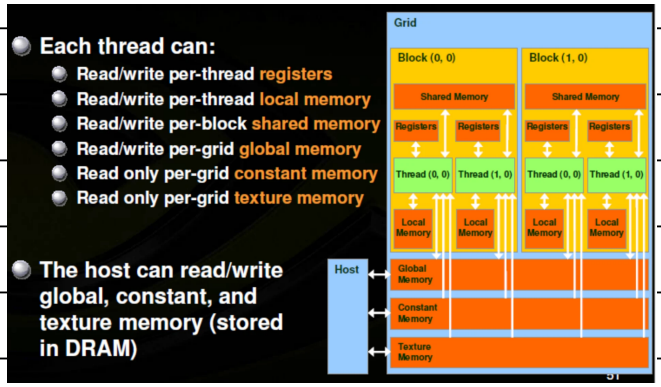
(2-D mesh (w/t wraparound) = torus)

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

Memory Hierarchy

→ Global/device memory
  → Can be accessed by all threads in the streaming multiprocessor (SM)
  → Can be accessed by CPU (host)
  → 200-400 clock cycles per access

→ Shared memory
  → In each SM, and shared by all threads in a SM.
  → Can be configured as x KB shared mem., and the rest as L1 cache
  → 20-40 clock cycles for access



→ Differences w/ CPU threads:
  → Fast context switching: A thread's context is stored in shared memory & registers until it completes.
  (zero overhead context switching)

  → Explicit cache management: User's program will have to bring the frequently used data manually.

Parallelization Principles

→ Overheads
  → Communication delay → Idling → Synchronization

  To prevent this, ensure program is load balanced

→ Evaluation metrics
  → Speedup, $S(p,n) = \frac{T(1,n)}{T(p,n)}$ // Time for sequential execution / // Time for parallel execution with p processors.
  → Ideally, $S(p,n) = p$ // equal distribution of loop

→ Usually, $S(p,n) < p$ // due to overheads
→ Rarely, $S(p,n) > p$ // when we get addl. benefits due to parallelisation, i.e. cache

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

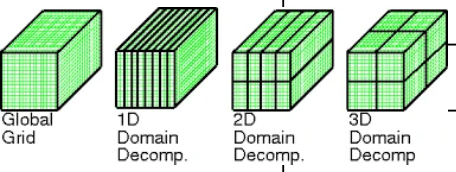
→ Speedup limits: Amdahl's law // always diminishing returns
→ Performance is always bottlenecked by sequential section
→ Places a limit on speedup:

$$S(p,n) \leq \frac{1}{f_s + \frac{f_p}{p}}$$

// f_s = time for sequential section
f_p = time for parallel section

→ Gustafson's law: Increase the problem proportionally to keep the overall time constant.

→ Parallelising a program
→ Step 1: Decomposition & Step 2: Assignment
→ Identify segments of code that can be parallelised.
→ How to group tasks? // use structured approach



→ Domain decomposition
→ Divide the data among nodes, each process owns & computes a portion of the data
→ This is specified by a process grid.
→ Data distribution: Schemes used to divide data
→ Blocks: for regular (uniform) compute
→ Block-cyclic: for imbalanced compute. Divides the data into blocks and distributes them among the processors in a cyclic fashion.

→ Step 3: Orchestration: Reducing synchronization overhead
→ Structures & synchronise communication
can be done by pairing data structures

→ Goals :

Eg not communicating intermediate results

→ Maximise data locality
→ Minimising volume of data exchange
→ Minimise frequency of data exchange
→ Reduce communication by replicating data

→ Minimise contention / hotspots
→ Do not use same communication pattern as the other processes.

→ Overlap interactions with computations
→ Sacrifice compute to save on communication.

→ Step 4 : Mapping
→ Which process runs on which processor ?
→ Dynamic mapping Ⅱself-scheduling work-stealing
→ A process holds a set of tasks, which it distributes to nodes as and when they become free.

Open MP

→ Shared memory using compiler directives
→ Uses fork-join model ; provides fine-grained parallelism
→ Supports nested parallelism

→ Constructs
→ Parallel : #pragma omp parallel <clauses>
→ private (list) : Declares the vars. in list to be private to each thread
→ shared (list) : Declares the vars in list as shared by all threads
→ firstprivate (list) : private (list) + initialisation
→ default (shared / none) : Sets default data scoping rule of variables within the parallel region.

→ if (exp): Executes the section in parallel if exp evaluates to true. Otherwise, the segment is executed in sequential.

→ num_threads (int): Defines a number of threads to use for the parallel region.

→ reduction (op: list): Reduces the values from all threads into a single variable.

→ For: # pragma omp for <clauses>

→ schedule clause:

→ schedule (static, chunk-size): Original data is split in chunks and divided among the threads in

→ schedule (dynamic, chunk-size): Original data are distributed among the threads in a dynamic manner.

