Parallel Numerical Algorithms

Chapter 1 – Parallel Computing

Michael T. Heath and Edgar Solomonik

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

1. Motivation
2. Architectures
   - Taxonomy
   - Memory Organization
3. Networks
   - Network Topologies
   - Graph Embedding
   - Topology-Awareness in Algorithms
4. Communication
   - Message Routing
   - Communication Concurrency
   - Collective Communication

Limits on Processor Speed

- Computation speed is limited by physical laws

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by
  - heat dissipation

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by
  - heat dissipation
  - thermal noise (degradation of signal-to-noise ratio)

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by
  - heat dissipation
  - thermal noise (degradation of signal-to-noise ratio)
  - quantum uncertainty at small scales

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by
  - heat dissipation
  - thermal noise (degradation of signal-to-noise ratio)
  - quantum uncertainty at small scales
  - granularity of matter at atomic scale

Limits on Processor Speed

- Computation speed is limited by physical laws
- Speed of conventional processors is limited by
  - line delays: signal transmission time between gates
  - gate delays: settling time before state can be reliably read
- Both can be improved by reducing device size, but this is in turn ultimately limited by
  - heat dissipation
  - thermal noise (degradation of signal-to-noise ratio)
  - quantum uncertainty at small scales
  - granularity of matter at atomic scale
- Heat dissipation is current binding constraint on processor speed

Moore's Law

- Loosely: complexity (or capability) of microprocessors doubles every two years

Moore's Law

- Loosely: complexity (or capability) of microprocessors doubles every two years
- More precisely: number of transistors that can be fit into given area of silicon doubles every two years

Moore’s Law

- Loosely: complexity (or capability) of microprocessors doubles every two years
- More precisely: number of transistors that can be fit into given area of silicon doubles every two years
- More precisely still: number of transistors per chip that yields minimum cost per transistor increases by factor of two every two years

Moore's Law

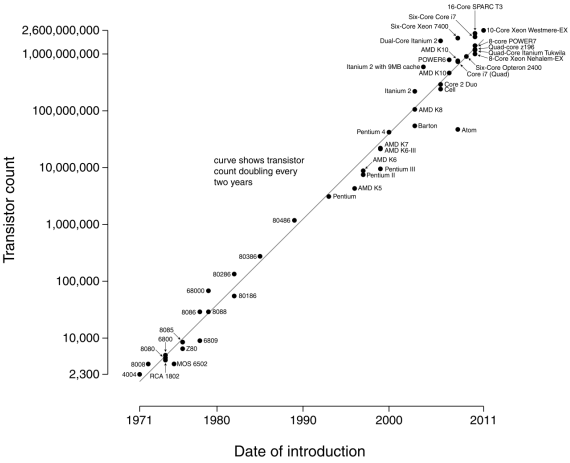
- Loosely: complexity (or capability) of microprocessors doubles every two years
- More precisely: number of transistors that can be fit into given area of silicon doubles every two years
- More precisely still: number of transistors per chip that yields minimum cost per transistor increases by factor of two every two years
- Does **not** say that microprocessor performance or clock speed doubles every two years

Moore's Law

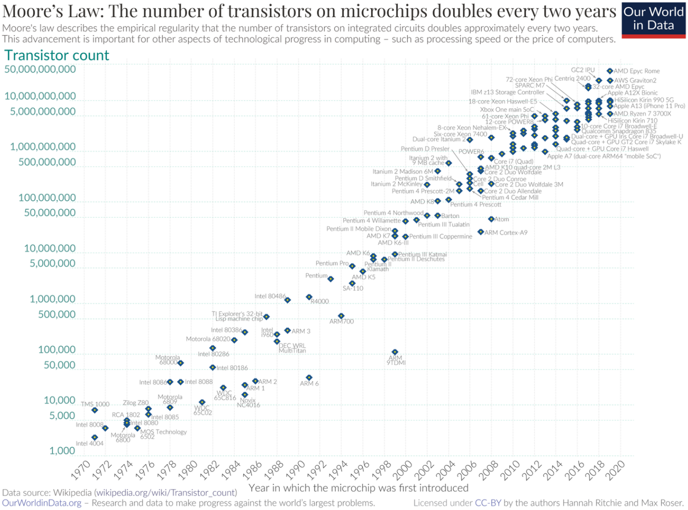
- Loosely: complexity (or capability) of microprocessors doubles every two years
- More precisely: number of transistors that can be fit into given area of silicon doubles every two years
- More precisely still: number of transistors per chip that yields minimum cost per transistor increases by factor of two every two years
- Does **not** say that microprocessor performance or clock speed doubles every two years
- Nevertheless, clock speed did in fact double every two years from roughly 1975 to 2005, but has now flattened at about 3 GHz due to limitations on power (heat) dissipation

Moore's Law

Microprocessor Transistor Counts 1971-2011 & Moore's Law

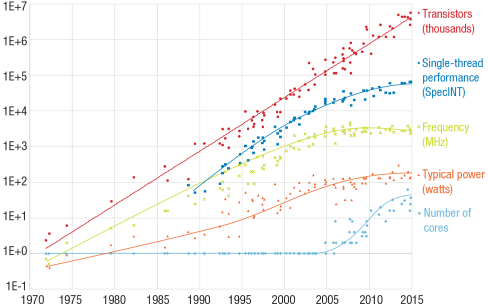


Moore’s Law, Architectures through 2020



The End of Dennard Scaling

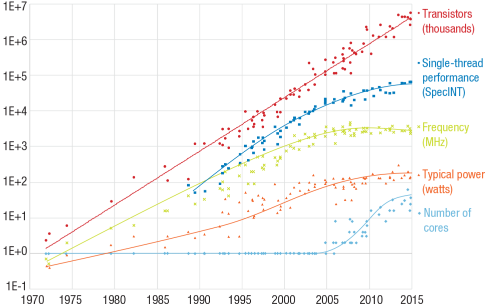
Dennard scaling: power usage scales with area, so Moore's law enables higher frequency with little increase in power



- current leakage caused Dennard scaling to cease in 2005

The End of Dennard Scaling

Dennard scaling: power usage scales with area, so Moore's law enables higher frequency with little increase in power



- current leakage caused Dennard scaling to cease in 2005
- so can no longer increase frequency without increasing power, must add cores or other functionality

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions
- SIMD instructions (multiple sets of operands per instruction)

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions
- SIMD instructions (multiple sets of operands per instruction)
- Memory hierarchy (larger caches and deeper hierarchy)

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions
- SIMD instructions (multiple sets of operands per instruction)
- Memory hierarchy (larger caches and deeper hierarchy)
- Multicore and multithreaded processors

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions
- SIMD instructions (multiple sets of operands per instruction)
- Memory hierarchy (larger caches and deeper hierarchy)
- Multicore and multithreaded processors

Consequences of Moore's Law

For given clock speed, increasing performance depends on producing more results per cycle, which can be achieved by exploiting various forms of parallelism

- Pipelined functional units
- Superscalar architecture (multiple instructions per cycle)
- Out-of-order execution of instructions
- SIMD instructions (multiple sets of operands per instruction)
- Memory hierarchy (larger caches and deeper hierarchy)
- Multicore and multithreaded processors

