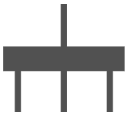


Parallel Computing

Author: Jose 胡冠洲 @ ShanghaiTech

Introduction

What & Why

Parallel Computing: Use multiple computation power together to solve a problem.

- :) With k processors, *ideally* can solve k times faster
- :) Memory size $\uparrow \Rightarrow$ solve larger problem, with more accuracy
- :) Fault tolerance $\uparrow$
- :(More prone to faults

Parallel Hardware consists of multiple independent processors communicating over an interconnect network. They:

- Have many different designs:
    1. Different processor types (CPU, GPU, FPGA, ASIC, Heterogeneous)
    2. Different interconnect networks
    3. *Shared memory* v.s. *Message passing*

- Exist at many different layers: instruction $\rightarrow$ core $\rightarrow$ chip $\rightarrow$ node $\rightarrow$ rack $\rightarrow$ ...

Parallel Software breaks a large problem into subproblems (**Tasks**) that can be solved *somewhat* independently. They:

- Must match the underlying Hardware
- Must respect dependencies between Tasks

Challenges of Parallelization

Parallel computing meets the following challenges:

1. *Communication*: Compute faster than can communicate & Gets worse when # of processors $\uparrow$
2. *Synchronization*: Interference & Dependencies between Tasks
3. *Scheduling*: Finding good allocation of Tasks $\rightarrow$ Processors
4. *Structured* (Specific) v.s. *Unstructured* (General but less efficient)
5. Some problems are NOT / don't seem to be parallelizable *inherently*
6. *Human factor*: Harder for human to design, track, and debug concurrent events

Current Status

Parallel Computing has become essential to taking advantage of *Moore's Law*.

Based on four kinds of processors:

- *Multicores* (多核, # $\leq$ 40, General purpose), e.g. Intel Xeon CPU
- *Manycores* (众核, Large number of simple cores), e.g. Nvidia GPU
- *FPGA* (Field programmable Gate Arrays): Configurable Hardware
- *ASIC* (Application Specific Integrated Circuits): Specialized & Expensive

When **Supercomputing** (High-Performance Computing, *HPC*) becomes popular, *Energy efficiency* is increasingly important.

Parallel Architecture

Instruction Level Parallelism

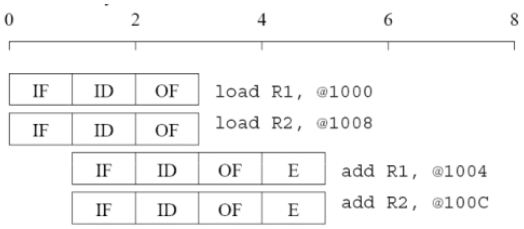
The lowest level of parallelism is **Instruction Level Parallelism** (*ILP*), which exists inside a single computing core.

- **Pipelining**:

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

  - Execute different pieces of an Instruction in different Pipeline Stages
  - Speedup $\approx$ # of stages
    - Need time to fill and drain
    - Requires flushing on branch mis-prediction
  - 3 Hazards (*Structural* / *Data* / *Control*)

- **Superscalar**:



  - Have the ability to issue multiple instructions at one clock signal (because of redundant functional units)
  - Data dependencies detected and organized on *Run-time*
- **Very Long Instruction Word** (*VLIW*):
  - Similar to Superscalar
  - Finds and packs parallelizable instructions on *Compile-time*
    - :) Can do more sophisticated search
    - :(Compiler doesn't have runtime states like branch history

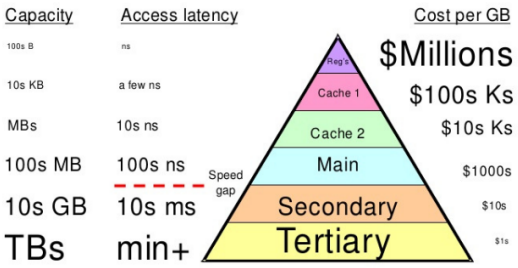
Memory Performance

Two separate measurements for performance:

- **Latency** (延迟) = Amount of time needed for a single data access request.
  - ⋆ when frequently transferring small requests
- **Bandwidth** (带宽, *Throughput*, 吞吐量) = Amount of data transferred per unit time.
  - ⋆ when transferring a large chunk at a time
  - Depends on Latency, but is a different measurement

CPU-Memory performance gap is huge nowadays.

- Therefore we need **Memory Hierarchy**:

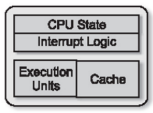
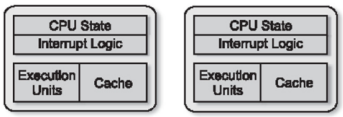
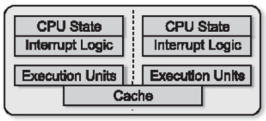
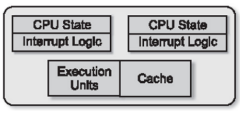


- **Caching** relies on *Temporal* and *Spatial Locality*
  - *Hit Rate* = # of access hits in Cache / Total # of accesses
  - Dramatically improves Latency, e.g.
    - Memory (100ns), but 90% hit Cache (5ns): $5 \times 0.9 + 100 \times 0.1 = 14.5$ ns

Thread Level Parallelism

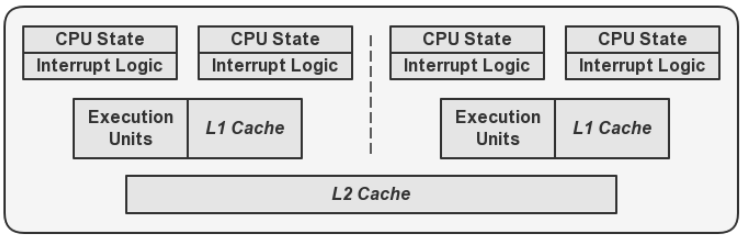
Computing cores & Caches can be combined in various ways to form processors. By integrating multiple execution units on a single chip, we are able to solve a task on multiple threads simultaneously, thus **Thread Level Parallelism** (*TLP*).

Multicore Architecture

Single-core	Multiprocessor	Multicore processor	Hyperthreading (超线程)
			
Simple	Separate processors	Cores share *L2 / L3* Cache	Threads share an execution unit, because their instruction streams can interleave with each other

> Hyperthreading is officially called **Simultaneous Multithreading** (*SMT*), but Intel implements it as two threads per core and called it *Hyperthreading*, thus this name is now more popular.

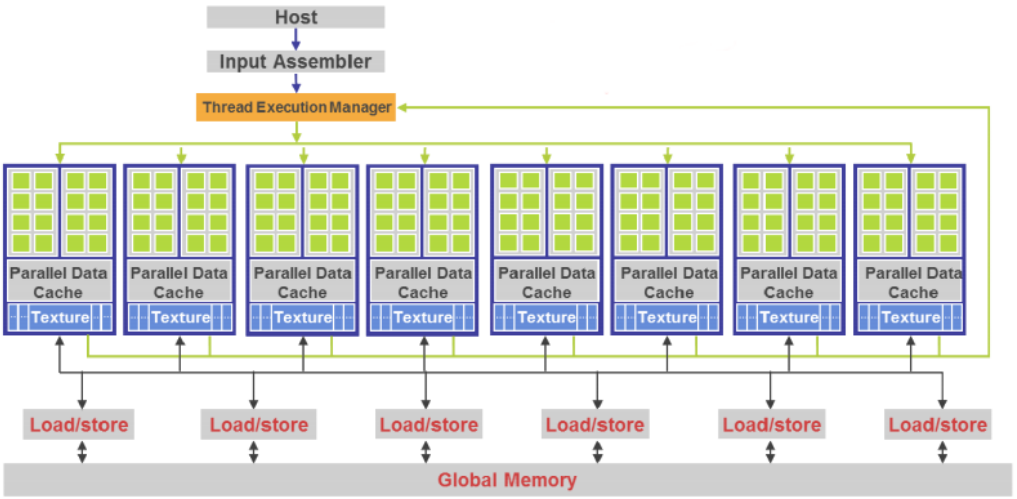
Nowadays we combine them:



Manycore Architecture

General Purpose Graphics Processing Unit (*GPGPU*) uses massive active threads to perform massive parallelism.

Example:



Flynn's Taxonomy

1. **SISD** (Single Instruction, Single Data): Naive
2. **SIMD** (Single Instruction, Multiple Data): Popular

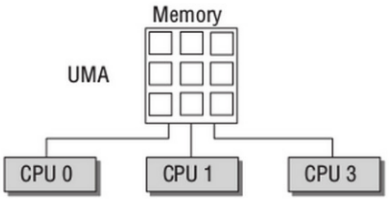
    - Digs **Data Level Parallelism** (*DLP*)
    - Effective & cheap to implement
    - Doesn't work when heavily branching / not balanced
3. **MISD** (Multiple Instruction, Single Data)
4. **MIMD** (Multiple Instruction, Multiple Data): General
    - Different cores in a chip form MIMD
    - Each core itself implements SIMD *AVX*

> Layered combination example 1: MIMD chip over SIMD core.

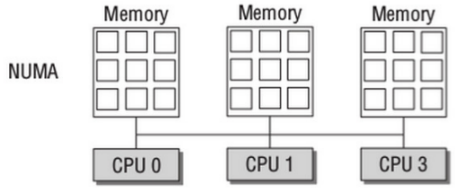
Shared / Distributed Memory

Shared Memory Architecture means using a single *Memory Address Space* for all processors.

- 1 Logical Memory Space, but can have multiple physical banks
- Do not limit the location of and access time to different parts of Memory
    - Uniform Memory Access (**UMA**):



    - Non-uniform Memory Access (**NUMA**):



- Limits *Scalability* due to Bandwidth requirement

Distributed Memory Architecture means each processor can only directly address its own Memory.

- Needs **Message Passing** to retrieve remote data
    - Nodes are connected through links
    - Use switches to achieve dynamic topology
- Large-scaled systems must be *Distributed*, since overhead of providing 1 Logical Memory Space is high

> Layered combination example 2: Distributed-memory cluster over Shared-memory node.

Communications

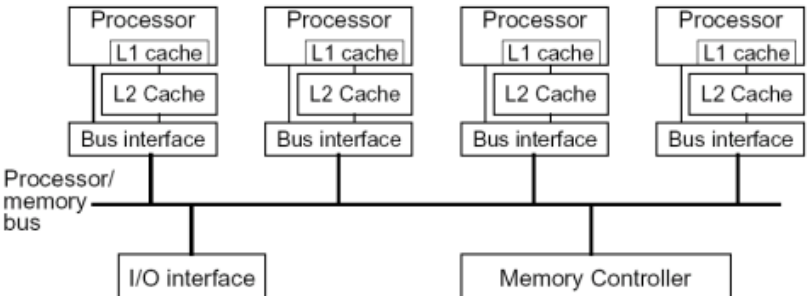
Communication Topologies

Interconnection network is built upon certain topology.

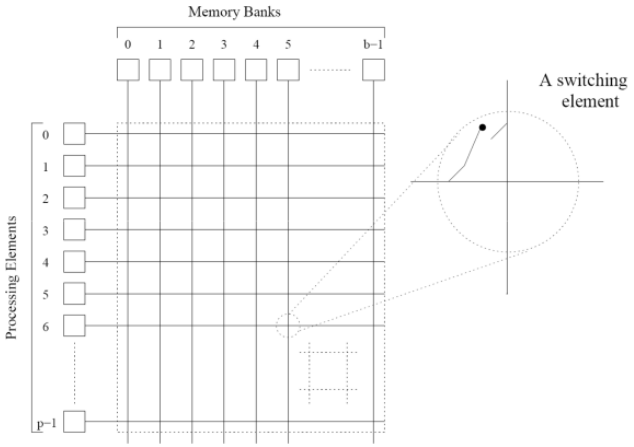
Shared-Memory: Bus Architecture

All Processors can communicate *directly* to each other, and towards Memory through common **Bus** (总线).

- Simple bus:



- *Crossbar* with switches:



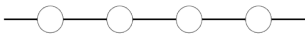
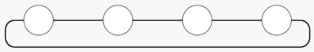
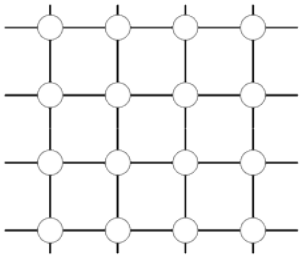
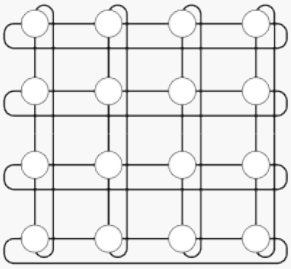
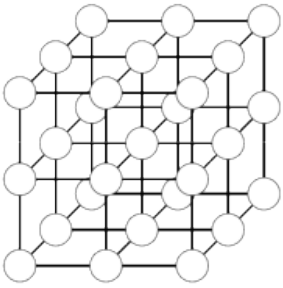
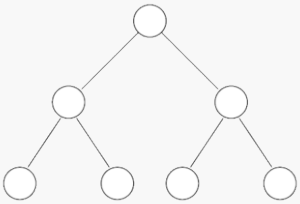
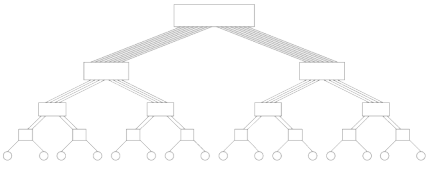
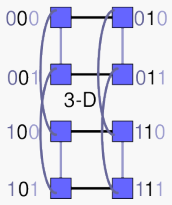
The Bus will become a bottleneck, resulting in limited Bandwidth and Scalability.

Distributed-Memory: Multihop Networks

Parameters of Multihop Networks:

- *Diameter*: Distance between farthest pair of processors
  - $\Rightarrow$ Worst Latency
- *Bisection Width*: Minimum number of links to bipartite the network into two almost equal halves
  - $\Rightarrow$ Potential communication bottleneck
  - *Bisection Bandwidth* $= \sum$ Bandwidth of links cut
- *Cost*: Number of links in the network

Different types of multihop network topologies are summarized below (p denotes # of processors):

Name	Topology	Diameter	Bisection	Cost
Linear Array		$p-1$	1	$p-1$
1-D Ring		$\frac{p}{2}$	2	p
2-D Mesh		$2\sqrt{p}-2$	$\sqrt{p}$	$2p-2\sqrt{p}$
2-D Torus		$\sqrt{p}-1$	$2\sqrt{p}$	$2p$
3-D Crystal		$3\sqrt[3]{p}-3$	$(\sqrt[3]{p})^2$	$3p-3(\sqrt[3]{p})^2$
Tree		$2\log p$	1	p
Fat Tree		$2\log p$	$\frac{p}{2}$	$\frac{p\log p}{2}$
Hypercubes		$\log p$	$\frac{p}{2}$	$\frac{p\log p}{2}$

> Obtaining d-Dimension Hypercubes:
>
> 1. Make two copies of $(d-1)$-Dimension Hypercubes
> 2. Link every corresponding node pairs, and extend it *ID* with 0 and 1 correspondingly
> 3. Every bit of a node's *ID* represents a Dimension

Cache Coherence Problem

To maintain a uniform Memory Space, only 1 logical copy of each variable. However, variables can be in multiple local caches, and the caches must be made **Coherent** through some protocol, to make all processors aware of changes.

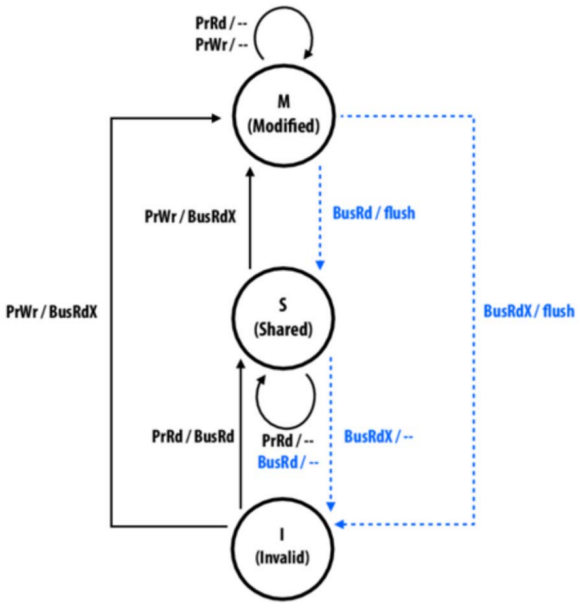
Protocols

Two major types of protocols:

- **Invalidate**: When process p_1 modifies `v`, the value in all other cached copies AND memory declared invalid.
  - Another process's access requires p_1 to write back the new value to memory first
  - Makes Latency $\uparrow$ when others do read
- **Update**: When process p_1 modifies `v`, writes the new value to memory and other caches.
  - Makes Bandwidth waste if others don't read

MSI Protocal: Widely used invalidation-based protocol. Represented by a Finite State Machine.