→ schedule (runtime): chunk size, no. of threads, etc. are decided at runtime.

→ Synchronisation directives:

→ # pragma omp master: this section can be executed only by master

→ # " " critical: defines the critical section

→ # " " atomic: this section must be executed w/o context switching

→ # " " flush (var-list): flush all variables in var-list s.t. the updates made to these variables are visible to all threads

→ # " " ordered: Threads should execute in order of thread no.

Message Passing Interface - MPI

→ Point-to-point communication:

* Rank is essentially the unique id of a process w.r.t. the particular communication context

Msg. identifier

MPI_SEND (buf, count, datatype, dest, tag, comm)

Message

Destination rank

communication context: used to indicate the group where the message must be sent

↑ node can have different ranks in different groups

MPI_RECV (buf, count, datatype, source, tag, comm, status)

Message

Source rank

sender & receiver tag should match

communication context

receiver can query MPI integers to check the status

* MPI_COMM_WORLD is the wild card communication context.

MPI_Comm_rank is used by a process to obtain its own rank.

Receiver source & tag field can be wildcarded by using MPI_ANY_SOURCE & MPI_ANY_TAG respectively.

→ Some utility functions:

→ MPI_Init: Initialize MPI environment

→ MPI_Finalize: Close

→ MPI_Comm_Size (comm, &size): Used to get total no. of processes in the group pointed to by the communicator

→ MPI_Wtime: Returns current wall clock time.

→ Non-blocking communication

I stands for immediate, because these function return immediately

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

MPI_IRECV (buf, count, datatype, source, tag, comm, request)

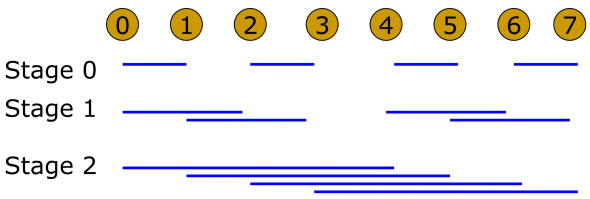
MPI_WAIT (request, status) → blocks until communication is complete

A sort of form for checking request status

MPI_TEST (request, flag, status) → checks if communication has completed

MPI_REQUEST_FREE (request) → used to de-allocate (force stop) a process.

→ Collective communication using barrier will result in blocking, because a thread is waiting for all other threads. To solve this, we can use the butterfly algorithm!
  → In the n^{th} round, each process i synchronises with process $i \oplus 2^k$ pairwise.
  → Requires $2 \log P$ piecewise synchronisation



CUDA Programming

→ A kernel is executed by a grid of thread blocks. A thread block is a batch of threads that can co-operate with each other by:
  → Sharing data using shared memory, or
  → Synchronising their execution
→ Threads from different blocks cannot co-operate

→ CUDA Memory spaces:

Memory	Cached?	Access Modes	Scope
Local	X	RW	One thread
Shared	N/A	RW	All threads in a block
Global	X	RW	All threads & host
Constant	✓	R	All threads & host
Texture	✓	R	All threads & host

→ Differences b/t CPU & CUDA threads
  → CUDA threads are lightweight
    → zero-overhead context switching
    → very minimal creation overhead
  → CUDA uses 1000s of threads

→ CUDA is a minimal extension of C
→ CUDA built-in variables:
  → threadIdx.{x,y,z} : thread ID within a block
  → blockIdx.{x,y,z} : block " " grid
  → blockDim.{x,y,z} : no. of threads in a block
  → gridDim.{x,y,z} : " " blocks " grid

→ CUDA function
  → cudaMalloc & cudaFree : Used to allocate & free memory on GPU
  → cudaMemCopy : to copy data from CPU to GPU
  → syncthreads() : to sync threads in a block (barrier)

PRAM

→ Ideal model
  → Unlimited processors → Shared global memory
  → Uniform access time → SIMD execution
→ Handling conflicts
  → EREW : exclusive read, exclusive write
  → CREW : common " " (standard)
  → CRCW : " common "
    → common : All processors are writing the same value
    → Arbitrary : One random process is allowed
    → Priority : Processor with least PID is allowed
    → Reduction : Values are reduced (by sum/avg)

Parallel Quick Sort

→ One processor identifies the pivot & relays it to all processors.
→ Each processor does local partitioning
→ Processors are grouped based on the partitions to ensure load balancing
→ All elements lower than the pivot will be shifted to group 1, and all other elements go to group 2. We use prefix sums to find the correct location of the data.
→ Eventually, P_0 will get the smallest numbers, P_1 will get next set of smallest numbers, etc. Each processor does a regular quick sort and then the data is merged.

→ Complexity: $\underbrace{O\left(\frac{N}{p}\log\left(\frac{N}{p}\right)\right)}_{\text{communication}} + \underbrace{O\left(\log^2 p + \frac{N}{p}\log p\right)}_{\text{compute}}$