Consequently, almost all processors today are parallel

High Performance Parallel Supercomputers

- Processors in today's cell phones and automobiles are more powerful than supercomputers of twenty years ago

High Performance Parallel Supercomputers

- Processors in today's cell phones and automobiles are more powerful than supercomputers of twenty years ago
- Nevertheless, to attain extreme levels of performance (petaflops and beyond) necessary for large-scale simulations in science and engineering, many processors (often thousands to hundreds of thousands) must work together in concert

High Performance Parallel Supercomputers

- Processors in today's cell phones and automobiles are more powerful than supercomputers of twenty years ago
- Nevertheless, to attain extreme levels of performance (petaflops and beyond) necessary for large-scale simulations in science and engineering, many processors (often thousands to hundreds of thousands) must work together in concert
- This course is about how to design and analyze efficient numerical algorithms for such architectures and applications

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams
  - special purpose, "data parallel" computers

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams
  - special purpose, "data parallel" computers
- *MISD*: multiple instruction streams, single data stream

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams
  - special purpose, "data parallel" computers
- *MISD*: multiple instruction streams, single data stream
  - not particularly useful, except perhaps in "pipelining"

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams
  - special purpose, "data parallel" computers
- *MISD*: multiple instruction streams, single data stream
  - not particularly useful, except perhaps in "pipelining"
- *MIMD*: multiple instruction streams, multiple data streams

Flynn's Taxonomy

Flynn's taxonomy: classification of computer systems by numbers of *instruction* streams and *data* streams:

- *SISD*: single instruction stream, single data stream
  - conventional serial computers
- *SIMD*: single instruction stream, multiple data streams
  - special purpose, "data parallel" computers
- *MISD*: multiple instruction streams, single data stream
  - not particularly useful, except perhaps in "pipelining"
- *MIMD*: multiple instruction streams, multiple data streams
  - general purpose parallel computers

SPMD Programming Style

SPMD (single program, multiple data): all processors execute same program, but each operates on different portion of problem data

- Easier to program than true MIMD, but more flexible than SIMD
- Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style

Architectural Issues

Major architectural issues for parallel computer systems include

- *processor coordination* : synchronous or asynchronous?

Architectural Issues

Major architectural issues for parallel computer systems include

- *processor coordination* : synchronous or asynchronous?
- *memory organization* : distributed or shared?

Architectural Issues

Major architectural issues for parallel computer systems include

- *processor coordination* : synchronous or asynchronous?
- *memory organization* : distributed or shared?
- *address space* : local or global?

Architectural Issues

Major architectural issues for parallel computer systems include

- *processor coordination* : synchronous or asynchronous?
- *memory organization* : distributed or shared?
- *address space* : local or global?
- *memory access* : uniform or nonuniform?

Architectural Issues

Major architectural issues for parallel computer systems include

- *processor coordination*: synchronous or asynchronous?
- *memory organization*: distributed or shared?
- *address space*: local or global?
- *memory access*: uniform or nonuniform?
- *granularity*: coarse or fine?

Architectural Issues

Major architectural issues for parallel computer systems include

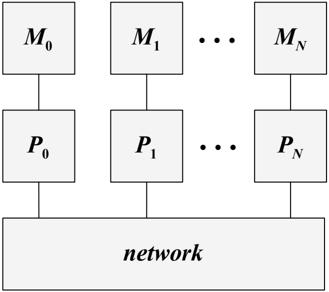
- *processor coordination* : synchronous or asynchronous?
- *memory organization* : distributed or shared?
- *address space* : local or global?
- *memory access* : uniform or nonuniform?
- *granularity* : coarse or fine?
- *scalability* : additional processors used efficiently?

Architectural Issues

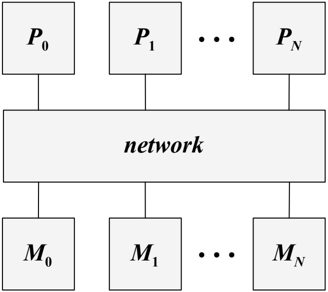
Major architectural issues for parallel computer systems include

- *processor coordination* : synchronous or asynchronous?
- *memory organization* : distributed or shared?
- *address space* : local or global?
- *memory access* : uniform or nonuniform?
- *granularity* : coarse or fine?
- *scalability* : additional processors used efficiently?
- *interconnection network* : topology, switching, routing?

Distributed-Memory and Shared-Memory Systems



distributed-memory multicomputer



shared-memory multiprocessor

Distributed Memory vs. Shared Memory

	distributed memory	shared memory

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization	harder	easier

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization	harder	easier
automatic parallelization		

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization	harder	easier
automatic parallelization	harder	easier

Hybrid systems are common, with memory shared locally within SMP (symmetric multiprocessor) nodes but distributed globally across nodes

Distributed Memory vs. Shared Memory

	distributed memory	shared memory
scalability	easier	harder
data mapping	harder	easier
data integrity	easier	harder
performance optimization	easier	harder
incremental parallelization	harder	easier
automatic parallelization	harder	easier

Hybrid systems are common, with memory shared locally within SMP (symmetric multiprocessor) nodes but distributed globally across nodes

Network Topologies

- Access to remote data requires communication

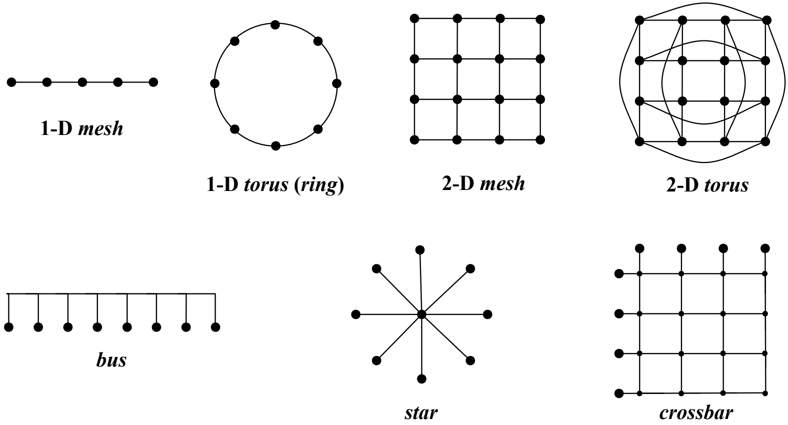
Network Topologies

- Access to remote data requires communication
- Direct connections would require $\mathcal{O}(p^2)$ wires and communication ports, which is infeasible for large p

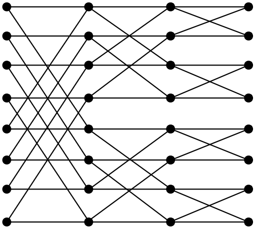
Network Topologies

- Access to remote data requires communication
- Direct connections would require $\mathcal{O}(p^2)$ wires and communication ports, which is infeasible for large p
- Limited connectivity necessitates routing data through intermediate processors or switches