- Each variable `v`'s copy can be in M (Modified / Dirty), S (Shared), or I (Invalid) state
  - S: `v` may exist in multiple caches; All copies up-to-date
  - M: `v` is dirty here; All other current copies MUST be invalidated (state $\rightarrow I$)
  - I: Someone modified his copy of `v` (state $\rightarrow M$), so value here is out-of-date
- A Read / Write action causes an edge transition, along with signals sent to Memory and other Caches
- FSM Diagram of MSI protocal:



- State change example:

Time ↓

Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x		x = 5, S		x = 5, S y = 12, S
	read y		y = 12, S	
x = x + 1		x = 6, D		x = 5, I y = 12, I
	y = y + 1		y = 13, D	
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I

Implementations

Protocols can be implemented on different types of cache system Hardwares.

- **Snoopy Cache System**: All processes listen on the bus for coherency traffic and respond accordingly; Naive
- **Directory Cache System**: Keep a *Directory* in Memory, indicating which processors hold Dirty / Shared copy of each memory block
  - Need p *Presence Bits* in each entry when p processors
  - When p_0 modifies `v`, only need to sent Invalidation Signal to processors presence in the S entry
  - When p_1 (in I) reads `v`, knows it should invoke p_0 by looking up the M entry

> Directory needs $O(mp)$ extra memory, where $m =$ # of memory blocks, and $p =$ # of processors. Can improve by store the directory distributively among processors.

False Sharing

False Sharing occurs when two Processors are modifying two different variables, who happen to be in the same *Cache Line*. Solve by trying to lay out data in a way that different Processors care about different Cache Lines.

Routing & Its Costs

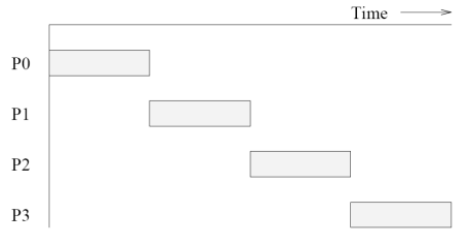
Time to transfer a message consists of:

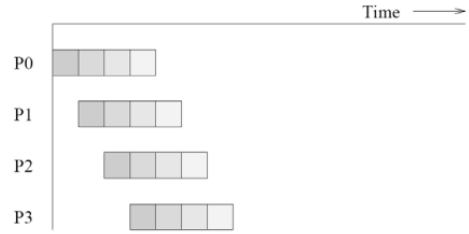
1. Startup Time t_s: Time for preparations (Header adding, ...)
2. Per-hop Time t_h: Time wasted on a router to determine the next hop
3. Per-word Time t_w: Time to actually transfer a word of data through a link

Transferring Strategies

Different transferring strategies have different communication costs. Assume transferring a message of size m over l links:

1. **Store & Forward**: Each node waits to receive entire message before forwarding it

    - Example:


    - Total Cost $= t_s + l(mt_w + t_h) \approx t_s + lmt_w$
2. **Packet Routing**: Break into p packets, each node can forward a packet it received
    - Example:


    - Total Cost $= t_s + (l + p - 1)(\frac{mt_w}{p} + t_h) = t_s + (l + m - 1)(t_w + t_h)$ assuming 1 *Word / Packet*
3. **Cut-through Switching**: Similar to Packet Routing, but
    - Only Packet 1 needs t_h
    - All following packets follow the same path

Considering that t_h << t_w and l << m, Overall Cost $\approx t_s + mt_w$.

> If *Congestion* (= Possible # of Communications / Bisection Width) on link is c, Over Cost $\approx t_s + cmt_w$.

Dimension Routing

If we adopted Mesh / Hypercube Topologies, we can easily order all the links along the dimension axes, thus fix the routing strategy in advance.

E-cube Routing: On Hypercube networks, from $10110 \rightarrow 11101$

1. Compute $10110 \oplus 11101 = 01011$
2. From LSB $\rightarrow$ MSB, make a cross on that dimension if the bit is 1

Dimension Routing's fixed-direction property can prevent Routing *Deadlocks*.

Process to Processor Mapping

The physical communication implementation is determined by Processors Topology (Hardware), BUT actual communication requests issued are determined by Processes (Software). We need to properly **Map** (**Embed**) Processes onto Processors, to maximize the communication efficiency.

- **Congestion** c = # of logical edges mapped onto that physical edge
- **Dilation** d = Max length of physical route between logical neighbors
- *Expansion* = # of Processes / # of Processors; Assume is 1

Goal is to Map a Process Topology T_L onto a Processor Topology T_P, making:

1. Short paths on T_L are also relatively short on T_P
2. Maximal Congestion is as small as possible

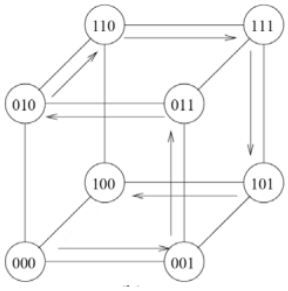
Line on Hypercube

"Mapping a length-2^k Line onto k-dimensional Hypercube"

Gray Codes: An ordering of k-digit binary numbers where neighbors only differ in 1 bit.

> Obtaining Gray codes G_k:
>
> 1. Make a reflected copy of $(k-1)$-bit Gray Codes
> 2. Prepend upper ones (original ones) with 0
> 3. Prepend lower ones (reflected ones) with 1

Ξ Procedure:



1. Mark Hypercube nodes with convention
2. Put Line node i onto Hypercube node $G_k(i)$

$\varkappa$ Max Congestion $= 1$; Dilation $= 1$.

Hypercube on Line

"Mapping k-dimensional Hypercube onto a length-2^k Line"

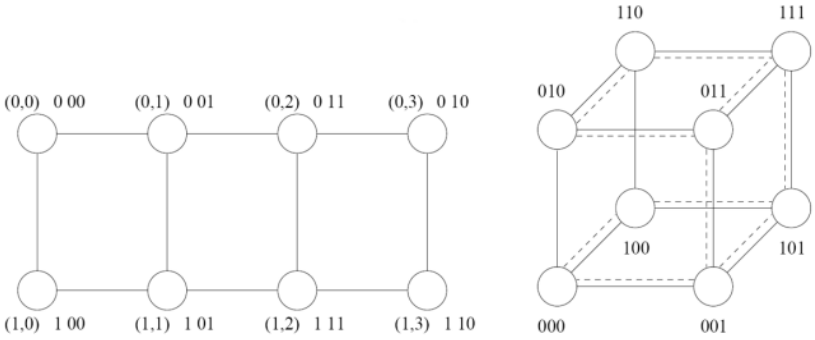
Ξ Procedure: Just the reverse of above.

$\varkappa$ Max Congestion $= 2^{k-1}$; Dilation $= 2^k$.

2-D Mesh on Hypercube

"Mapping a $2^r \times 2^s$ Mesh onto a $(r+s)$-dimensional Hypercube"

Ξ Procedure:



1. Mark Hypercube nodes with convention ($(r+s)$-bits in length)
2. Put Mesh node (i, j) onto Hypercube node $G_r(i) \mid\mid G_s(j)$
   - $\mid\mid$ means *Concatenation*

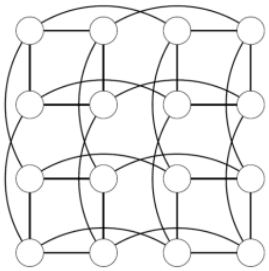
$\varkappa$ Max Congestion $= 1$; Dilation $= 1$.

> Each different Row / Column of the Mesh maps onto a distinct Sub-Hypercube.

Hypercube on 2-D Mesh

"Mapping a $2k$-dimensional Hypercube onto a $2^k \times 2^k$ Mesh"

Ξ Procedure:



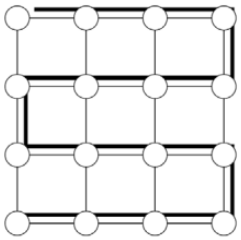
1. Mark Hypercube nodes with convention ($2k$-bits in length)
2. Put Hypercube nodes with the same *last* k digits onto Row r of the Mesh
    - Using Hypercube to Line mapping (Graycode)

$\varkappa$ Max Congestion $= 2^{k-1}$; Dilation $= 2^k$.

Line on Mesh

"Mapping a length-p Line onto a mesh with p nodes"

Ξ Procedure: *Zig-Zag Embedding*



$\varkappa$ Max Congestion $= 1$; Dilation $= 1$.

Mesh on Line

"Mapping a $\sqrt{p} \times \sqrt{p}$ Mesh onto a length-p Line"

Ξ Procedure: Just the reverse of above.

$\varkappa$ Max Congestion $= \sqrt{p}$; Dilation $= 2\sqrt{p} - 1$.

Performance Analysis

Speedup Notations

For a given problem X and a parallel algorithm A, let:

- $T_s =$ Minimum time to solve X on 1 Processor, i.e. Time for best Sequential algorithm
- $T_1 =$ Time that algorithm A takes using 1 Processor, $\geq T_s$
- $T_p =$ Time that algorithm A takes using p Processors

The following notations are introduced for performance speedup analysis:

- **Absolute Speedup** $S_p^* = \frac{T_s}{T_p}$
- **Relative Speedup** $S_p = \frac{T_1}{T_p}$, comparing A with itself on different machine sizes
- **Work** (**Cost**) $W_p = pT_p$, total Time Cycles burned by all Processors, $\geq T_s$
- **Effciency** $E_p = \frac{S_p}{p}$, typically ≤ 1

Hard to achieve Linear Scalability (i.e. $S_p = \Theta(p) \Rightarrow$ hold E_p as a constant), because various kind of **Overheads** (including Communication, Synchronization, Load Imbalance, ...) will be introduced when using more Processors in parallel.

> Sometimes the best sequential algorithm is not parallelizable, but a (sequentially) worse algorithm can be parallelized. This becomes a *Trade-off* between T_s and T_p.

DAG Model

We can represent a Problem as a **Directed Acyclic Graph** (*DAG*, 有向无环图) of Tasks. It should include:

- *Weighted* Node n, representing a Task n, along with its needed Time
- *Weighted Directed* Edge (u, v), representing a dependency that v cannot start until u is done, and along with the cost of Communication between u and v

> NO Cycles in the DAG (since the computation must terminate).

Given the following parameters of Graph G:

- $C = \Sigma$ Node & Edge Weights
- $D =$ Length of the **Critical Path** of G
- $T_p =$ Time taken by p Processors to solve G
- $T_\infty =$ Minimum Time possible to solve G, allowing arbitrary # of Processors

We have two laws that bound the performance:

1. **Work Law**: $W_p = pT_p \geq C \Rightarrow T_p \geq \frac{C}{p}$
2. **Span Law**: $T_\infty \geq D$

Laws & Metrics

There are multiple Laws & Metrics that models the performance gain of parallelization.

- **Amdahl's Law**: $T_p = fT_1 + \frac{(1-f)T_1}{p} \Rightarrow S_p = \frac{T_1}{T_p} = \frac{1}{f+\frac{1-f}{p}} \leq \frac{1}{f}$
  - Solving the same-sized problem in parallel
  - There's f fraction ($0 \leq f \leq 1$) of a problem that is inherently sequential, CANNOT be parallelized
  - Called *Strong Scaling*
- **Gustafson's Law**: $T_p = fT_1 + \frac{p(1-f)T_1}{p} = T_1; T_1' = fT_1 + p(1-f)T_1 \Rightarrow S_p = \frac{T_1'}{T_p} = p - f(p-1)$
  - With p Processors, we solve a problem whose parallelizable part is p-times larger
  - Can view it as Amdahl's Law with $f' = \frac{f}{f+p(1-f)}$
  - More practical, since when we use parallelization we try to solve larger problems
  - Called *Weak Scaling / Scaled Speedup*
- **Karp-Flatt Metric**: $S_p^{stat} = \frac{1}{f+\frac{1-f}{p}} \Rightarrow$ Determine true fraction $f = \frac{\frac{1}{S_p^{stat}} - \frac{1}{p}}{1-\frac{1}{p}}$ by statistically observing S_p^{stat}

    - S_p^{stat} is acquired through experiment results, and f here is dynamic with different p
    - If f increases with p, it means *Overhead* is introduced by parallelization
- **Efficiency**: $W_p(n) = W_1(n) + \gamma(n,p) \Rightarrow E_p(n) = \frac{W_1(n)}{W_1(n)+\gamma(n,p)} = \frac{1}{1+\frac{\gamma(n,p)}{W_1(n)}}$
    - $W_1(n)$ represents the total useful Work needed for size-n Problem
    - $\gamma(n,p)$ means the Overhead from parallelism when we solve size-n Problem with p Processors
        - $p \uparrow \Rightarrow E \downarrow$ ($\gamma(n,p)$ increases)
        - $n \uparrow \Rightarrow E \uparrow$ ($W_1(n)$ increases faster than $\gamma(n,p)$ typically)
        - If $W_1(n) = \Omega(\gamma(n,p))$, this is called **Isoefficiency**

Distributed-Memory Programming Essentials

MPI Programming

MPI is a *Distributed Memory* Programming Interface Standard.

- A Standard, the library itself can have multiple different implementations (OpenMPI, MPICH, IntelMPI, ...)
- For C / C++ / Fortran languages
- Processes each have local memory, and they communicate through Sending & Receiving messages
    - Processes automatically map onto processors
    - Uses **Single Program Multiple Data** (*SPMD*) model
        - All processes run the same piece of program
        - Program itself contains branchings / selections to differentiate processes

For details about MPI Programming, refer to MPI 在线教程.

Collective Communication Details

Communications are internally *Point-to-point*, but efficient **Collectives** (e.g. `gather`) can be implemented by parallelizing these Point-to-point communications. On different hardware topologies and for different Collective operations, we have different optimal solutions.

We suppose:

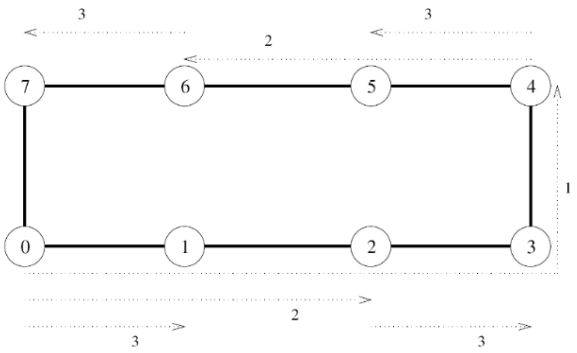
- Per-hop time t_h is very small compared to t_w, so sending over multiple hops *does not* affect latency, still $t_s + mt_w$
- However, *Congestion* on a link *affects* latency, $t_s + cwt_w$
    - Sending two different messages over one link in the *same direction* causes Congestion
    - Assume each link is *bi-directional*, so messages back and forth on one link does not cause Congestion

> Lines are then equivalent to Rings.

Broadcast on Ring

"Broadcast on a length-p Ring"

Ξ Procedure:



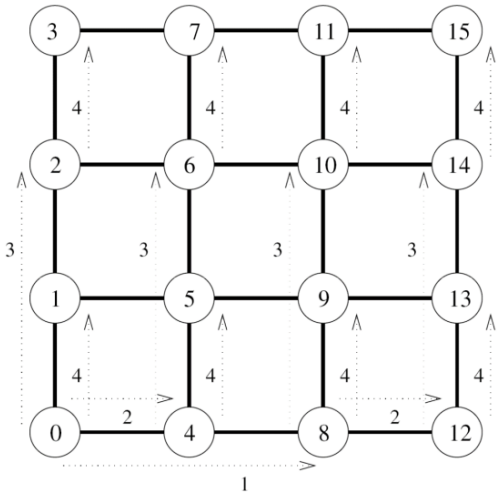
↝ Total time $= (t_s + mt_w) \log p$.

> Reduction on the same topology is just the reverse procedure.