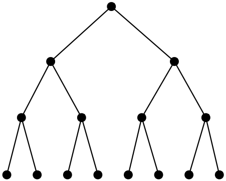
Some Common Network Topologies



Some Common Network Topologies



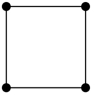
butterfly



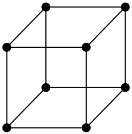
binary tree

0-*cube*

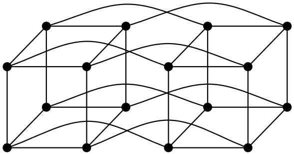
1-*cube*



2-*cube*



3-*cube*



4-*cube*

hypercubes

Graph Terminology

- *Graph*: pair (V, E), where V is set of vertices or nodes connected by set E of edges

Graph Terminology

- *Graph*: pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph*: graph in which any two nodes are connected by an edge

Graph Terminology

- *Graph* : pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph* : graph in which any two nodes are connected by an edge
- *Path* : sequence of contiguous edges in graph

Graph Terminology

- *Graph*: pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph*: graph in which any two nodes are connected by an edge
- *Path*: sequence of contiguous edges in graph
- *Connected graph*: graph in which any two nodes are connected by a path

Graph Terminology

- *Graph*: pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph*: graph in which any two nodes are connected by an edge
- *Path*: sequence of contiguous edges in graph
- *Connected graph*: graph in which any two nodes are connected by a path
- *Cycle*: path of length greater than one that connects a node to itself

Graph Terminology

- *Graph* : pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph* : graph in which any two nodes are connected by an edge
- *Path* : sequence of contiguous edges in graph
- *Connected graph* : graph in which any two nodes are connected by a path
- *Cycle* : path of length greater than one that connects a node to itself
- *Tree* : connected graph containing no cycles

Graph Terminology

- *Graph*: pair (V, E), where V is set of vertices or nodes connected by set E of edges
- *Complete graph*: graph in which any two nodes are connected by an edge
- *Path*: sequence of contiguous edges in graph
- *Connected graph*: graph in which any two nodes are connected by a path
- *Cycle*: path of length greater than one that connects a node to itself
- *Tree*: connected graph containing no cycles
- *Spanning tree*: subgraph that includes all nodes of given graph and is also a tree

Graph Models

- Graph model of network: nodes are processors (or switches or memory units), edges are communication links

Graph Models

- Graph model of network: nodes are processors (or switches or memory units), edges are communication links
- Graph model of computation: nodes are tasks, edges are data dependences between tasks

Graph Models

- Graph model of network: nodes are processors (or switches or memory units), edges are communication links
- Graph model of computation: nodes are tasks, edges are data dependences between tasks
- Mapping task graph of computation to network graph of target computer is instance of *graph embedding*

Graph Models

- Graph model of network: nodes are processors (or switches or memory units), edges are communication links
- Graph model of computation: nodes are tasks, edges are data dependences between tasks
- Mapping task graph of computation to network graph of target computer is instance of *graph embedding*
- *Distance* between two nodes: number of edges (*hops*) in *shortest* path between them

Network Properties

Some network properties affecting its physical realization and potential performance

- *degree* : maximum number of edges incident on any node; determines number of communication ports per processor

Network Properties

Some network properties affecting its physical realization and potential performance

- *degree*: maximum number of edges incident on any node; determines number of communication ports per processor
- *diameter*: maximum distance between any pair of nodes; determines maximum communication delay between processors

Network Properties

Some network properties affecting its physical realization and potential performance

- *degree*: maximum number of edges incident on any node; determines number of communication ports per processor
- *diameter*: maximum distance between any pair of nodes; determines maximum communication delay between processors
- *bisection bandwidth*: (balanced min cut) smallest number of edges whose removal splits graph into two subgraphs of equal size; determines ability to support simultaneous global communication

Network Properties

Some network properties affecting its physical realization and potential performance

- *degree*: maximum number of edges incident on any node; determines number of communication ports per processor
- *diameter*: maximum distance between any pair of nodes; determines maximum communication delay between processors
- *bisection bandwidth*: (balanced min cut) smallest number of edges whose removal splits graph into two subgraphs of equal size; determines ability to support simultaneous global communication
- *edge length*: maximum physical length of any wire; may be constant or variable as number of processors varies

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var
binary tree	2^k-1	3	$2(k-1)$	1	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var
binary tree	2^k-1	3	$2(k-1)$	1	var
hypercube	2^k	k	k	2^{k-1}	var

Network Properties

Network	Nodes	Deg.	Diam.	Bisect. W.	Edge L.
bus/star	$k+1$	k	2	1	var
crossbar	k^2+2k	4	$2(k+1)$	k	var
1-D mesh	k	2	$k-1$	1	const
2-D mesh	k^2	4	$2(k-1)$	k	const
3-D mesh	k^3	6	$3(k-1)$	k^2	const
n-D mesh	k^n	$2n$	$n(k-1)$	k^{n-1}	var
1-D torus	k	2	$k/2$	2	const
2-D torus	k^2	4	k	$2k$	const
3-D torus	k^3	6	$3k/2$	$2k^2$	const
n-D torus	k^n	$2n$	$nk/2$	$2k^{n-1}$	var
binary tree	2^k-1	3	$2(k-1)$	1	var
hypercube	2^k	k	k	2^{k-1}	var
butterfly	$(k+1)2^k$	4	$2k$	2^k	var

Graph Embedding

Graph embedding: $\phi\colon V_s \to V_t$ maps nodes in source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$. Edges in G_s are mapped to paths in G_t.

Graph Embedding

Graph embedding: $\phi\colon V_s \to V_t$ maps nodes in source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$. Edges in G_s are mapped to paths in G_t.

- *load*: maximum number of nodes in V_s mapped to same node in V_t

Graph Embedding

Graph embedding: $\phi\colon V_s \to V_t$ maps nodes in source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$. Edges in G_s are mapped to paths in G_t.

- *load*: maximum number of nodes in V_s mapped to same node in V_t
- *congestion*: maximum number of edges in E_s mapped to paths containing same edge in E_t

Graph Embedding

Graph embedding: $\phi\colon V_s \to V_t$ maps nodes in source graph $G_s = (V_s, E_s)$ to nodes in target graph $G_t = (V_t, E_t)$. Edges in G_s are mapped to paths in G_t.

- *load*: maximum number of nodes in V_s mapped to same node in V_t
- *congestion*: maximum number of edges in E_s mapped to paths containing same edge in E_t
- *dilation*: maximum distance between any two nodes $\phi(u), \phi(v) \in V_t$ such that $(u, v) \in E_s$

Graph Embedding

- Uniform load helps balance work across processors

Graph Embedding

- Uniform load helps balance work across processors
- Minimizing congestion optimizes use of available bandwidth of network links

Graph Embedding

- Uniform load helps balance work across processors
- Minimizing congestion optimizes use of available bandwidth of network links
- Minimizing dilation keeps nearest-neighbor communications in source graph as short as possible in target graph

Graph Embedding

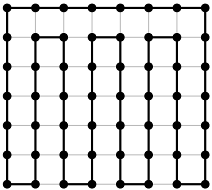
- Uniform load helps balance work across processors
- Minimizing congestion optimizes use of available bandwidth of network links
- Minimizing dilation keeps nearest-neighbor communications in source graph as short as possible in target graph
- Perfect embedding has load, congestion, and dilation 1, but not always possible

Graph Embedding

- Uniform load helps balance work across processors
- Minimizing congestion optimizes use of available bandwidth of network links
- Minimizing dilation keeps nearest-neighbor communications in source graph as short as possible in target graph
- Perfect embedding has load, congestion, and dilation 1, but not always possible
- Optimal embedding difficult to determine (NP-complete, in general), so heuristics used to determine good embedding

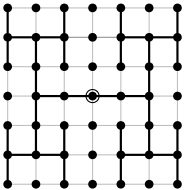
Examples: Graph Embedding

For some important cases, good or optimal embeddings are known, for example



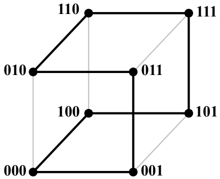
ring in 2-D mesh

dilation 1



binary tree in 2-D mesh

dilation $\lceil (k-1)/2 \rceil$



ring in hypercube

dilation 1

Gray Code

Gray code : ordering of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position

Example: binary reflected Gray code of length 16

$$0000 = 0$$

Gray Code

Gray code : ordering of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position

Example: binary reflected Gray code of length 16

$$\begin{aligned} 0000 &= 0 \\ 0001 &= 1 \end{aligned}$$

Gray Code

Gray code : ordering of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position

Example: binary reflected Gray code of length 16

$$\begin{aligned} 0000 &= 0 \\ 0001 &= 1 \\ 0011 &= 3 \\ 0010 &= 2 \end{aligned}$$

Gray Code

Gray code: ordering of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position

Example: binary reflected Gray code of length 16

$$\begin{aligned} 0000 &= 0 \\ 0001 &= 1 \\ 0011 &= 3 \\ 0010 &= 2 \\ 0110 &= 6 \\ 0111 &= 7 \\ 0101 &= 5 \\ 0100 &= 4 \end{aligned}$$

Gray Code

Gray code : ordering of integers 0 to $2^k - 1$ such that consecutive members differ in exactly one bit position

Example: binary reflected Gray code of length 16

$0000 = 0$		$1100 = 12$
$0001 = 1$		$1101 = 13$
$0011 = 3$		$1111 = 15$
$0010 = 2$		$1110 = 14$
$0110 = 6$		$1010 = 10$
$0111 = 7$		$1011 = 11$
$0101 = 5$		$1001 = 9$
$0100 = 4$		$1000 = 8$

Hypercubes

- Hypercube of dimension k, or k-cube, is graph with 2^k nodes numbered $0, \ldots, 2^k - 1$, and edges between all pairs of nodes whose binary numbers differ in one bit position

Hypercubes

- Hypercube of dimension k, or k-cube, is graph with 2^k nodes numbered $0, \ldots, 2^k - 1$, and edges between all pairs of nodes whose binary numbers differ in one bit position
- Hypercube of dimension k can be created recursively by replicating hypercube of dimension $k - 1$ and connecting their corresponding nodes

Hypercubes

- Hypercube of dimension k, or k-cube, is graph with 2^k nodes numbered $0, \ldots, 2^k - 1$, and edges between all pairs of nodes whose binary numbers differ in one bit position
- Hypercube of dimension k can be created recursively by replicating hypercube of dimension $k - 1$ and connecting their corresponding nodes
- Visiting nodes of hypercube in Gray code order gives *Hamiltonian cycle*, embedding ring in hypercube

Hypercubes

- Hypercube of dimension k, or k-cube, is graph with 2^k nodes numbered $0, \dots, 2^k - 1$, and edges between all pairs of nodes whose binary numbers differ in one bit position
- Hypercube of dimension k can be created recursively by replicating hypercube of dimension $k - 1$ and connecting their corresponding nodes
- Visiting nodes of hypercube in Gray code order gives *Hamiltonian cycle*, embedding ring in hypercube
- For mesh or torus of higher dimension, concatenating Gray codes for each dimension gives embedding in hypercube

Hypercubes

- Hypercube of dimension k, or k-cube, is graph with 2^k nodes numbered $0, \ldots, 2^k - 1$, and edges between all pairs of nodes whose binary numbers differ in one bit position
- Hypercube of dimension k can be created recursively by replicating hypercube of dimension $k - 1$ and connecting their corresponding nodes
- Visiting nodes of hypercube in Gray code order gives *Hamiltonian cycle*, embedding ring in hypercube
- For mesh or torus of higher dimension, concatenating Gray codes for each dimension gives embedding in hypercube
- Hypercubes provide an effective paradigm for low-diameter target network in designing parallel algorithms

Optimality in Network Topology Design

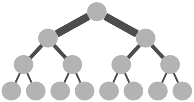
- Hypercubes are near-optimal networks, in the sense that they can execute any communication pattern with $O(\log(p))$ slowdown via randomizing the data layout

Optimality in Network Topology Design

- Hypercubes are near-optimal networks, in the sense that they can execute any communication pattern with $O(\log(p))$ slowdown via randomizing the data layout
- A more refined notion of optimality considers the physical space necessary to build the network

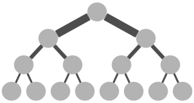
Optimality in Network Topology Design

- Hypercubes are near-optimal networks, in the sense that they can execute any communication pattern with $O(\log(p))$ slowdown via randomizing the data layout
- A more refined notion of optimality considers the physical space necessary to build the network
  - Fat trees (switched binary trees) which assign each link more bandwidth to higher-level switches are optimal in this sense within polylogarithmic factors



Optimality in Network Topology Design

- Hypercubes are near-optimal networks, in the sense that they can execute any communication pattern with $O(\log(p))$ slowdown via randomizing the data layout
- A more refined notion of optimality considers the physical space necessary to build the network
  - Fat trees (switched binary trees) which assign each link more bandwidth to higher-level switches are optimal in this sense within polylogarithmic factors



  - When increasing processors, bisection bandwidth scales with $O(p^{2/3})$ as opposed to $O(1)$ for binary trees

Low diameter networks

- The Cray Dragonfly network has diameter 3

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes
  - a single pair of nodes connects each pair of groups

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes
  - a single pair of nodes connects each pair of groups
- Given a target diameter r, the Moore bound provides a lower-bound on the degree d

$$p \leq 1 + d\sum_{i=1}^{r-1}(d-1)^i \qquad \left(\text{asymptotically}, p = O(d^r)\right)$$

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes
  - a single pair of nodes connects each pair of groups
- Given a target diameter r, the Moore bound provides a lower-bound on the degree d

$$p \leq 1 + d\sum_{i=1}^{r-1}(d-1)^i \qquad \left(\text{asymptotically}, p = O(d^r)\right)$$

- Slim Fly nearly attains this bound for diameter 2

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes
  - a single pair of nodes connects each pair of groups
- Given a target diameter r, the Moore bound provides a lower-bound on the degree d

$$p \leq 1 + d\sum_{i=1}^{r-1}(d-1)^i \qquad \left(\text{asymptotically}, p = O(d^r)\right)$$

- Slim Fly nearly attains this bound for diameter 2
- Slim Fly arranges processors into two 2D grids, with each processor connecting to some nodes in its columns and some nodes in the other grid