Broadcast on Mesh

"Broadcast on a $\sqrt{p} \times \sqrt{p}$ Mesh"

Ξ Procedure:



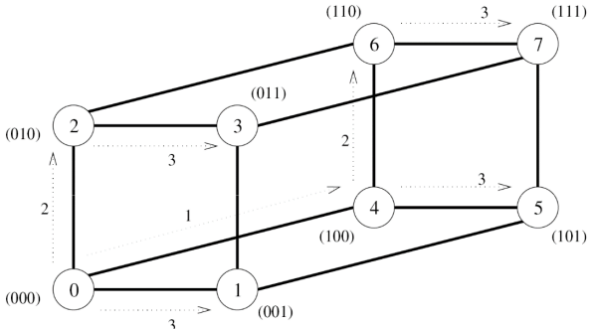
1. The Root first broadcast on the row, $\log \sqrt{p} = \frac{\log p}{2}$ steps
2. Then the row broadcast on their columns, $\log \sqrt{p} = \frac{\log p}{2}$ steps

↝ Total time $= (t_s + mt_w) \log p$.

Broadcast on Hypercube

"Broadcast on a Hypercube with p Processors"

Ξ Procedure:



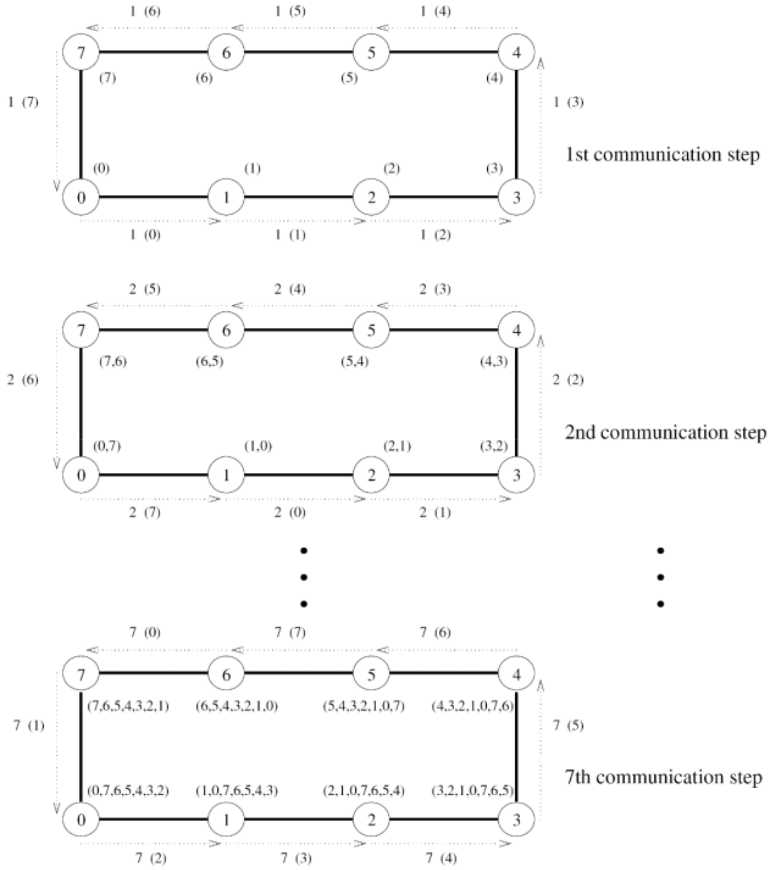
- One Dimension per step, in total $\log p$ steps

$\varkappa$ Total time $= (t_s + mt_w) \log p$.

All-to-all on Ring

"All-to-all on a length-p Ring"

Ξ Procedure:



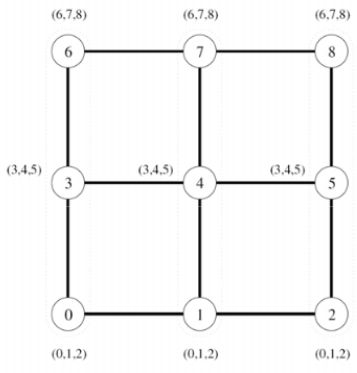
- Keep passing size-m data down the Ring
- Process store any new data they receive, so repeat $p - 1$ steps then done

$\varkappa$ Total time $= (t_s + mt_w)(p - 1)$.

All-to-all on Mesh

"All-to-all on a $\sqrt{p} \times \sqrt{p}$ Mesh"

Ξ Procedure:



(b) Data distribution after rowwise broadcast

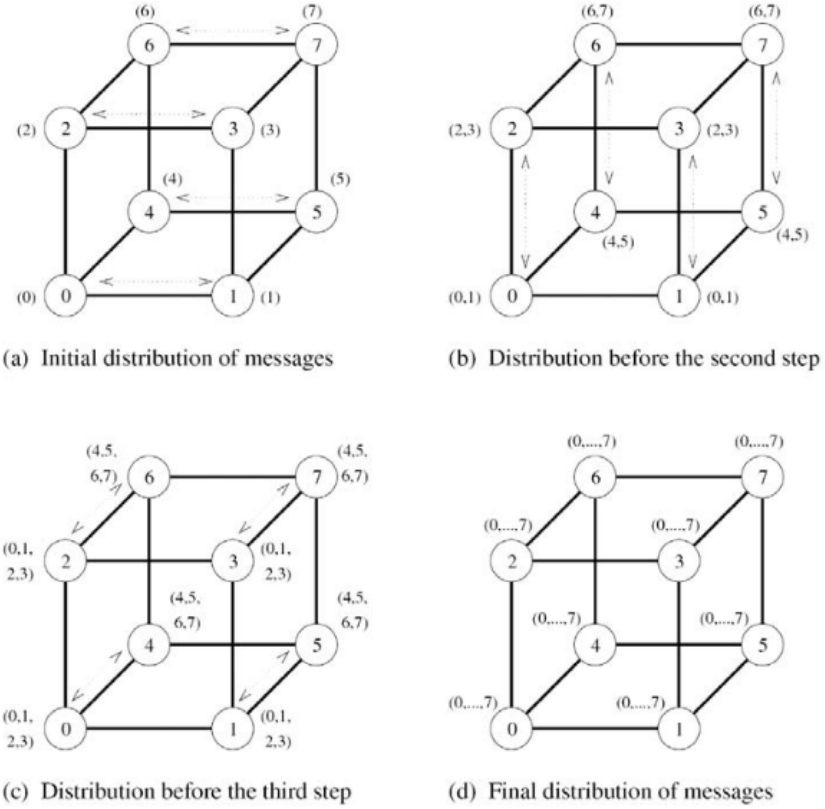
1. First do All-to-all on each row, using the above Ring scheme
2. Second do All-to-all on each column
    - Notice size of each message is $m\sqrt{p}$ now

$\varkappa$ Total time $= (t_s + mt_w)(\sqrt{p} - 1) + (t_s + m\sqrt{p}t_w)(\sqrt{p} - 1) = 2t_s(\sqrt{p} - 1) + mt_w(p - 1)$.

All-to-all on Hypercube

"All-to-all on a Hypercube with p Processors"

Ξ Procedure:



(a) Initial distribution of messages

(b) Distribution before the second step

(c) Distribution before the third step

(d) Final distribution of messages

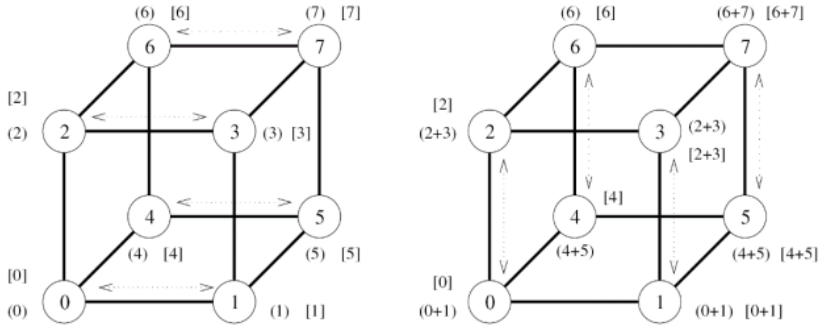
- Exchange on one dimension per step, but size of message to transfer doubles

$\varkappa$ Total time $= \Sigma_{i=1}^{\log p}(t_s + m2^{i-1}t_w) = t_s \log p + mt_w(p - 1)$.

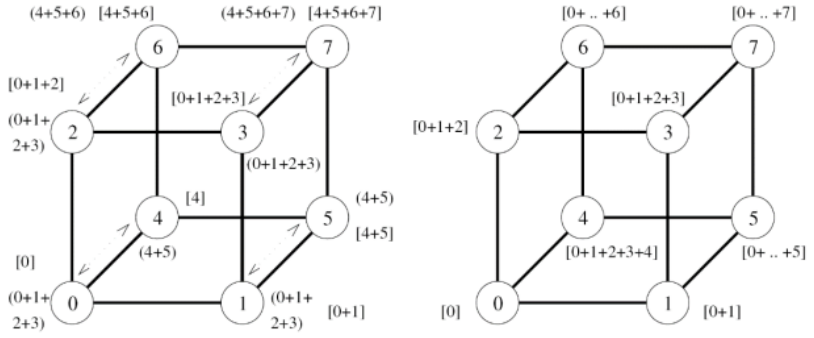
Prefix Sum on Hypercube

"Prefix sum on a Hypercube with p Processors"

Ξ Procedure:



(a) Initial distribution of values (b) Distribution of sums before second step



(c) Distribution of sums before third step (d) Final distribution of prefix sums

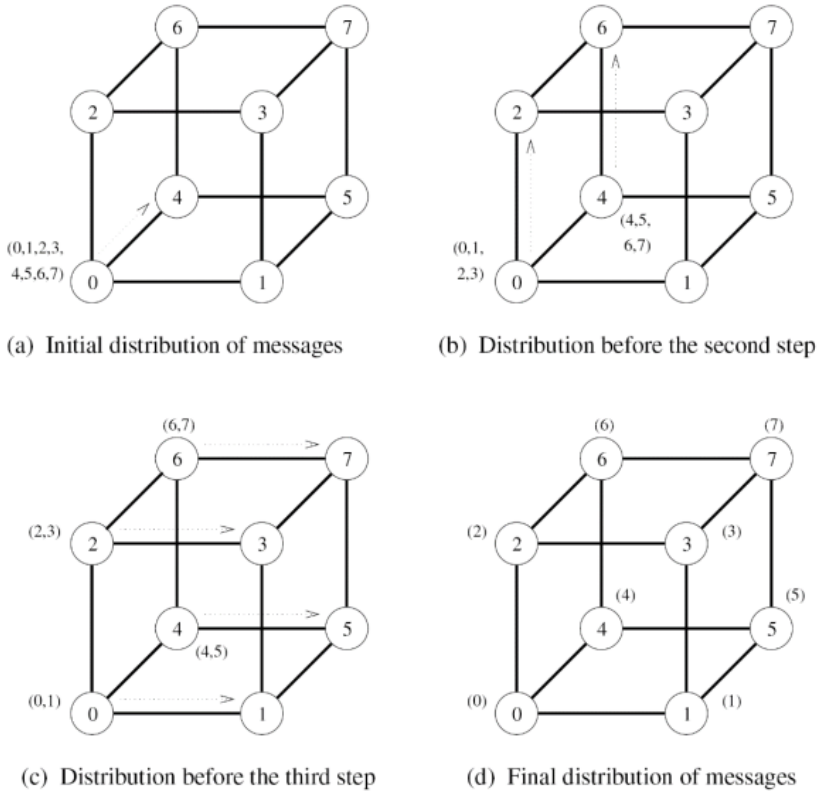
- Exchange on one dimension per step, size of message always m (the sum)
- A Process adds the thing it receives only when receiving from smaller node

$\varkappa$ Total time $= (t_s + mt_w) \log p$.

Scatter on Hypercube

"Scatter on a Hypercube with p Processors"

Ξ Procedure:



(a) Initial distribution of messages (b) Distribution before the second step

(c) Distribution before the third step (d) Final distribution of messages

- One Dimension per step, but size of message to transfer halves

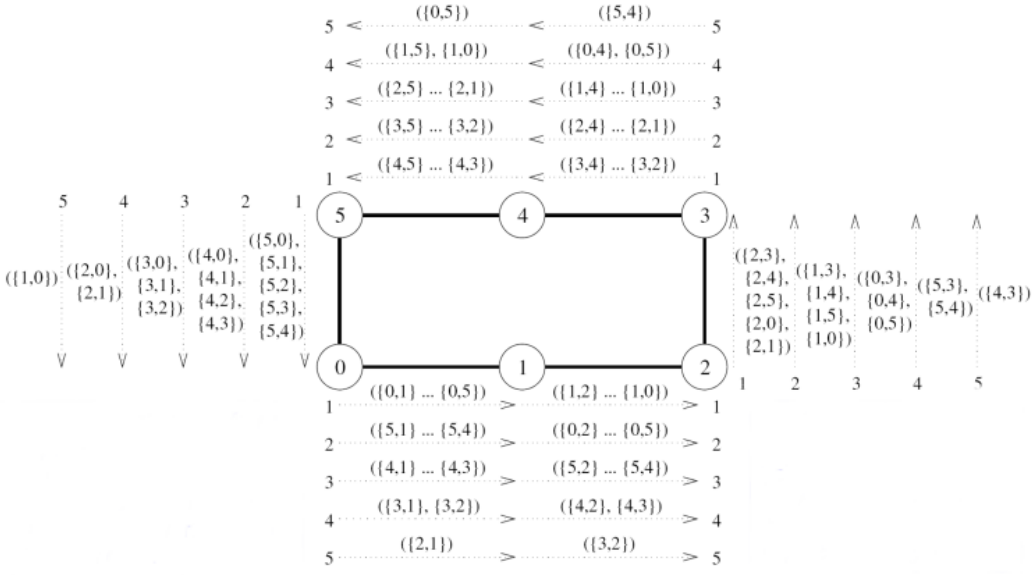
$\varkappa$ Total time $= \Sigma_{i=\log p}^{1}(t_s + m2^{i-1}t_w) = t_s \log p + mt_w(p-1)$.

> Gather on the same topology is just the reverse procedure.

Personalized on Ring

"All-to-all personalized on a length-p Ring"

Ξ Procedure:



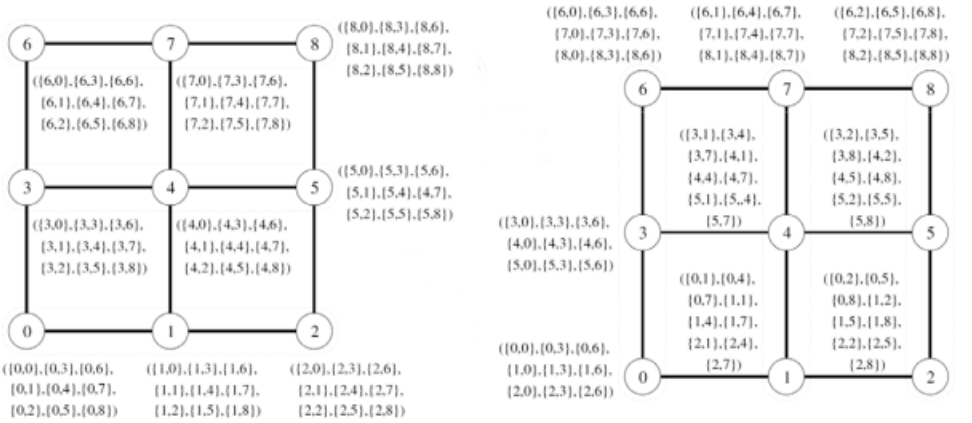
- One Dimension per step, but size of message decrease by m every step

$\varkappa$ Total time $= \Sigma_{i=p-1}^{1}(t_s + m(p-i)t_w) = (t_s + m\frac{p}{2}t_w)(p-1)$.