Low diameter networks

- The Cray Dragonfly network has diameter 3
  - define densely connected groups (cliques) of nodes
  - a single pair of nodes connects each pair of groups
- Given a target diameter r, the Moore bound provides a lower-bound on the degree d

$$p \leq 1 + d\sum_{i=1}^{r-1}(d-1)^i \qquad \left(\text{asymptotically}, p = O(d^r)\right)$$

- Slim Fly nearly attains this bound for diameter 2
- Slim Fly arranges processors into two 2D grids, with each processor connecting to some nodes in its columns and some nodes in the other grid
- The Slim Fly network yields degree of roughly $\sqrt{p}$

Topology-Awareness in Algorithms

- *Topology-aware algorithms* aim to execute effectively on specific network topologies

Topology-Awareness in Algorithms

- *Topology-aware algorithms* aim to execute effectively on specific network topologies
- If mapped ideally to a network topology, applications and algorithms often see significant performance gains

Topology-Awareness in Algorithms

- *Topology-aware algorithms* aim to execute effectively on specific network topologies
- If mapped ideally to a network topology, applications and algorithms often see significant performance gains
- However, real applications are executed on a subset of nodes of a distributed machine, which may not have the same connectivity structure as the overall machine

Topology-Awareness in Algorithms

- *Topology-aware algorithms* aim to execute effectively on specific network topologies
- If mapped ideally to a network topology, applications and algorithms often see significant performance gains
- However, real applications are executed on a subset of nodes of a distributed machine, which may not have the same connectivity structure as the overall machine
- Moreover, network-topology-specific optimziations are typically not performance-portable

Topology-Awareness in Algorithms

- *Topology-aware algorithms* aim to execute effectively on specific network topologies
- If mapped ideally to a network topology, applications and algorithms often see significant performance gains
- However, real applications are executed on a subset of nodes of a distributed machine, which may not have the same connectivity structure as the overall machine
- Moreover, network-topology-specific optimziations are typically not performance-portable
- Nevertheless, topologies provide a convenient visual model for design of parallel algorithms

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones
- An algorithm designed for a densly-connected network typically incurs a bounded amount of overhead on more sparsely-connected ones

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones
- An algorithm designed for a densly-connected network typically incurs a bounded amount of overhead on more sparsely-connected ones
- Ideally, parallel algorithms should be *topology-oblivious*, i.e. perform well on any reasonable network topology

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones
- An algorithm designed for a densly-connected network typically incurs a bounded amount of overhead on more sparsely-connected ones
- Ideally, parallel algorithms should be *topology-oblivious*, i.e. perform well on any reasonable network topology
- A good parallel algorithm design methodology is to

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones
- An algorithm designed for a densly-connected network typically incurs a bounded amount of overhead on more sparsely-connected ones
- Ideally, parallel algorithms should be *topology-oblivious*, i.e. perform well on any reasonable network topology
- A good parallel algorithm design methodology is to
  1. try to obtain cost-optimality for a fully-connected network

Topology-Obliviousness in Algorithms

- An algorithm designed for a sparsely-connected network is typically as efficient on more densly-connected ones
- An algorithm designed for a densly-connected network typically incurs a bounded amount of overhead on more sparsely-connected ones
- Ideally, parallel algorithms should be *topology-oblivious*, i.e. perform well on any reasonable network topology
- A good parallel algorithm design methodology is to
  1. try to obtain cost-optimality for a fully-connected network
  2. organize it so it achieves the same cost on some network topology that is as sparsely-connected as possible

Message Passing

Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = \alpha + \beta\,\mathrm{s}$$

- α = *startup time* = *latency* (i.e., time to send message of length zero)

Message Passing

Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = \alpha + \beta\, \text{s}$$

- α = *startup time* = *latency* (i.e., time to send message of length zero)
- β = incremental *transfer time* per word ($1/\beta$ = *bandwidth* in words per unit time)

Message Passing

Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = \alpha + \beta\,\text{s}$$

- α = *startup time* = *latency* (i.e., time to send message of length zero)
- β = incremental *transfer time* per word ($1/\beta$ = *bandwidth* in words per unit time)
- s = *length* of message in words

Message Passing

Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = \alpha + \beta\,\mathrm{s}$$

- α = *startup time* = *latency* (i.e., time to send message of length zero)
- β = incremental *transfer time* per word ($1/\beta$ = *bandwidth* in words per unit time)
- s = *length* of message in words

Message Passing

Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = \alpha + \beta\,\text{s}$$

- α = *startup time* = *latency* (i.e., time to send message of length zero)
- β = incremental *transfer time* per word ($1/\beta$ = *bandwidth* in words per unit time)
- s = *length* of message in words

For real parallel systems $\alpha \gg \beta$, so we often simplify $\alpha + \beta \approx \alpha$

Algorithmic Communication Cost

Let p processors send a message of size s in a ring

- ps is the *communication volume* (total amount of data sent)

Algorithmic Communication Cost

Let p processors send a message of size s in a ring

- ps is the *communication volume* (total amount of data sent)
- However, the execution time depends on whether we send the messages concurrently or in sequence

Algorithmic Communication Cost

Let p processors send a message of size s in a ring

- ps is the *communication volume* (total amount of data sent)
- However, the execution time depends on whether we send the messages concurrently or in sequence
- The *communication time* models execution time in terms of per-message costs

Algorithmic Communication Cost

Let p processors send a message of size s in a ring

- ps is the *communication volume* (total amount of data sent)
- However, the execution time depends on whether we send the messages concurrently or in sequence
- The *communication time* models execution time in terms of per-message costs
  - if the messages are sent *simultaneously*,

$$T_{\text{sim-ring}}(s) = T_{\text{msg}}(s) = \alpha + s \cdot \beta$$

Algorithmic Communication Cost

Let p processors send a message of size s in a ring

- ps is the *communication volume* (total amount of data sent)
- However, the execution time depends on whether we send the messages concurrently or in sequence
- The *communication time* models execution time in terms of per-message costs
  - if the messages are sent *simultaneously*,

$$T_{\text{sim-ring}}(s) = T_{\text{msg}}(s) = \alpha + s \cdot \beta$$

  - if the messages are sent *in sequence*,

$$T_{\text{seq-ring}}(s, p) = p \cdot T_{\text{msg}}(s) = p \cdot (\alpha + s \cdot \beta)$$

Message Routing

Messages sent between nodes that are not directly connected must be *routed* through intermediate nodes

Message routing algorithms can be

- *minimal* or *nonminimal*, depending on whether shortest path is always taken

Message Routing

Messages sent between nodes that are not directly connected must be *routed* through intermediate nodes

Message routing algorithms can be

- *minimal* or *nonminimal*, depending on whether shortest path is always taken
- *static* or *dynamic*, depending on whether same path is always taken

Message Routing

Messages sent between nodes that are not directly connected must be *routed* through intermediate nodes

Message routing algorithms can be

- *minimal* or *nonminimal*, depending on whether shortest path is always taken
- *static* or *dynamic*, depending on whether same path is always taken
- *deterministic* or *randomized*, depending on whether path is chosen systematically or randomly

Message Routing

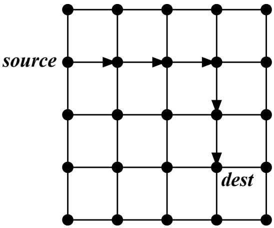
Messages sent between nodes that are not directly connected must be *routed* through intermediate nodes