Personalized on Mesh

"All-to-all personalized on a $\sqrt{p} \times \sqrt{p}$ Mesh"

Ξ Procedure:



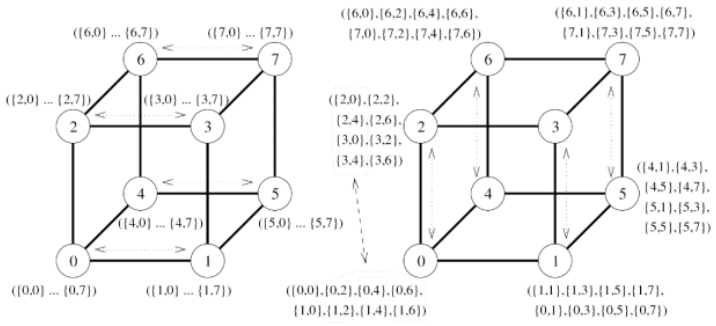
1. First, each node splits its outgoing data to $\sqrt{p}$ groups, each for a column; Then do personalized on the rows, with message size mp but only to $\sqrt{p}$ nodes
2. Second, do similar personalized on the columns

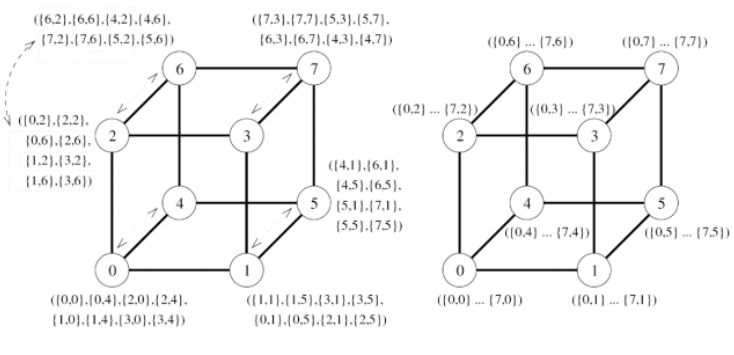
ϰ Total time $= 2(t_s + m\frac{p}{2}t_w)(\sqrt{p}-1) = (2t_s + mpt_w)(\sqrt{p}-1)$.

Personalized on Hypercube

"All-to-all personalized on a Hypercube with p Processors"

Ξ Procedure:

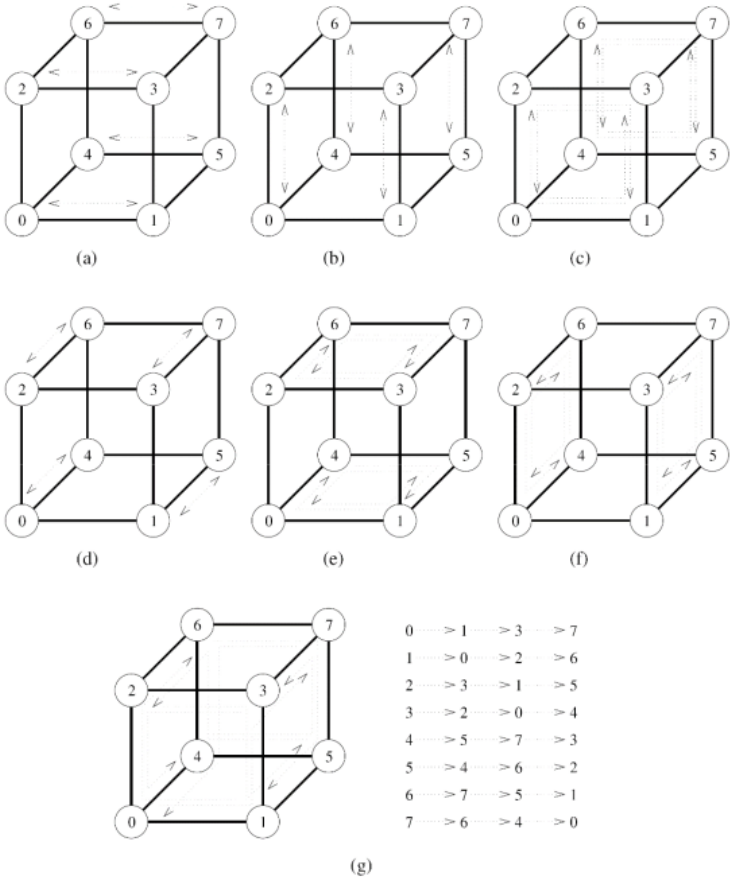




- Exchange size-$m\frac{p}{2}$ data on one Dimension each step, directing messages for the opposite half

↗ Total time $= (t_s + m\frac{p}{2}t_w)\log p$. *NOT Optimal*.

Ξ Procedure:



- In step j, Process i sends message to Process $i \oplus j$
  - Using *E-cube* Routing for such sending
  - No Congestion will happen
- Repeat $p-1$ steps

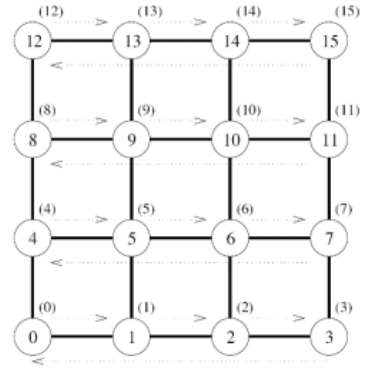
↗ Total time $= (t_s + mt_w)(p-1)$. *Optimal*.

> The lower bound of such comunication $= \frac{\text{Data size each Process receives} \times \text{Average distance between Processes}}{\text{Total \# of links}}$

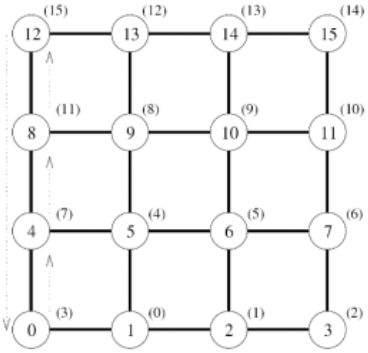
Circular Shift on Mesh

"Circular shift of k on a $\sqrt{p} \times \sqrt{p}$ Mesh", each node sends a message to node $(p+k) \bmod p$

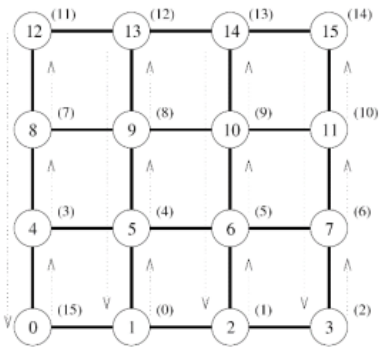
Ξ Procedure:



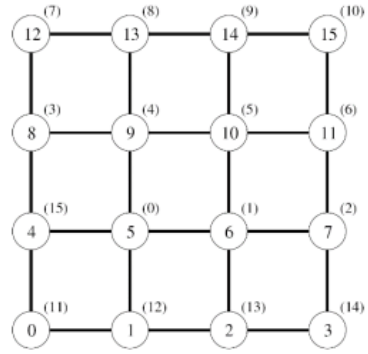
(a) Initial data distribution and the first communication step



(b) Step to compensate for backward row shifts



(c) Column shifts in the third communication step



(d) Final distribution of the data

1. First settle the shifts along the rows
    1. Do a circular shift on the rows
    2. Do a *Compensate* shift on the *First* column
    3. Repeat $k \bmod p$ times
2. Second settle the shifts along the columns, need to repeat $\lfloor \frac{k}{\sqrt{p}} \rfloor$ times

$\varkappa$ Total time $= (t_s + mt_w)(\sqrt{p} - 1)$, since movement on either Dimension is at most $\sqrt{p}$.

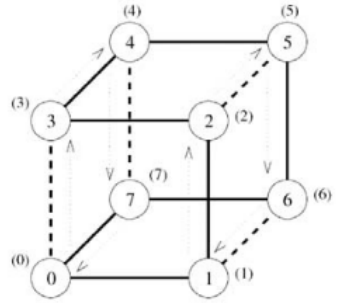
Circular Shift on Hypercube

"Circular shift of k on a Hypercube with p Processors"

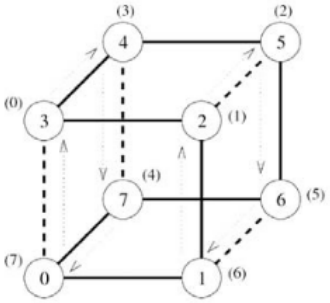
For any two nodes differing by $2^i, i > 0$, they are 2-links away in the Hypercube.

> Can be proved by Gray-code Construction.

Ξ Procedure:

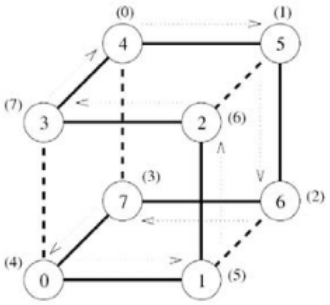


First communication step of the 4-shift

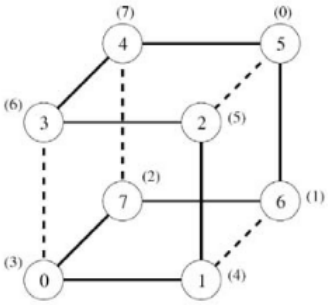


Second communication step of the 4-shift

(a) The first phase (a 4-shift)



(b) The second phase (a 1-shift)



(c) Final data distribution after the 5-shift

- Write k in binary, then do shifts on Dimensions with digit-1
  - For Dimension $i > 0$, shift twice can make 2^i difference
  - For Dimension $i = 0$, only shift once

$\varkappa$ Total time $= (t_s + mt_w) \log p$, since shifting along at most $\log p$ Dimensions.

Shared-Memory Programming Essentials

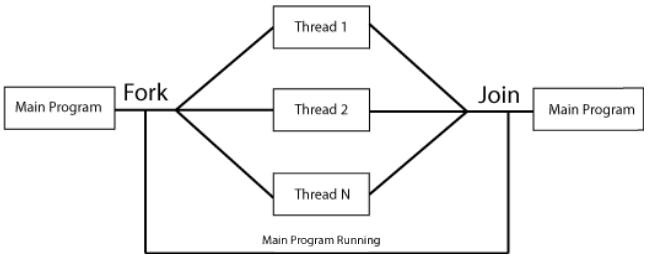
Threads Interfaces

Shared-memory programming relies on Threads. Different platforms follow different Thread interface standard:

- POSIX Thread (*pthread*)
- JAVA Multithreading
- ...

They directly provide programmers the ability to manually manipulate Threads, but programmers must handle everything about the threads.

Multithreading usually follows the **Fork-Join** Model:



OpenMP Programming

OpenMP is a higher-level *Shared Memory* programming technique.

- Programmers use it by adding *Compiler Directives*
- It wraps the lower-level multithreading operations
  - ↑ Easier to use
  - ↓ Loses some flexibility

For details about OpenMP Programming, refer to OpenMP 参考手册.

PGAS Languages

Partitioned Global Address Space (PGAS) is another model for Shared-memory programming. It includes a number of languages: Unified Parallel C (UPC), Coarray Fortran (CAF), ...

Global address space is divided into partitions, and each thread has an affinity to one partition. Not widely used, so omitted here.

Synchronization

In the Shared-memory scenario, different Threads interact with the same memory. Therefore, routines that access the same piece of data will introduce the **Synchronization** problems.

- A routine is *Thread-safe* if it can be called by multiple threads simultaneously, and ALWAYS produces correct results
- If not Thread-safe, this routine must be put in a **Critical Section**, allowing at most one thread executing it at the same time (i.e. ensuring **Mutual Exclusion**)

Eliminating Concurrenct Bugs

There are a bunch of ways to help programmers achieve mutual exclusion:

1. Semaphores, **Locks** (0-1 Semaphore), Conditional Variables ⋆
   - Blocking locking
   - Non-blocking locking, test first and does something else if lock is currently not available
2. Transactional Memory
   - Hardware (HTM)
   - Software (STM)
3. Own concurrent algorithm w/o SW & HW support

Critical Section Criteria

A solution to Critical Section (*CS*) problem should satisfy the following criteria:

1. **Mutual Exclusion**
2. *Progress* (**Deadlock Freedom**): If no process is in CS and several want to get in, at least one should succeed
3. *Bounded Waiting* (**Wait Freedom**): If a process is waiting to get in CS, it should succeed in finite time

Implementation of Locks

> We suppose Memory is *Sequentially Consistent* here.

Software Solutions

There are theoretical Software solutions, which do not need Hardware support, but are extremely inefficient.

Peterson's Mutex Algorithm

For two threads with ID 0 and 1.

```
void lock() {
    int i = threadID;
    int j = 1 - i;
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i);
}

void unlock() {
    int i = threadID;
    flag[i] = false;
}
```

Lamports's Bakery Algorithm

For multiple threads.

```
Initially: flags are all false, labels are all 0;

void lock() {
    int i = threadID;
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (Exists k < i: flag[k] && label[k] < label[i]);
}

void unlock() {
    int i = threadID;
    flag[i] = false;
}
```

Hardware Supports

With Hardware atomic instruction supports:

1. *Boolean* `TAS(x)`: Set `x` to true; Return its original value
2. *Boolean* `CAS(x, v, v')`: If `v == v'`, set `x` to `v`; Return `x`'s current value
3. `x.getAndSet(v)`: Set `x` to `v` (allow non-boolean); Return its original value
4. `x.getAndIncrement()`: Increment `x`; Return its original value
5. `x.compareAndSet(x', v)`: If `x == x'`, set `x` to `v`; Return `x == x'`

We can build locks with practical performance.

Test-and-Set Locks

```
Initially: state = false;

void lock() {
    while (TAS(state));
}

void unlock() {
    state.set(false);
}
```

TTAS (Test-Test-and-Set) Locks

Test first so that repeatedly reading from cached copy of lock value, thus improves peformance.

```
Initially: state = false;

void lock() {
    while (true) {
        while (state.get());
        if (!TAS(state)) return;
    }
}

void unlock() {
    state.set(false);
}
```

Backoff Locks

Set thread to sleep a while when others are in CS.

```
Initially: state = false;

void lock() {
    while (true) {
        while (state.get());
        if (!TAS(state)) return;
        else sleep(DELAY);
    }
}

void unlock() {
    state.set(false);
}
```

Anderson's Queue Lock

For limited number of threads (< `size`), since the queue is a circular array.

```
Initially: tail = 0, flags are all false;

void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlot.set(slot);
    while(!flag[slot]);
}

void unlock() {
    int slot = mySlot.get();
    flag[slot] = false;
    flag[(slot+1) % size] = true;
}
```

CLH Queue Lock

Efficient except for cacheless NUMA architectures, since spinning on predecessor's `pred.locked`.

```
Initially: tail points to a dummy false QNode;

void lock() {
    myNode.locked = true;
    QNode pred = tail.getAndSet(myNode);
    myPred.set(pred);
    while (pred.locked);
}

void unlock() {
    myNode.locked = false;
    myNode.set(myPred);
}
```

MCS Queue Lock

Avoids spinning on predecessor, but incurs more reads and writes, and requires CAS.

```
Initially: tail points to NULL;

void lock() {
    tail.getAndSet(myNode);
    if (pred != NULL) {
        myNode.locked = true;
        pred.next = myNode;
        while (myNode.locked);
    }
}

void unlock() {
    if (myNode.next == NULL) {
        if (tail.compareAndSet(myNode, null)) return;
        while (myNode.next == NULL);
    }
    myNode.next.locked = false;
```

```
18      myNode.next = NULL;
19  }
```

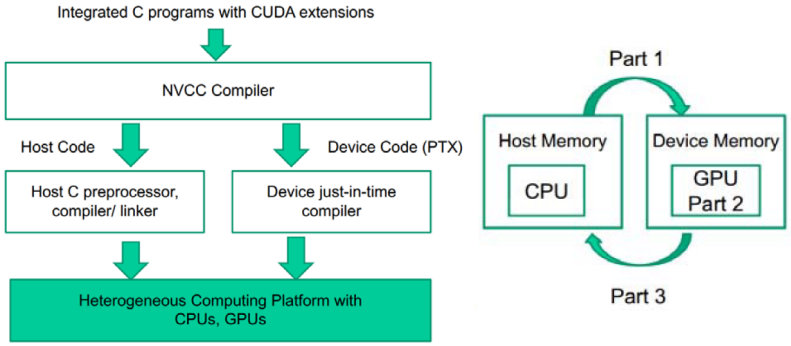
GPU Programming Essentials

GPU Programming Model

General Purpose GPUs (GPGPU) follow *Manycore* architecture. Longer Latency per task (∵ each core simpler), but much higher Throughput per unit time (∵ thousands of cores) ⇒ Best for problems with intense data parallelism.

Popular GPU programming platform is *Nvidia CUDA* (Compute Unified Device Architecture) / *OpenCL* / *OpenACC*.