Message routing algorithms can be

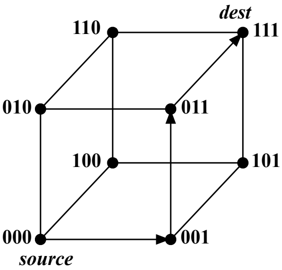
- *minimal* or *nonminimal*, depending on whether shortest path is always taken
- *static* or *dynamic*, depending on whether same path is always taken
- *deterministic* or *randomized*, depending on whether path is chosen systematically or randomly
- *circuit switched* or *packet switched*, depending on whether entire message goes along reserved path or is transferred in segments that may not all take same path

Message Routing

Most regular network topologies admit simple routing schemes that are static, deterministic, and minimal



2-D mesh



hypercube

Store-and-Forward vs. Cut-Through Routing

Store-and-forward routing: entire message is received and stored at each node before being forwarded to next node on path, so

$$T_{\text{route}} = (\alpha + \beta \mathrm{s})D, \text{ where } D = \text{distance in hops}$$

Store-and-Forward vs. Cut-Through Routing

Store-and-forward routing: entire message is received and stored at each node before being forwarded to next node on path, so

$$T_{\text{route}} = (\alpha + \beta \text{s}) D, \text{ where } D = \text{distance in hops}$$

Cut-through (or *wormhole*) routing: message broken into segments that are pipelined through network, with each segment forwarded as soon as it is received, so

$$T_{\text{route}} = \alpha + \beta \text{s} + t_h D, \text{ where } t_h = \text{incremental time per hop}$$

Store-and-Forward vs. Cut-Through Routing

Store-and-forward routing: entire message is received and stored at each node before being forwarded to next node on path, so

$$T_{\text{route}} = (\alpha + \beta \text{s})D, \text{ where } D = \text{distance in hops}$$

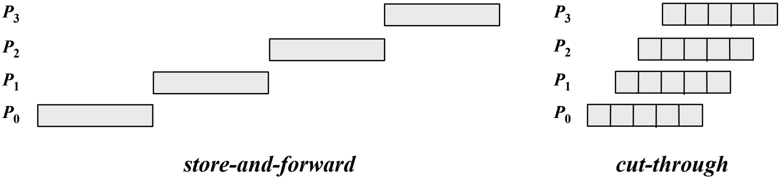
Cut-through (or *wormhole*) routing: message broken into segments that are pipelined through network, with each segment forwarded as soon as it is received, so

$$T_{\text{route}} = \alpha + \beta \text{s} + t_h D, \text{ where } t_h = \text{incremental time per hop}$$

Generally $t_h \leq \alpha$, so we can treat both as network latency,

$$T_{\text{route}} = \alpha D + \beta \text{s}$$

Store-and-Forward vs. Worlmhole Routing



Cut-through (wormhole) routing greatly reduces distance effect, but aggregate bandwidth may still be significant constraint

Communication Concurrency

For given communication system, it may or may not be possible for each node to

- send message while receiving another simultaneously on *same* communication link

Communication Concurrency

For given communication system, it may or may not be possible for each node to

- send message while receiving another simultaneously on *same* communication link
- send message on one link while receiving simultaneously on *different* link

Communication Concurrency

For given communication system, it may or may not be possible for each node to

- send message while receiving another simultaneously on *same* communication link
- send message on one link while receiving simultaneously on *different* link
- send or receive, or both, simultaneously on *multiple* links

Communication Concurrency

For given communication system, it may or may not be possible for each node to

- send message while receiving another simultaneously on *same* communication link
- send message on one link while receiving simultaneously on *different* link
- send or receive, or both, simultaneously on *multiple* links

Communication Concurrency

For given communication system, it may or may not be possible for each node to

- send message while receiving another simultaneously on *same* communication link
- send message on one link while receiving simultaneously on *different* link
- send or receive, or both, simultaneously on *multiple* links

We will generally assume a processor can send or receive only one message at a time (but can send one and receive one simultaneously).

Collective Communication

Collective communication: multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter

Collective Communication

Collective communication : multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather

Collective Communication

Collective communication: multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather
- *All-to-One + One-to-All*: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan

Collective Communication

Collective communication : multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather
- *All-to-One + One-to-All*: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- *All-to-All*: All-to-all

Collective Communication

Collective communication: multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather
- *All-to-One + One-to-All*: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- *All-to-All*: All-to-all

Collective Communication

Collective communication: multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather
- *All-to-One + One-to-All*: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- *All-to-All*: All-to-all

The distinction between the last two types is made due to their different cost characteristics

Collective Communication

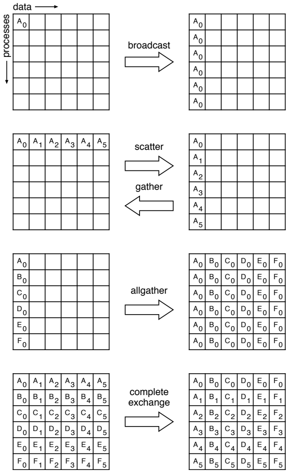
Collective communication : multiple nodes communicating simultaneously in systematic pattern, which we can classify as

- *One-to-All*: Broadcast, Scatter
- *All-to-One*: Reduce, Gather
- *All-to-One + One-to-All*: Allreduce (Reduce+Broadcast), Allgather (Gather+Broadcast), Reduce-Scatter (Reduce+Scatter), Scan
- *All-to-All*: All-to-all

The distinction between the last two types is made due to their different cost characteristics

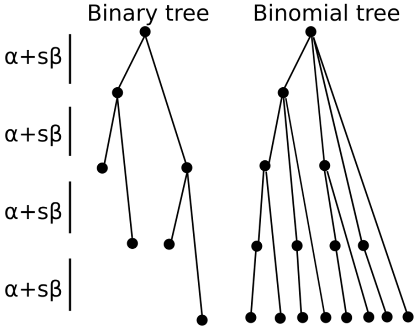
MPI (Message-Passing Interface) provides all of these as well as variable size versions (e.g. (All)Gatherv, All-to-allv).