Host + *Device* Model:



CUDA Programming

CUDA architecture defines the following mapping from Software to Hardware:

Software	— **Mapping** →	**Hardware**
Kernel **Grid**	— Start a **Kernel** on GPU Card(s) →	GPU Card
Block	— 1 / more Block(s) onto 1 SM →	Stream Multiprocessor (**SM**)
Warp	— 1 SM executes 1 Warp (32 Threads) per SIMD cycle →	
Thread	— 1 Core is executing 1 Thread at certain time →	Core

- An SM has its local shared memory and registers ⇒ Thread switching is fast
  - *Gigathread Engine* assigns Blocks to SMs
  - Per SM *Scheduler* responsible for managing threads in this SM
- Synchronizations:
  - Different Blocks in a Kernel may execute in any order
  - Different Threads in a Block can do `__syncthreads()` barrier
  - Different Threads in a Warp executes simultaneously

Common Syntax

```
__global__ void kernelFunc(args);
cudaMalloc((void **) &d_x, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(x, d_x, size, cudaMemcpyDeviceToHost);
kernelFunc<<<ceil(n/t),  t>>>(args);
cudaFree(d_x);
```

For details about CUDA Programming, refer to CUDA 参考手册.

Thread Mapping

Threads / Blocks mapping can both be 1-D, 2-D, or 3-D.

1-D mapping of an array:

```
/* Start kernel. */
kernelFunc<<<ceil(n/t),  t>>>(args);

/* Fetch index. */
idx = blockIdx.x * blockDim.x + threadIdx.x;
```

2-D mapping of a matrix:

```
/* Start kernel. */
dim3 dimGrid(WIDTH / TILE_WIDTH, WIDTH / TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
kernelFunc<<<dimGrid, dimBlock>>>(args);

/* Fetch index. */
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
```

> Matrix can be stored in two different orders:
>
> 1. *Row-major*: $A_{row,col}$ = `A[row * n + col]`
> 2. *Column-major*: $A_{row,col}$ = `A[row + col * n]`

Choosing Block Size

Three considerations when choosing the right block size:

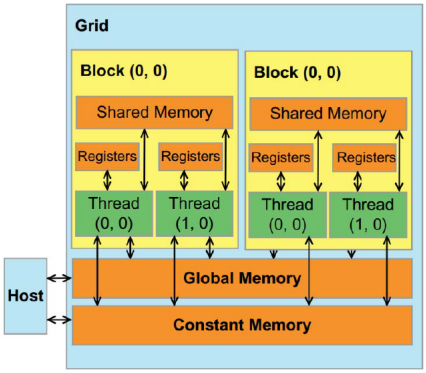
- Hardware restrictions:
  - Max # of Threads assigned to an SM
  - Max # of Blocks assigned to an SM
  - Max # of Threads per block
- Available registers per SM $\Rightarrow$ *Occupancy* = # of Threads per SM / Max # of Threads per SM
- Thread work imbalance $\Rightarrow$ Avoid huge Blocks

> Normally we choose 16×16-sized Blocks.

GPU Memory & Warps

Memory Organization

Nvidia GPU memory architecture:



To address Latency issues, use *Massive Multi-Threading* (**MMT**) strategy that when one thread is doing global memory access or I/O, switch to another thread.

- $\Rightarrow$ Long Latency per thread, but high overall Throughput
- This is the reason why we usually assign ~1,000 threads per SM

To address Bandwidth issues, *Compute to Global Memory Access Ratio* (**CGMA**) = # of floating point operations : # of memory operations needed.

- CGMA = 3 $\Rightarrow$ 200 GB/s Bandwidth brings 200 / 4 $\times$ 3 = 150 GFLOPS << Theoretical peak
- Therefore need to exploit *Data Reuse*, e.g.
  - Tiled ($m \times m$) matrix multiplication, brings m times performance improvement
  - Using registers / shared memory to cache temporal results, and write to global on at the end

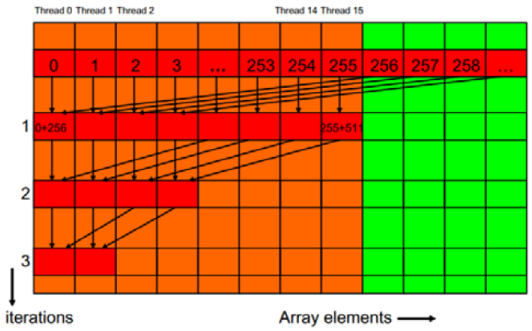
Thread Warps

The unit of "SIMDness" is a Thread **Warp** of 32 Threads:

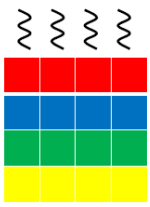
- Ideally, 32 Threads in a Warp do the same thing in a cycle
- If there are branches, Threads who branch execute in the cycle, while others idle

Warps are efficient if the following conditions are met:

- In code logic, fewer **Control Flow Divergence**
  - e.g.

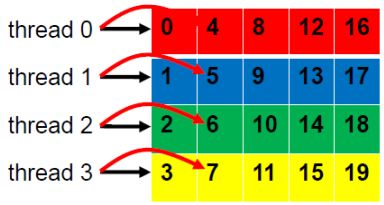


- *In global memory access*, exploit **Memory Coalescing**: Threads in a Warp access nearby locations
  - e.g.



  - In CPU, such pattern will cause *False Sharing* because of the existence of cache, but

    - In GPU, it reduces memory transfer size so that improves performance
- *In shared memory access*, avoid **Bank Conflicts**: Threads in a Warp access different banks
    - e.g.



    - Because simultaneous accesses to one bank will be delayed to multiple cycles

CUDA Intrinsics

CUDA provides several intrinsic instructions for programmers:

- Atomics: `atomicInc`, `atomicAdd`, `atomicMax`, `atomicExch`...
    - To improve performance with atomics:
        1. Split the problem into local ones
        2. Gather local results into a global result
- Warp-aggregations: `__ballot(v)`, `__ffs(mask)`, `__popc(mask)`, `__shfl(res, leader)`, `__shfl_up(res, delta)`, `__shfl_xor(res, mask)`, ...
    - Since Warps are SIMD, intrinsics do not need `__syncthreads()`

Scenary: Prefix Sum

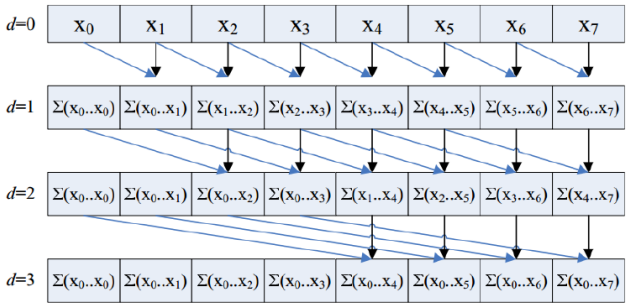
Prefix Sum (*Scan*) is the procedure from $[x_0, x_1, \dots, x_{n-1}] \Rightarrow [x_0, x_0 + x_1, \dots, x_0 + \cdots + x_{n-1}]$. It is very useful in many applications, because $+$ can be replaced by any associative operation.

> In practical conditions using P degree of multiprocessing, an algorithm of S phases which does W total work will at least take $max(S, W/P)$ steps to finish. So by "*Efficient*" we mean Work-efficient.

Sequential Prefix Sum algorithm obviously takes $O(n)$ time.

Naive Algorithm

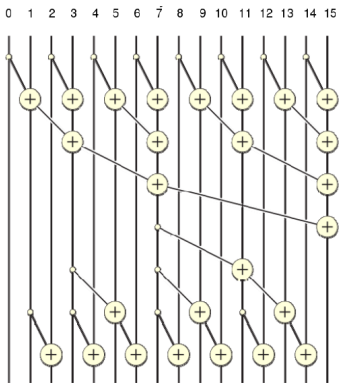
Ξ Procedure: (only needs 2 buffers)



ϰ Phases $= \log n$; Work $= O(n \log n)$.

Efficient Algorithm

Ξ Procedure:

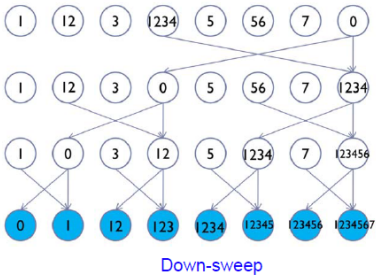


$\varkappa$ Phases $= 2\log n$; Work $= 2n = O(n)$.

Exclusive Scans

"Do a Prefix Sum without adding the value of oneself"

Ξ Procedure:

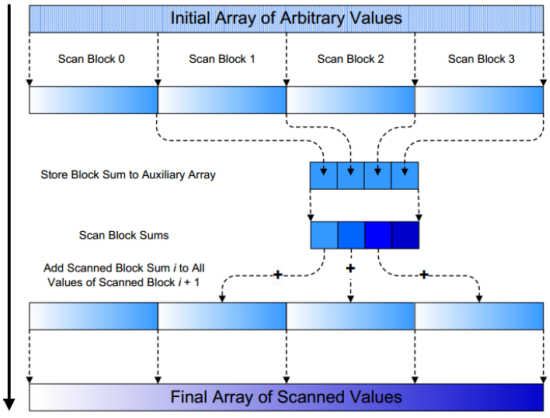


1. Up-sweep is the same as Inclusive Scans
2. At Down-sweep:
    1. First set the final sum to 0
    2. Then perform a half-butterfly downwards

Arbitrary Input Size

"Do a Prefix Sum with arbitrary input size"

Ξ Procedure:



Segmented Scan

"Do a Segmented Prefix Sum"

Define the new operation: $c_1 \odot c_2 = c_1 + c_2$ when in one segment, or c_2 when in two separate segments.

Scenery: BFS

Breadth-First Search (*BFS*) is also widely used in various problems.

- A graph residing in Graphic Memory may contain ~100,000,000 Edges
- Average degree is low ⇒ Sparse graphs
- *Diameter* = farthest distance between node pairs

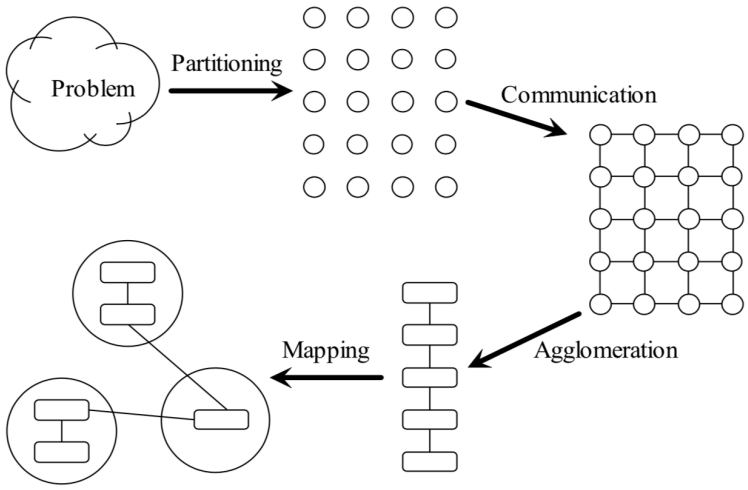
Cutting-edge parallel BFS combines the following three techniques:

1. Load-balanced Gathering:
    - For small # of neighbors, use Prefix Sum gathering
    - For moderate # of neighbors, use a Warp
    - For large # of neighbors, use an entire Block
2. Avoid adding previously visited nodes
3. Duplicate culling using hash table
    - Won't remove all duplicates, but effective

Parallel Algorithm Design

Foster's Methodology

Key idea: Delaying machine-dependent considerations until later steps.



1. **Partitioning**: Partition the application into a number of tasks that can execute in parallel
    - Types:
        - **Data partitioning** (*Domain decomposition*): Data divided into parts
        - **Algorithmic partitioning** (*Functional decomposition*): Algorithm divided into pipelined tasks
    - Attentions:
        - Number of tasks at least 10x of processors
        - Tasks roughly the same size
        - Number of tasks increasing with problem size
2. **Communication**: Determine what data is passed between what tasks
    - Attentions:
        - Should be balanced among tasks, no bottlenecks

        - Each task tries to overlap computations and communication
3. **Agglomeration**: Grouping tasks into larger supertasks, eliminate communication between primitive tasks that agglomerated into one supertask
    - Attentions:
        - Increase locality
        - Supertasks should have similar computational and communication costs
4. **Mapping**: Assigning supertasks to actual processors
    - Goals:
        - Balance the load, maximize processor utilization
        - Minimize interprocessor communication
    - Types:
        - *Static allocation*: Pre-define the task allocation
        - *Dynamic allocation*: e.g. *Work Stealing*; Allocator should not become a bottleneck

Scenery: Floyd-Warshall APSP

Floyd-Warshall all-pairs shortest path algorithm can be coded as:

```
for (k = 0; k < n; k++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i, j] = min(a[i, j], a[i, k] + a[k, j]);
```

In each outer iteration k, $A_{i,k} = min(A_{i,k}, A_{i,k} + A_{k,k}) = A_{i,k}$ does not change; Same for $A_{k,j} \Rightarrow$ Partition into n^2 independent tasks, each holding 1 value of the matrix.

For a given k, $A_{i,j}$ depends on $A_{i,k}$ and $A_{k,j}$, therefore each $A_{i,k}$ broadcasts its value to its row i, and each $A_{k,j}$ broadcasts its value to its column j.

Agglomerate into *row-strip* / *column-strip* supertasks. For example in row-strip, no need to broadcast $A_{i,k}$ to its row, but still needs to broadcast every $A_{k,j}$ to its column $\Rightarrow$ Owner of row k should use `MPI_Broadcast` to broadcast row k to other supertasks.

Mapping will just be one supertask per computing node.

> Speedup is $O(p)$.

Loop Parallelism

Data Dependence Analysis

Let S_1 and S_2 be two statements in a sequential execution, they can have three types of dependencies:

- $S_1 \overset{T}{\longrightarrow} S_2$, *True Dependence (RAW)*, S_1 writes to a location that will be read by S_2
- $S_1 \overset{A}{\longrightarrow} S_2$, *Anti Dependence (WAR)*, S_1 reads to a location that will be written by S_2
- $S_1 \overset{O}{\longrightarrow} S_2$, *Output Dependence (WAW)*, S_1 writes to the same location written by S_2

All dependent statements must run in the same order as sequential, and must be on the same processor.

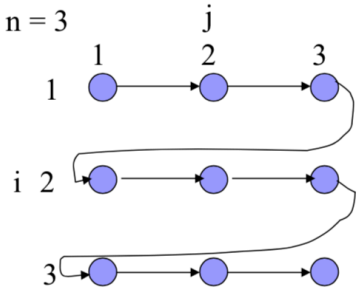
Loop Dependence Analysis

Denote $S[i, j, \ldots]$ as statement S in iteration $i, j, \ldots$. Two types of loop dependence:

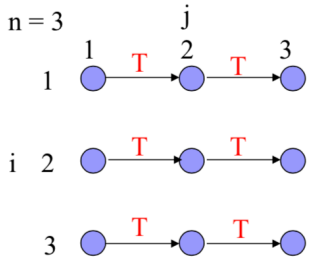
1. **Loop-carried** Dependence: Dependence exists across different iterations, e.g. $S_1[i] \overset{T}{\longrightarrow} S_1[i+1]$
2. **Loop-independent** Dependence: Dependence exists within the same iteration, e.g. $S_1[i] \overset{T}{\longrightarrow} S_2[i]$

Also two kinds of iteration graphs to help us extract loop parallelism:

- **Iteration-space Traversal Graph** (*ITG*):



- **Loop-carried Dependence Graph** (*LDG*):



There are several criterion that may guide us to parallelize nested loops. First, we can parallelize totally-independent components in LDG, by letting them execute on different processors simultaneously. Second, for loop with strongly dependent LDG, we can split the statements into 2 parts: dependent parts + independent parts (i.e. split the loop into two loops). Then the independent parts can benefit from parallelism. This is called "**Loop Fission**".

Example:

```
// Before Loop Fission
for (i = 0; i < n; ++i)
  a[i] = a[i-1] + b[i] * c[i] + d[i];

// After Loop Fission & Parallelization
#pragma omp parallel for schedule(static)
for (i = 0; i < n; ++i)
  temp[i] = b[i] * c[i] + d[i];
for (i = 0; i < n; ++i)
  a[i] = a[i-1] + temp[i];

// Better solution using OpenMP to avoid temporary storage
#pragma omp parallel for ordered private(t) schedule(static)
for (i = 0l i < n; ++i) {
  t = b[i] * c[i] + d[i];
  #pragma omp ordered
  a[i] = a[i-1] + t;
}
```

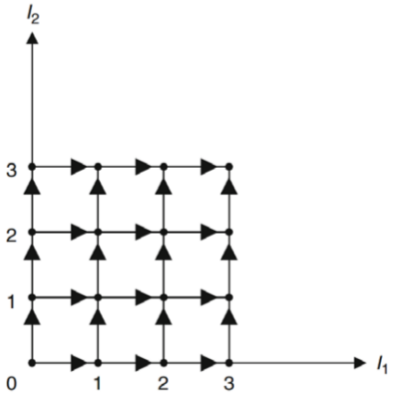
Distance & Direction Vectors

For every loop dependence edge from iteration $T_1 \rightarrow T_2$, we can use the following vectors to describe it:

- **Distance Vector** = $T_2 - T_1$, e.g. $(1, -2)$
- **Direction Vector** = $sign(T_2 - T_1)$, e.g. $(+, -)$
    - Only $(+,+), (+,0), (+,-), (0,+), (0,0)$ are valid direction vectors
    - $(0,-), (-,+), (-,0), (-,-)$ are invalid, because otherwise we depend on a value yet to be computed

Based on direction vectors, we can transform or create new loop indices to make parallelization possible in nested loops. This is called "**Loop Skewing**".

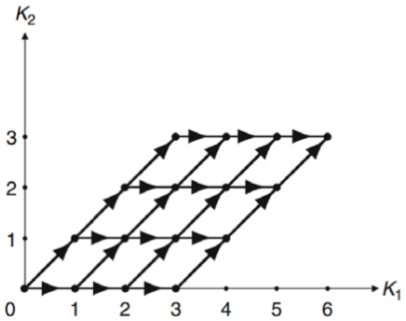
Example: Original nested loop LDG



We use a *Unimodular* linear transformation $U = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ to right-multiply the loop iterations. This gives new indices $(K_1, K_2) = (I_1, I_2)U = (I_1 + I_2, I_2)$. Two theorems that ensures correctness:

- If for every original distance vector v, vU is still valid, then the transformation is correct
- If for every result direction vector v, the d-th coordinate is all $+$, then all inner loops deeper than level d are independent (can run in parallel)

The nested loop LDG new becomes:



New loop bounds can be calculated: $0 \le K_1 \le 6$ and $max(0, K_1 - 3) \le K_2 \le min(3, K_1)$.

Algorithmic Analysis

If there is no way to restructure a loop to increase its parallelism, we should consider restructure the algorithm to eliminate dependences.

- Need to understand the purpose of algorithm well
- For non-deterministic / approximate algorithms, ignoring some dependences may still give valid results

Load Balancing & Scheduling

Load Balancing Tradeoffs

Goal of **Load Balancing** is to schedule processors to do similar amounts of work $\Rightarrow$ Finish a set of tasks as quickly as possible. We must consider the following important tradeoffs:

- To finish tasks faster, load balancing itself cannot introduce too much overhead
- Static v.s. Semi-static v.s. Dynamic
    - To statically balance the load we need *Load Estimation* by:
        1. Infer from the code / data (best case)
        2. Profile the task, and assume its future behavior matches the past
        3. Build a mathematical model
    - To dynamically balance the load: when load changes, we need *Rebalance*
- Load balancing in different granularities
- Centralized v.s. Distributed v.s. Hierarchical

Static Load Balancing

Denote time for the last task to finish as *makespan*. Consider the simplest case: tasks sizes are known, no precedence constraints between tasks, and ignore communication costs. Even under this situation, static load balancing is *NP-Complete* (can reduce to SUBSET-SUM problem).

List Scheduling

List Schedulig (*Graham*, *LS*) is the simlest greedy scheduling algorithm. List tasks in any order, then whenever a processor is idle, give it the next task in list to execute.

Worst-case: $M(LS) \leq 2M_{opt}$, so we call it a "2-approximation".

Longest Processing Time Scheduling

Longest Processing Time Scheduling (*LPT*) is similar to LS scheduling, except that it first sorts the task list in non-increasing order.

Worst-case: $M(LPT) \leq \frac{4}{3}M_{opt}$, so we call it a "4/3-approximation".

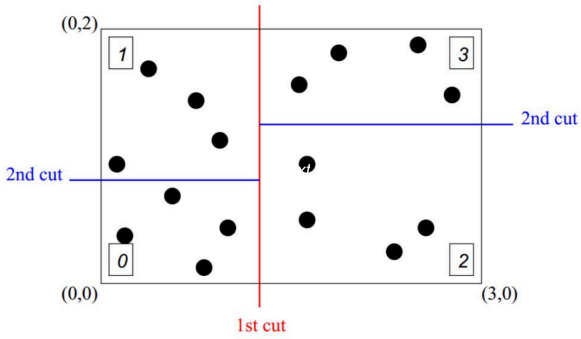
> Proof of bound omitted here. Refer to Slide 17, Page 13-14.

Geometric Load Balancing

In parallel applications working on data over geometric coordinates, nearby tasks communicate with each other, so we wanna load balance & put nearby tasks together on the same processor.

Recursive Bisection

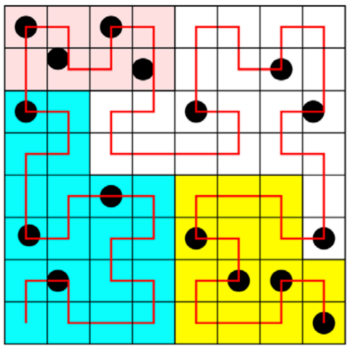
Ξ Procedure:



1. First partite tasks evenly along x-axis
2. Then partite each group evenly along y-axis
3. Repeat until desired number of groups

Space Filling Curve

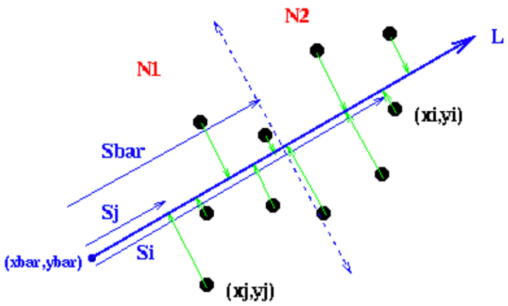
[1] Procedure:



1. Draw a *Space Filling Curve* (SFC) across the data space, which maps all tasks onto a 1D line
2. Partite nodes along the line evenly

Inertial Partitioning

[1] Procedure:



1. Find a rotation axis L, so that when all nodes rotate around L, the overall moment of intertia is minimal
2. Project all tasks onto L
3. Find the median of all projection points on L, draw a perpendicular line, partite tasks according to that line
4. Repeat for every partition until desired number of groups

Graph-based Partitioning

Given a explicit graph describing the task relationships (might be unweighted / weighted), we wanna partite the graph into two halves, such that each part has approximiately same number of tasks, and also minimizing the cut edge weights. Optimal cut is *NP-Complete*, so we use proper heuristics.

This is mostly a graph algorithmic problem, so we just do a quick summary here:

- Local search algorithms, e.g. **Kernighan-Lin** partitioning
- **Spectral** partitioning: get *Laplacian Matrix* $L(G)$ of the graph G, then partition according to the second smallest eigenvector
- **Multilevel** partitioning: make a coarse-grained graph, and do load balancing on the coarse graph to improve partitioning speed

Dynamic Load Balancing

Dynamic load balancing mostly takes the form of **Work Stealing**. Each processor maintains a *Double-ended Queue* (Deque) of tasks. An idle processor fetches the task on top to execute. Any new task is inserted into the bottom of the deque. When its deque is empty, it steals one from the bottom of another processor's deque.

Incurs a lot of overhead when one processor is frequently producing new tasks, and other processors have nothing to do except for stealing.

Parallel Matrix Algorithms

Sparse Matrices

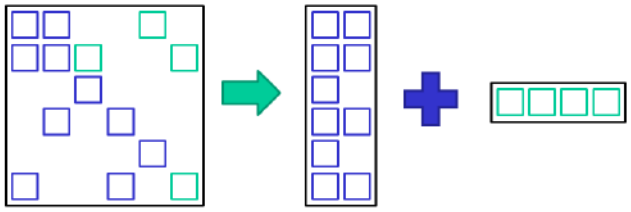
Sparse Matrix is where most of the elements are zero. They have many kinds of optimized storage formats.

Sparse Matrix Storage Formats

Sparse matrices can be stored in compact with various formats:

1. *DIA* format: $\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$ $\texttt{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$ $\texttt{offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix}$
   - Effective when non-zeros lie on a few diagonals
2. *ELL* format: $\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$ $\texttt{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$ $\texttt{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$
   - Also called *ELL Row Packing*
   - Effective when roughly the same number of elements per row
3. *COO* format: $\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$ $\begin{aligned} \texttt{row} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$
   - Most general purpose, no restriction on matrix shape
   - Memory inefficient, because stores the same row index repeatedly
4. *CSR* format: $\texttt{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$ $\begin{aligned} \texttt{ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \texttt{indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \texttt{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$
   - Compacts the same repeated row index

- Flexible and efficient, widely used

5. *Hybrid* format: 
   - Combines ELL & COO

SpMV CUDA Kernels

Sparse Matrix-Vector Multipliation (*SpMV*) is the multiplication of a sparse matrix and a colomn vector. We will need efficient representations for sparse matrices, then design corresponding algorithms.

1. ELL kernel

```
data    [1 2 5 6 7 8 3 4 * * 9 *]
indices [0 1 0 1 1 2 2 3 * * 3 *]
```

   - With above example:

```
Iteration 0 [0 1 2 3                 ]
Iteration 1 [         0 1 2 3        ]
Iteration 2 [                 0 1 2 3]
```

2. CSR scalar kernel:

```
indices [0 1 1 2 0 2 3 1 3]
   data [1 7 2 8 5 3 9 6 4]
```

   - With above example:

```
Iteration 0 [0     1     2       3   ]
Iteration 1 [  0     1     2       3]
Iteration 2 [                 2     ]
```

3. CSR vector kernel:

```
indices [0 1 1 2 0 2 3 1 3]
   data [1 7 2 8 5 3 9 6 4]
```

   - With above example:

```
Warp 0 [0 0                   ]
Warp 1 [    1 1               ]
Warp 2 [        2 2 2         ]
Warp 3 [              3 3     ]
```

4. COO kernel:

```
   row [0 0 1 1 2 2 2 3 3]
indices [0 1 1 2 0 2 3 1 3]
   data [1 7 2 8 5 3 9 6 4]
```

   - With above example:

```
Iteration 0 [0 1 2 3           ]
Iteration 1 [       0 1 2 3    ]
Iteration 2 [               0  ]
```

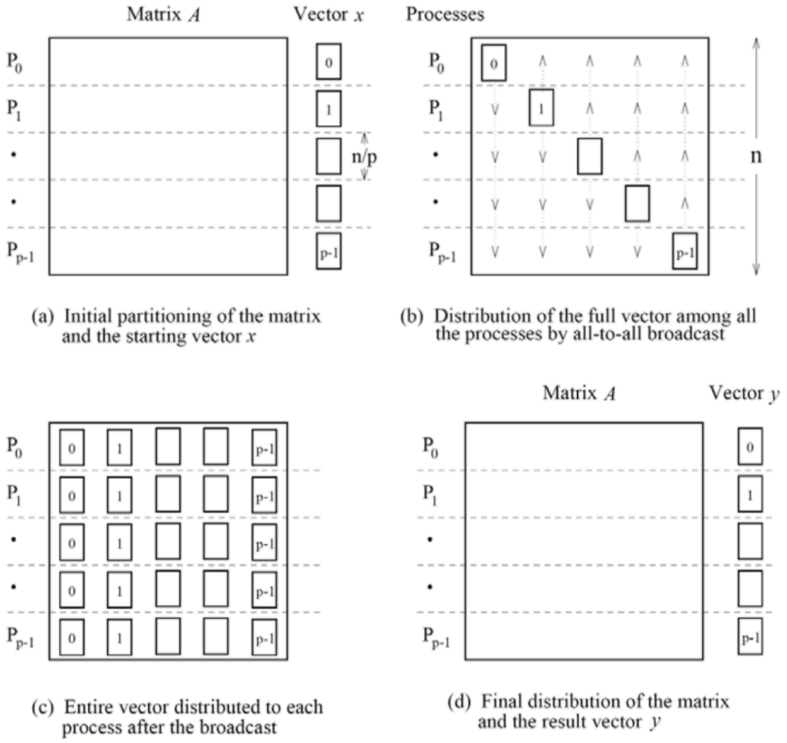
Dense Matrices

Dense Matrix are used in many scientific computing applications, and they have great potential to be parallelized.

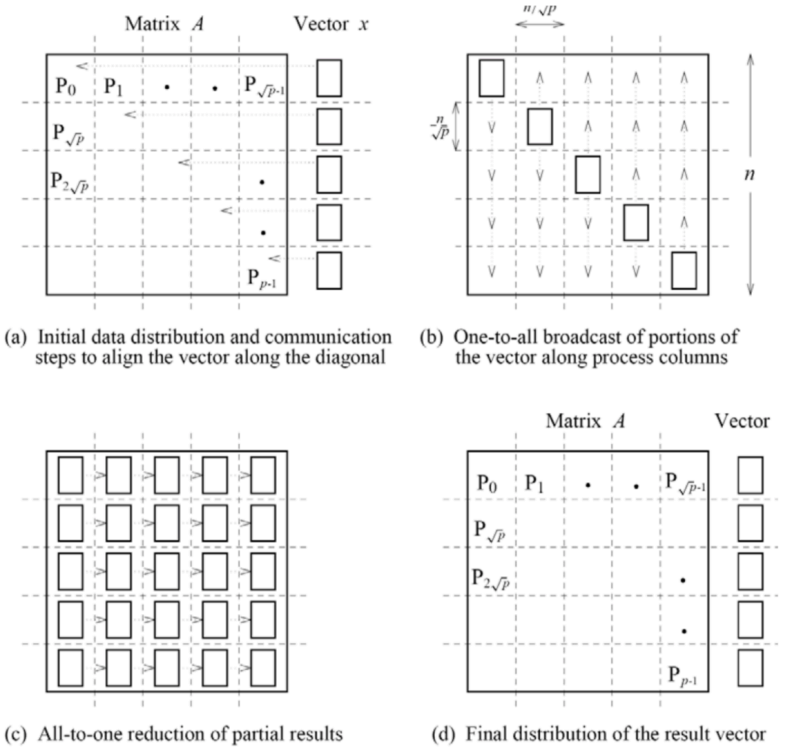
DeMV Algorithms

Dense Matrix-Vector Multiplication (*DeMV*) has its own regular patterns.