Collective Communication



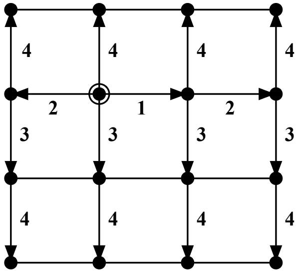
Broadcast

Broadcast: *source* node sends same message of size s to each of $p-1$ other nodes

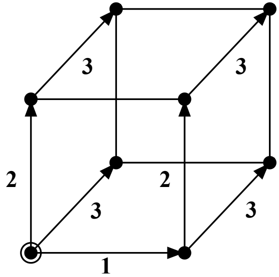


Binary or binomial trees are often used for one-to-all collectives like broadcast, but any spanning tree will do

Broadcast



2-D mesh



hypercube

Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p - 1)\,(\alpha + \beta \mathrm{s})$

Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p - 1)\,(\alpha + \beta \mathrm{s})$
- 2-D mesh: $T = 2(\sqrt{p} - 1)\,(\alpha + \beta \mathrm{s})$

Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p-1)\,(\alpha + \beta \mathrm{s})$
- 2-D mesh: $T = 2(\sqrt{p}-1)\,(\alpha + \beta \mathrm{s})$
- hypercube: $T = \log p\,(\alpha + \beta \mathrm{s})$

Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p - 1)\,(\alpha + \beta \mathrm{s})$
- 2-D mesh: $T = 2(\sqrt{p} - 1)\,(\alpha + \beta \mathrm{s})$
- hypercube: $T = \log p\,(\alpha + \beta \mathrm{s})$

Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p-1)\,(\alpha + \beta \mathrm{s})$
- 2-D mesh: $T = 2(\sqrt{p}-1)\,(\alpha + \beta \mathrm{s})$
- hypercube: $T = \log p\,(\alpha + \beta \mathrm{s})$

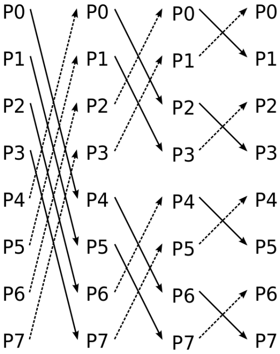
For long messages, bandwidth utilization may be enhanced by breaking message into segments and either

- *pipeline* segments along *single* spanning tree, or
- send each segment along *different* spanning tree having *same* root

For example, hypercube with 2^k nodes has k *edge-disjoint* spanning trees for any given root node

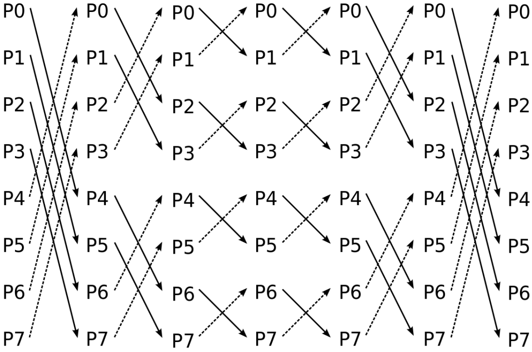
Butterfly Protocols

All collective-communication can be done near-optimally with *butterfly protocols*, which use all links of a hypercube network



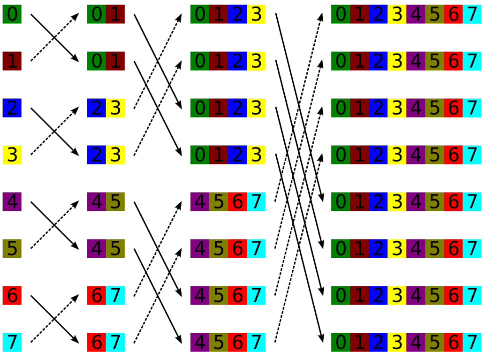
Butterfly Protocols

All collective-communication can be done near-optimally with *butterfly protocols*, which use all links of a hypercube network



Butterfly Allgather (Recursive Doubling)

Allgather: each of p nodes sends message to all other nodes



Cost of Butterfly Allgather

The butterfly has $\log_2(p)$ levels. The size of the message doubles at each level until all s elements are gathered, so the total cost is

$$T_{\text{allgather}}(s,p) = \begin{cases} 0 & : p = 1 \\ T_{\text{allgather}}(s/2, p/2) + \alpha + \beta(s/2) & : p > 1 \end{cases}$$

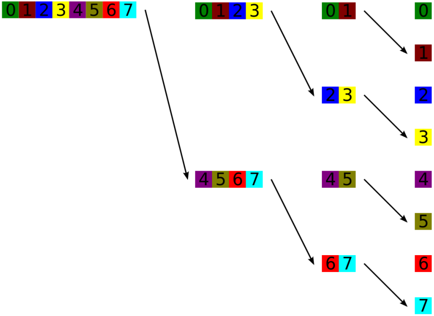
$$\approx \alpha \log_2(p) + \sum_{i=1}^{\log_2(p)} \beta s/2^i$$

$$\approx \alpha \log_2(p) + \beta s$$

The geometric summation in the cost analysis is typical for butterfly protocols for one-to-all and all-to-one collectives

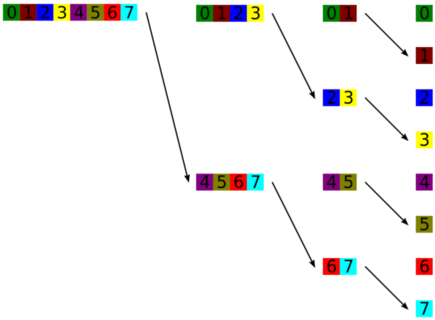
Butterfly Scatter

Scatter: *source* node sends message of size s/p to each of $p - 1$ other nodes



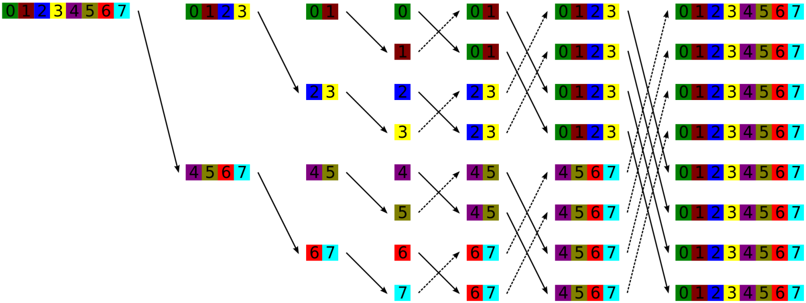
Butterfly Scatter

Scatter: *source* node sends message of size s/p to each of $p-1$ other nodes

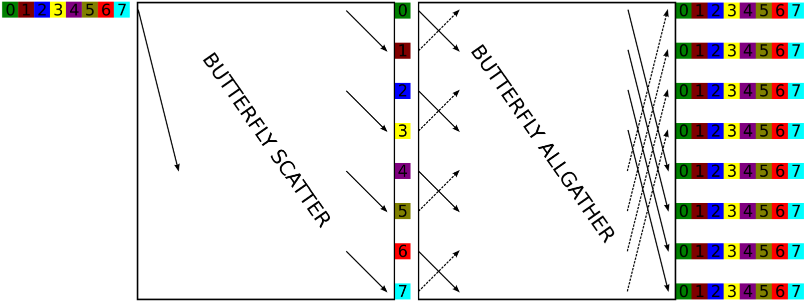


Note that the messages are forwarded down a binomial and not a binary spanning tree of nodes.