1. 1D processor partition:
    - ⊟ Procedure:



    - ⇝ Total work = $p(T_{comm} + T_{comp}) = p(t_s \log p + t_w n + \frac{n^2}{p}) = O(p \log p + np + n^2)$; Isoefficiency achieved when $p = O(n)$
2. 2D processor partition:
    - ⊟ Procedure:

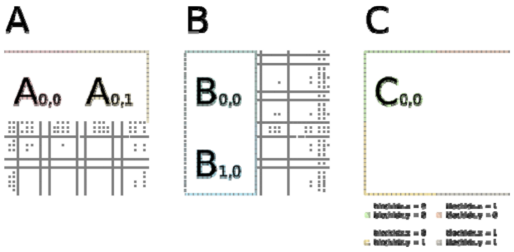


- ⇝ Total work = $p(T_{comm} + T_{comp}) = p((t_s + t_w \frac{n}{\sqrt{p}}) \log \sqrt{p} + \frac{n^2}{p}) = O(p \log p + n\sqrt{p} \log p + n^2)$; Isoefficiency achieved when $p = O(\frac{n^2}{\log^2 p})$, more *scalable* than 1D

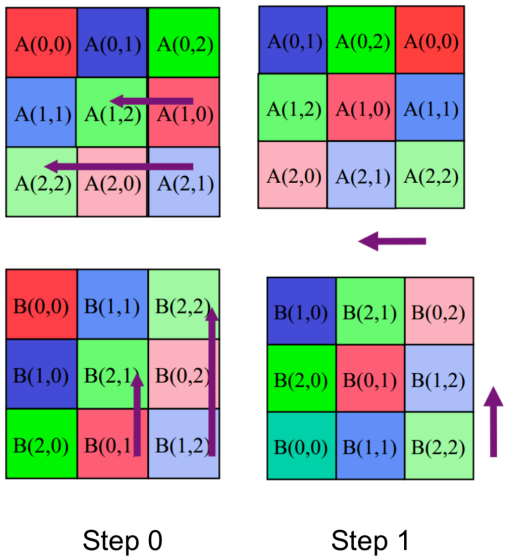
Parallel Matrix-Matrix Multiplication

There are also a lot of ways to optimize multiplication of dense matrices:

1. Block-by-Block partition:
   - ⇝ Procedure:



     - Every processor $p_{i,j}$ stores the corresponding block from A and B initially
     - Every processor $p_{i,j}$ broadcasts its $A_{i,j}$ block to Row-i, and broadcasts its $B_{i,j}$ block to Col-j (In total $2\sqrt{p}$ all-to-all broadcasts)
     - Then all processors have what they need, they compute result block $C_{i,j}$
   - ⇝ Total work = $p(T_{comm} + T_{comp}) = p(\log \sqrt{p} + \frac{n^2}{p}\sqrt{p} + (\frac{n}{\sqrt{p}})^3 \sqrt{p}) = O(p \log \sqrt{p} + n^2\sqrt{p} + n^3)$; Isoefficiency achieved when $p = O(n^2)$
   - ⇝ Total memory usage is high: $p \cdot 3\frac{n^2}{\sqrt{p}} = O(n^2\sqrt{p})$
2. *Cannon's* MM algorithm:
   - ⇝ Procedure:

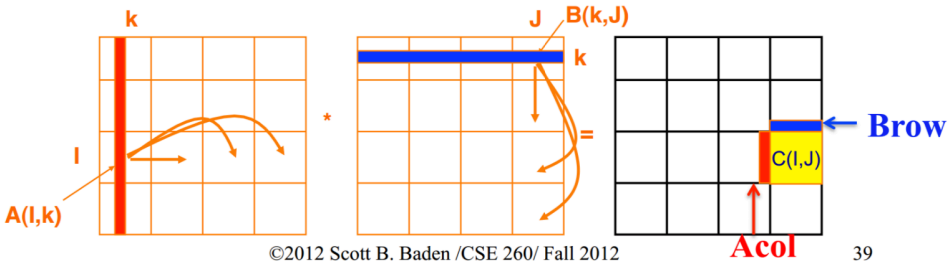


     - Every processor $p_{i,j}$ stores the corresponding block from A and B initially
     - Step 0:
       - Rotate blocks in the i-th row of A left by i; Rotate blocks in the i-th column of B up by i, as shown in "step 0" above
       - Then each processor multiplies the A block it gets with the B block it gets. This will

produce one component of the corresponding $C_{i,j}$ block
    - For all the rest $\sqrt{p}-1$ steps:
        - Shift matrix A left by 1; Shift matrix B up by 1, as shown in "step 1" above
        - Do the multiplication to produce another component of $C_{i,j}$ block
    - Add the $\sqrt{p}$ components together to get the true $C_{i,j}$ result
- $\varkappa$ Total work = $p(T_{comm}+T_{comp}) = p(p\sqrt{p}+n^2\sqrt{p}+\frac{n^3}{p}) = O(p\sqrt{p}+n^2\sqrt{p}+n^3)$; Isoefficiency achieved when $p = O(n^2)$
- $\varkappa$ Total memory usage at any time = $O(n^2)$

> Drawback: Can only deal with square matrices, and n must be dividable by $\sqrt{p}$.

3. *SUMMA* algorithm:
    - Ξ Procedure:

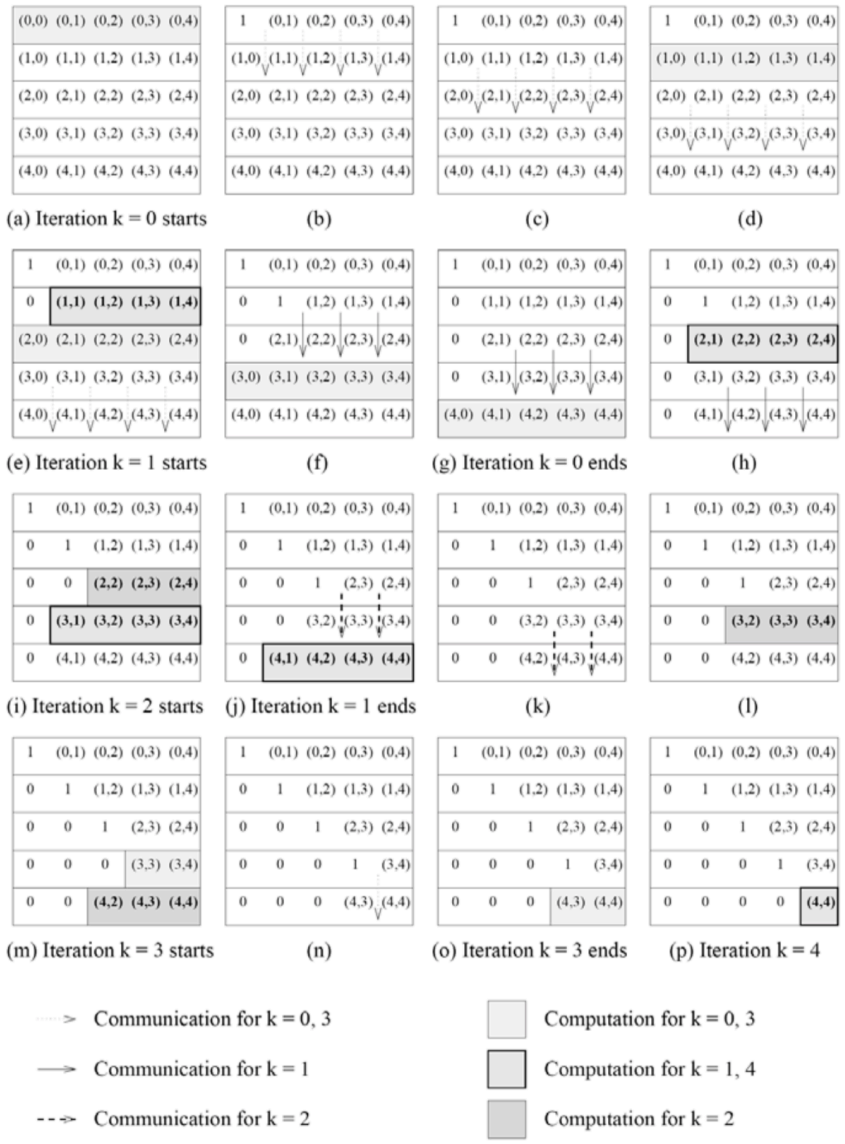


        - Uses the outer product way of MM multiplication (kij-order iteration)
        - In the k-th iteration:
            - Every processor holding the k-th column of A broadcasts its segment of that column to its row; Every processor holding the k-th row of B broadcasts its segment of that row to its column
            - Then each processor calculates the outer prodect of the two segments it received, and accumulates it into $C_{i,j}$
    - $\varkappa$ Total work larger than Cannon's, because of n broadcasting
4. 3D processor array multiplication:
    - Can take use of n^3 processors for multiplication, and can theoretically achieve $O(\log n)$ time

Parallel Gaussian Elimination

Gaussian Elimination is used for solving linear systems or get the LU decomposition of a matrix. Serially takes $O(n^3)$ time.

1. Naive parallelization (n processors):
    - Ξ Procedure: In the k-th iteration, processor k divides the k-th row by $A_{k,k}$, then broadcasts this row to processors below it to do the elimination in parallel
    - $\varkappa$ Total work = $n \cdot O(n^2 \log n) = O(n^3 \log n)$, NOT cost optimal
2. Pipelined parallelization (n processors):
    - Ξ Procedure:



    - When a row has done elimination, it sends the values down to the next row
    - Once a row receives values from a previous row, it first passes them down to the next row, before doing eliminations on itself using these values
  - Total work = $n \cdot O(n)$ cycles $\cdot\, O(n)$ operations per cycle $= O(n^3)$, is cost optimal
3. Load balancing for $< n$ processors:
  - Simply assigning consecutive rows to a processor will cause unbalanced workload, becaues upper processors finish earlier than later processors in the pipeline
  - Can assign rows in a *Round-Robin* (cyclic) way

Iterative Matrix Algorithms

For general large matrices, direct methods like Gaussian Elimination are too slow. To get an approximate solution, we apply iterative algorithms that eventually converge to the true solution.

To solve the system $Ax = b$, write $A = M - N$, where M^{-1} is easy to compute. Then the solution x^* follows $x^* = Cx^* + d$, where $C = M^{-1}N$ and $d = M^{-1}b$. Starting from an arbitrary initial guess $x^{(0)}$, we repeatedly compute $x^{(k+1)} = Cx^{(k)} + d$ until it is close enough to x^*. Convergence criteria: $\lim_{k\to\infty} C^k = 0$, i.e. $|\text{Largest eigenvalue of } C| < 1$.

Jacobi Method

Write $A = M - N$, where:

- $M = D$, the main diagonal of A; Invert is simply inverting each element
- $N = L + R$, where:
  - L is the negative of the lower triangular part of A without D
  - R is the negative of the upper triangular part of A without D

Thus we have:

- $C = D^{-1}(L + R) \Rightarrow c_{i,j} = -a_{i,j}/a_{i,i}$ for $i \neq j$, and $c_{i,i} = 0$
- $d = D^{-1}b \Rightarrow d_i = b_i/a_{i,i}$

We can see that $x_i^{(k+1)} = \frac{1}{a_{i,i}}(b_i - \Sigma_{j=1,j\neq i}^{n} a_{i,j}x_j^{(k)})$. Every element of $x^{(k+1)}$ only depends on $x^{(k)}$, thus can be calculated in parallel. Suppose matrix A and vector b are distributed across n processors in row-wise order, after each iteration k, all processors use `MPI_Allgather` to get all values of the old x.

Gauss-Seidel Method

Write $A = M - N$, where:

- $M = D - L$; Invert can be done by forward substitution
- $N = R$

Similarly we have $x_i^{(k+1)} = \frac{1}{a_{i,i}}(b_i - \Sigma_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \Sigma_{j=i+1}^{n} a_{i,j}x_j^{(k)})$. Here $x_i^{(k+1)}$ depends not only on $x^{(k)}$ but also on all $x_j^{(k+1)}$ for $j < i$.

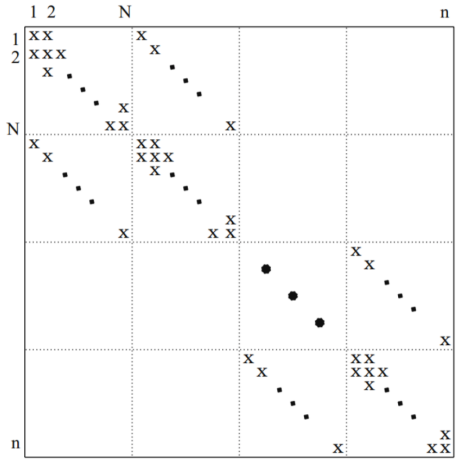
> Gauss-Seidel converges faster than Jacobi, but has less parallelism.
>
> To obtain even faster convergence, Successive Over-Relaxation (*SOR*) method introduces ω parameter: $x_i^{(k+1)} = \frac{\omega}{a_{i,i}}(b_i - \Sigma_{j=1}^{i-1} a_{i,j}x_j^{(k+1)} - \Sigma_{j=i+1}^{n} a_{i,j}x_j^{(k)}) + (1 - \omega)x_i^{(k)}$. Convergence speed depends on how we choose ω.

To parallel Gauss-Seidel, for every element $x_i^{(k+1)}$, split its computation onto p processors, and then call an `MPI_Allreduce` to sum up.

2-D Poisson's Equation

Poisson's Equation describes a partial different equation which describes a potential field. We wish to find a function $\phi(x, y)$ such that $-\Delta\phi = -(\frac{\partial^2\phi}{\partial^2 x} + \frac{\partial^2\phi}{\partial^2 y}) = f$. By discretizing the 2D space into $(N + 2) \times (N + 2)$ points, we have: $4\phi_{i,j} - \phi_{i+1,j} - \phi_{i-1,j} - \phi_{i,j+1} - \phi_{i,j-1} = (\frac{1}{N+1})^2 f_{i,j}$. This leads to N^2 linear equations, each for one interior point $1 \leq i, j \leq N$.

Write vector x as $x_k = \phi_{i,j}$, and vector b as $b_k = (\frac{1}{N+1})^2 f_{i,j}$, for $k = i + (j - 1)N$. The equation can be transformed into: $4x_k - x_{k+1} - x_{k-1} - x_{k+N} - x_{k-N} = b_k$. Construct matrix A as follows:

$A =$ 

$(n \times n, n = N^2)$, where:

- The main diagonal is all 4
- Other four diagonals are all -1

The problem is now to solve $Ax = b$. We can then use iterative Gauss-Seidel method to get a numerical solution. Here $x_i^{(k+1)}$ only depends on $x_{i-1}^{(k+1)}$ and $x_{i-N}^{(k+1)}$, so if we put all elements of x onto a $N \times N$ grid, each point only depends on the left point and top point. Thus we can extract more parallelism from Gauss-Seidel, by iterating through all anti-diagonals of the grid from up-left to bottom-right, and within each diagonal, compute all elements in parallel.

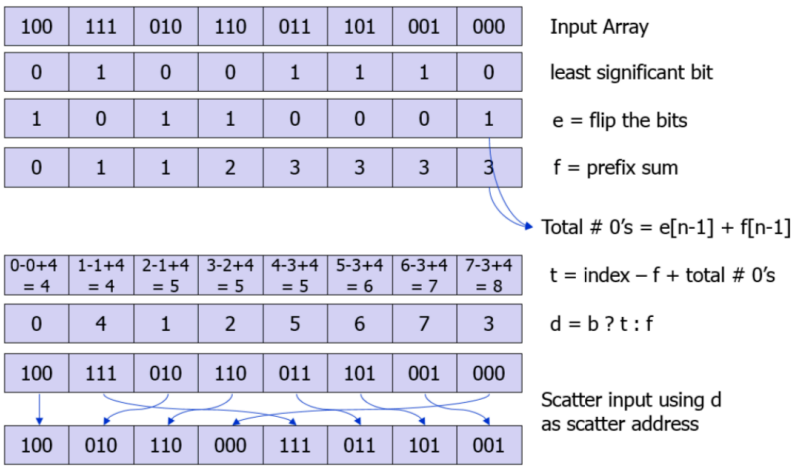
> To extract even more parallelism, we can reorder the mapping from $\phi_{i,j} \to x_k$ using *Red-Black Ordering*. Then we can calculate all red points in parallel, then calculate all black points in parallel.

Parallel Sorting Algorithms

Parallel Radix Sort

In **Radix Sort** (only effective for numerical values), we apply *Stable* sorting from least-significant digits to most-significant digits. Suppose we write all numbers in binary form, then in each phase we are just sorting 0s and 1s.

Ξ Procedure: In each phase, do：



$\varkappa$ Total time = $O(m \log n)$, where m is number of bits per value.

> Radix sort is one of the most efficient and widely-used sorting algorithm in practical use.

Parallel Merge Sort

Merge Sort takes $O(\log n)$ stages. We can do each merge stage in parallel. For simplicity, suppose all values are unique. Denote: $rank(x, S) =$ number of values in S that $\leq x$, it is obvious that when merging A with B, $rank(x, A|B) = rank(x, A) + rank(x, B)$. For exmaple for $x \in A$, RHS can be easily computed by:

- $rank(x, A)$ is just the index of x in A
- $rank(x, B)$ can be computed by a *Binary Search* in $O(\log n)$ time

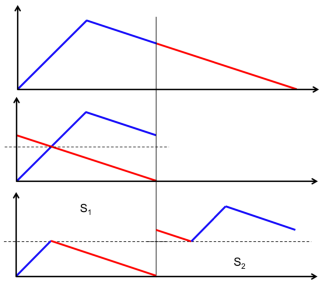
Thus, using n processors, every merge state can be done in $O(\log n)$ time.

$\varkappa$ Total time = $O(\log^2 n)$; Total work = $O(n \log^2 n)$, NOT work-efficient.

Bitonic Sort

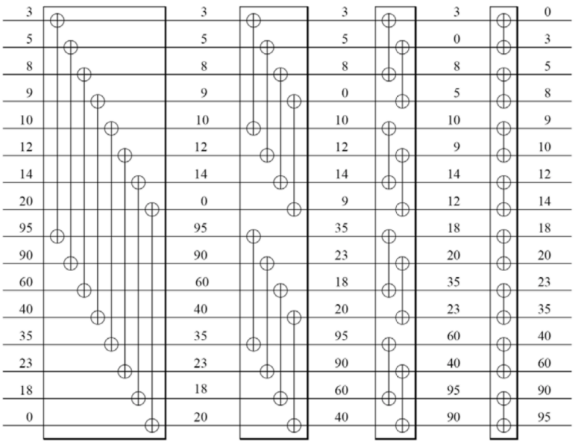
A *Bitonic Sequence* is a sequence which:

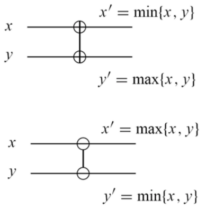
- Definition:
  - First increases $\uparrow$, then decreases $\downarrow$;
  - Or any *Rotation* of such sequence
- Property:



  - Split a Bitonic Sequence S into 2 halves, overlap these 2 halves
  - S_1 gets all the smaller values; S_2 gets all the bigger values
  - Then both S_1 and S_2 are Bitonic Sequences, and everything in $S_1 \leq$ everything in S_2

Taking advantage of such property, a **Bitonic Merge** operation sorts a Bitonic Sequence of length n by recursively applying $\log n$ phases. Denote *Increasing* Bitonic Merge as $\oplus$, and *Decreasing* Bitonic Merge as $\ominus$. A Bitonic Merge can be implemented as a network of comparators:

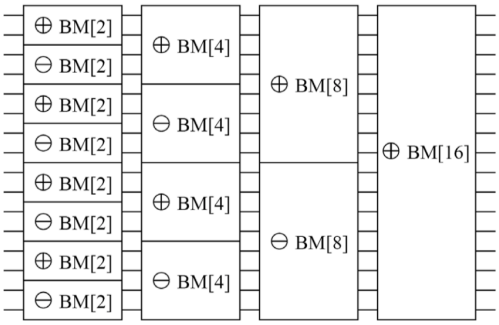
For example $\oplus BM[16] =$ 

, where 

> All comparators in a phase can work simultaneously, so $BM[n]$ takes $\log n$ time.

Any sequence of length 1 is naturally Bitonic. Thus, for arbitrary input sequence, a **Bitonic Sort** (双调排序) conducts Bitonic Merges starting from scale 2 (with interleaved directions):

Ξ Procedure:



$\varkappa$ Total time = $O(\log n)$ phases of merges $\times$ $O(\log n)$ time per merge phase $= O(\log^2 n)$; Total work = $O(n \log^2 n)$, NOT work-efficient.

Sample Sort

Sample Sort is a randomized sorting algorithm. The main idea is to distribute n values onto p processors, such that everything on processor k is smaller than processor $k+1$. Then every processor just do a local sorting and a global order is inherently achieved.

1. Naive implementation:
    - Ξ Procedure:
        1. Pick $p-1$ pivots randomly, arrange them as $t_1 < t_2 < \cdots < t_{p-1}$
        2. Loop through all values, distribute value v to processor k iff $t_k \le v < t_{k+1}$
        3. Then every processor just do a local sorting
    - $\varkappa$ Total time = $O(s \log s)$, where s is the maximum bucket size, depends on the balance of pivots; On average $s = \Theta(\frac{n \log n}{p})$, so total time is not very optimal
2. *Sampling* implementation:
    - Ξ Procedure:
        1. Every processor starts with $\frac{n}{p}$ values
        2. Each processor picks λ values randomly, send them to processor 0; In total λp values are picked, arrange them in increasing order
        3. Processor 0 picks every λ-th value from above as a pivot; This produces p pivots
        4. Processor 0 broadcasts the pivot set to all other processors
        5. Each processor loops through its local values, send them to the corresponding processor
        6. Then every processor just do a local sorting
    - Sampling by a proper scale λ can significantly improve workload balance; Proof omitted

Parallel Fast Fourier Transform

Fast Fourier Transform (FFT)

The n-th **Root of Unity** is a complex number $\omega_n = e^{\frac{2\pi i}{n}}$. It has the following properties:

- $\omega_n^n = 1$
- $\omega_n^{k+\frac{n}{2}} = -\omega_n^k$
- $(\omega_n^k)^2 = \omega_n^{2k} = \omega_{\frac{n}{2}}^k$

Given a polynomial $A = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$, a **Discrete Fourier Transform** (*DFT*) computes $A(\omega_n^0), A(\omega_n^1), \ldots, A(\omega_n^{n-1})$. Normally, each value can be computed in $O(n)$ time, thus we need $O(n^2)$ time to do a Discrete Fourier Transform. A **Fast Fourier Transform** (*FFT*) does this in $O(n \log n)$ time by *divide-and-conquer*.

Denote:

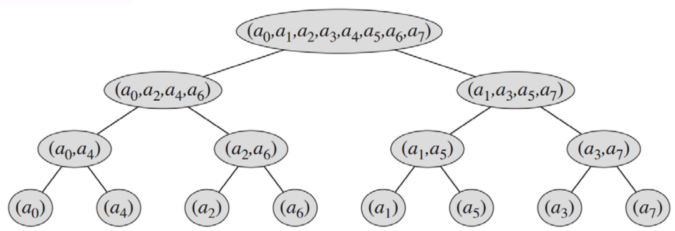
- $A^{[0]} = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}$
- $A^{[1]} = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$

Then $A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2)$. According to the property of roots of unity, we can get all n values of A by: for $0 \le k \le \frac{n}{2} - 1$

- $A(\omega_n^k) = A^{[0]}(\omega_{\frac{n}{2}}^k) + \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k)$
- $A(\omega_n^{k+\frac{n}{2}}) = A^{[0]}(\omega_{\frac{n}{2}}^{k+\frac{n}{2}}) + \omega_n^{k+\frac{n}{2}} A^{[1]}(\omega_{\frac{n}{2}}^{k+\frac{n}{2}}) = A^{[0]}(\omega_{\frac{n}{2}}^k) - \omega_n^k A^{[1]}(\omega_{\frac{n}{2}}^k)$

This means once we get the $\frac{n}{2}$ values of $A^{[0]}(\omega_{\frac{n}{2}}^k)$ and the $\frac{n}{2}$ values of $A^{[1]}(\omega_{\frac{n}{2}}^k)$, for $0 \le k \le \frac{n}{2} - 1$, we can obtain all n values of A in linear time. Notice that the procedure of computing $A^{[0]}$ / $A^{[1]}$ are just FFTs of $\frac{n}{2}$ problem size! Here the problem fits in the classic divide-and-conquer model. Recursion relation gives $T(n) = 2T(\frac{n}{2}) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$.

Recursion tree looks like:

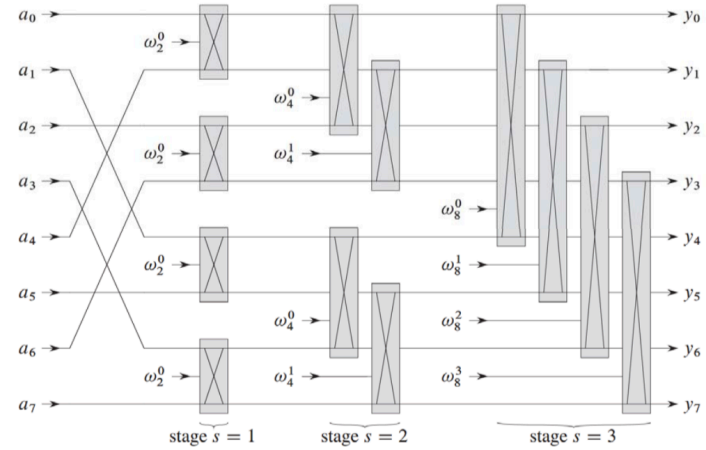


> Two aspects of understanding the physical meaning of FFT:
>
> 1. DFT transforms a discrete finite signal in *time domain* $[a_0, a_1, \ldots, a_{n-1}]$ into its corresponding *frequency domain* components $[A_0, A_1, \ldots, A_{n-1}]$, where each $A_k = \Sigma_{j=0}^{n-1} a_j e^{\frac{2\pi i k}{n}} = \Sigma_{j=0}^{n-1} a_j \omega_n^k$ according to Fourier Transform. FFT accelerates this computation to $O(n \log n)$ time.
> 2. When multiplying two n-th order polynomials A and B to get the result polynomial C:
>   - If they are represented as coefficients $[a_0, a_1, \ldots, a_{n-1}]$ and $[b_0, b_1, \ldots, b_{n-1}]$, then each result coefficient $c_k = \Sigma_{j=0}^{k} a_j b_{k-j}$. The overall computation requires $O(n^2)$ time.
>   - An n-th order polynomial can also be represented as $n+1$ point-value-pairs $[(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_n, A(x_n))]$ and $[(x_0, B(x_0)), (x_1, B(x_1)), \ldots, (x_n, B(x_n))]$. Under this representation, the $n+1$ C values can be computed in $O(n)$ time. Thus, if we can efficiently convert *coefficient-representation* to *point-value-pairs-representation* (and vice versa), then the polynomial multiplication will be much faster. FFT & Invert-FFT does such conversion in $O(n \log n)$ time, where x_k is chosen as ω_n^k. This enables polynomial multiplicaiton in $O(n \log n)$ time.

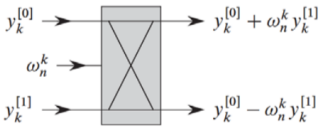
FFT Circuits

FFT can be parallelized in a circuit manner, similar to bitonic sorting.

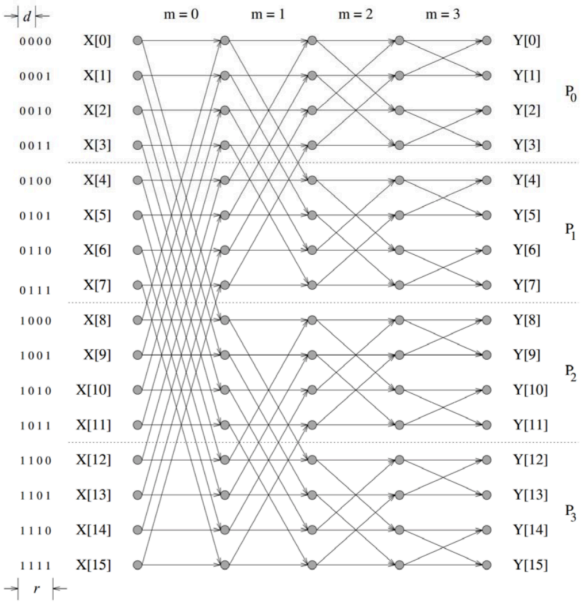
1. Naive implementation:
    - Ξ Procedure:



, where



    - ⇝ In total $\log n$ stages, BUT requries n processors
2. *Binary Exchange* FFT (Bufferfly-FFT):
    - Ξ Procedure:



    - ⇝ In total $\log n$ stages; We can have arbitrary number of processors, and communication only happens in first $\log p$ stages (afterwards all value exchanges happen inside the processor locally)

2-D Transpose FFT

Binary exchange FFT has poor work efficiency when $\frac{t_w}{t_{comp}}$ is large. A 2-D transpose FFT always maintains isoefficiency for $W = O(p^2 \log p)$, regardless of machine parameters. Suppose $\sqrt{n}$ is a power of 2, arrange all n inputs on a grid, and distribute a column of values to a processor in a *Round-Robin* order:

Initial value distribution: 

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

, we see that all values that a processor needs in the first $\frac{\log n}{2}$ phases are already in its hand. After the first half of computation, transpose the distribution (can be done in $O(\frac{n}{p})$ time on hypercube), and the rest $\frac{\log n}{2}$ phases still require no communication.

Parallel Searching Algorithms

Discrete Optimization Problems (DOP)

A **Discrete Optimization Problem** (*DOP*) consists of a set of solutions S and a cost function f. The *Minimization* problem tries to find $x^{\backslash *} \in S$ s.t. $f(x^{\backslash *})$ is the smallest.

Searching a DOP can be modeled by a search tree / graph. If it is not a tree, we need:

- Duplicate detection, e.g. using a hash table to store explored nodes
- Cost update, e.g. update the cost of a node when reaching it the second time and found that current cost is smaller than original

Branch & Bound method (剪枝搜索) can be used in search graph traversal. for every node p on the frontier, we maintain a pair $(m(p), M(p))$, where $m(p) \leq$ the minimum possible value of any descendant of p, and $M(p) \geq$ the maximum possible value of any descendant of p.

- For example in knapsack problems, let m be value of current knapsack at p, and M be if all remaining items are put into the knapsack
- If at some timepoint, $m(p) \geq M(q)$, then all descendants of p do not need to be explored any more

Parallel Graph Traversal

Static assignments on graph traversals can cause significant load imbalance, because different branches may have different depth / number of descendants. We can overcome this by *Work Stealing*. Several things need to be controlled:

- Choose who to steal from:
  - Local (asynchronous) Round-Robin
  - Global Round-Robin
  - Randonmized work stealing
- Control how much work to steal:
  - Set a *cutoff* depth, and don't steal from a processor who has pasted the cutoff depth

For DFBB, A*, IDA*, ..., they all have the potential of parallelization, but contains a lot of detailed techniques and tradeoffs to make them efficient. (Overall, searching algorithm is one of the hardest to parallelize effectively.)

> Parallel graph traversals can introduce *superlinear* speedup. This phenomenon is called **Speedup Anomaly**. Reason is that when there are multiple processors, we might explore the graph in a different order of sequential algorithm. Then, if the solution is at a relatively shallow branch, when exploring multiple branches simultaneously we might meet the solution much earlier than the

> sequential algorithm. Since the search stops whenever a solution is found, this results in total work $W_p < W_s$.

PRAM Model

Concepts

Parallel Random Access Machine (*PRAM*) model is an old theoretical model that generalizes a parallel machine. The machine has:

- Multiple computing processors (CPUs)
- An infinitely large shared memory
- All processors execute in synchronized steps, in each step, it
    - Reads a memory location
    - Compute on the data
    - Then writes a memory location

Considering memory conflicts, there are different kinds of PRAM machines:

- Exclusive Read Exclusive Write (*EREW*)
- Concurrent Read Exclusive Write (*CREW*)
- Exclusive Read Concorrent Write (*ERCW*)
- Concurrent Read Concurrent Write (*CRCW*), writes to the same location will be reduces by certain rule, e.g. max

Number of steps till a PRAM algorithm terminates is called **Depth**. PRAM algorithms often takes polylogarithmic depth ($O(\log^k n)$) using $O(n^k)$ processors. Minimizing work of PRAM algorithms (making it work-efficient) are more important than minimizing depth.

Algorithms

The following classic PRAM algorithms are discussed in the slides:

1. Basic algorithms:
    1. Parallel carry lookahead addition
    2. Superfast max finding
    3. Parallel list ranking
    4. Prefix sum on linked list
2. Graph algorithms:
    1. Coloring a cycle
    2. Independent set on a line
    3. Connected components finding
3. Tree algorithms:
    1. Euler tour of a tree
    2. Rooting a tree
    3. Node depths
    4. Post-order numbering
    5. Number of descendants
    6. Evaluating binary expression tree
    7. Lowest common ancestors

8. Range minima

MapReduce Model

Concepts

MapReduce is a programming model for batch processing on distributed cluster of commodity servers. Designed to be run dynamically allocated cloud servers. Widely-used in academia and industry due to low cost and ease of use. Problems that perform simple operations sequentially on a large amount of data can fit in MapReduce model.

MapReduce runs on *Key-Value Pairs*. For every Map+Reduce stage:

- *Map*: $(k, v) \to [(k', v')]$, mappers map a pair to 1 / more new pairs, then send them to reducers
- *Reduce*: $(k, [v]) \to [(k', v')]$, collects all values whose key is k, and reduce them into results

One MapReduce job can contain multiple stages. All input / intermediate result / output are all stored as files in a distribuetd persistent storage.

> **Spark**: An in-memory & fault-tolerant optimized solution for iterative algorithms, based on RDD. It overcomes the drawback that for iterative algorithms, MapReduce stores & loads intermidiate results to / from persistent storage.

Algorithms

The following classic MapReduce algorithms are discussed in the slides:

1. Filtering
2. Select top k
3. Distinct values
4. Inner joins
5. Word co-occurrence
6. Breadth first search
7. Single source shortest paths
8. *PageRank*