Butterfly Broadcast



Butterfly Broadcast



$$T_{\text{broadcast}} = T_{\text{scatter}} + T_{\text{allgather}} = 2T_{\text{allgather}}$$

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

Generally, we can turn any broadcast algorithm into a reduction algorithm by reversing the flow of information, so we see

- Broadcast done effectively by Scatter + Allgather

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

Generally, we can turn any broadcast algorithm into a reduction algorithm by reversing the flow of information, so we see

- Broadcast done effectively by Scatter + Allgather
- Reduction done effectively by Reduce-Scatter + Gather

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

Generally, we can turn any broadcast algorithm into a reduction algorithm by reversing the flow of information, so we see

- Broadcast done effectively by Scatter + Allgather
- Reduction done effectively by Reduce-Scatter + Gather
- Allreduce done effectively by Reduce-Scatter + Allgather

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

Generally, we can turn any broadcast algorithm into a reduction algorithm by reversing the flow of information, so we see

- Broadcast done effectively by Scatter + Allgather
- Reduction done effectively by Reduce-Scatter + Gather
- Allreduce done effectively by Reduce-Scatter + Allgather

Reduction

Reduction : data from all p nodes are combined by applying specified associative operation $\oplus$ (e.g., sum, product, max, min, logical OR, logical AND) to produce overall result

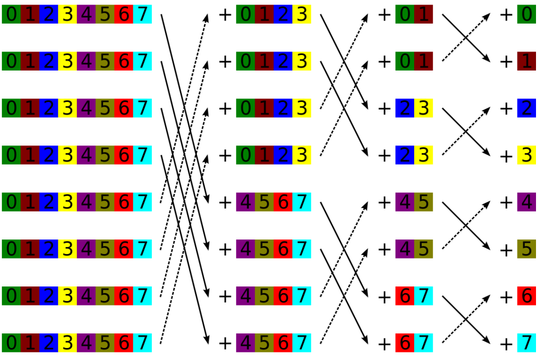
Generally, we can turn any broadcast algorithm into a reduction algorithm by reversing the flow of information, so we see

- Broadcast done effectively by Scatter + Allgather
- Reduction done effectively by Reduce-Scatter + Gather
- Allreduce done effectively by Reduce-Scatter + Allgather

These one-to-all + all-to-one collectives have butterfly protocols with equivalent cost.

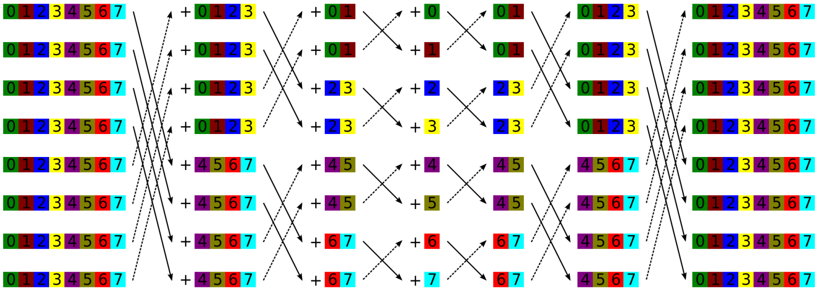
Butterfly Reduce-Scatter (Recursive Halving)

Reduce-scatter: a reduction with the result *distributed* over all p nodes

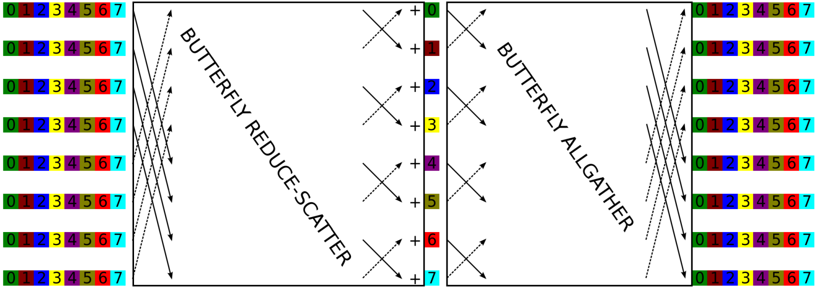


Butterfly Allreduce

Allreduce: a reduction with the result *replicated* on all p nodes

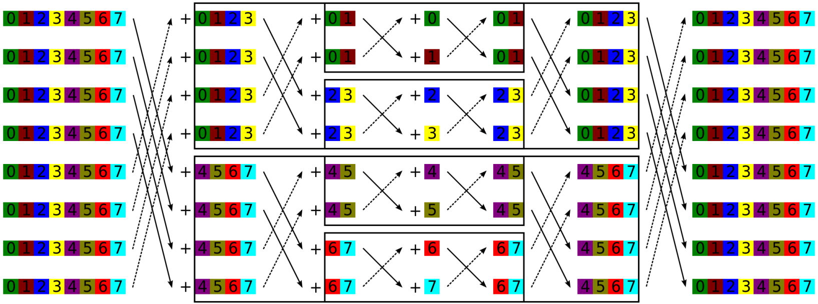


Butterfly Allreduce



$$T_{\text{allreduce}} = T_{\text{reduce-scatter}} + T_{\text{allgather}}$$

Butterfly Allreduce: note recursive structure of butterfly



Scan or Prefix

Scan or *prefix*: given data values $x_0, x_1, \ldots, x_{p-1}$, one per node, along with associative operation $\oplus$, compute sequence of partial results $y_0, y_1, \ldots, y_{p-1}$, where

$$y_k = x_0 \oplus x_1 \oplus \cdots \oplus x_k,$$

and y_k is to reside on node k, $k = 0, \ldots p - 1$

Scan or Prefix

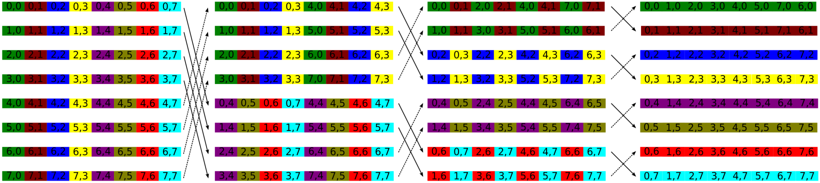
Scan or *prefix*: given data values $x_0, x_1, \ldots, x_{p-1}$, one per node, along with associative operation $\oplus$, compute sequence of partial results $y_0, y_1, \ldots, y_{p-1}$, where

$$y_k = x_0 \oplus x_1 \oplus \cdots \oplus x_k,$$

and y_k is to reside on node k, $k = 0, \ldots p - 1$

Scan can be implemented via a butterfly protocol similar to Allreduce, except intermediate results must be stored while doing recursive halving to be recombined when doing recursive doubling

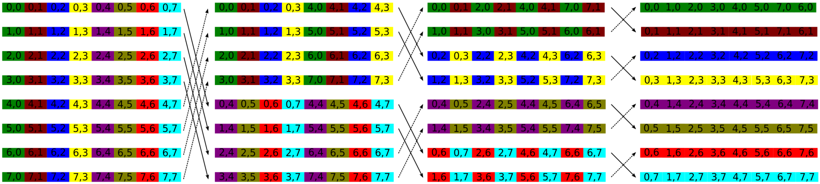
Butterfly All-to-All



The size of the message stays the same at each level, so

$$T_{\text{all-to-all}}(s, P) = \qquad = \alpha \log_2(P) + \beta s \log_2(P)/2$$

Butterfly All-to-All



The size of the message stays the same at each level, so

$$T_{\text{all-to-all}}(s, P) = \qquad = \alpha \log_2(P) + \beta s \log_2(P)/2$$

Its possible to do All-to-All in less bandwidth cost (as low as βs by sending directly to targets) at the cost of more messages (as high as αP if sending directly)

Collectives on Mesh and Torus Networks

Butterfly protocols cannot be mapped to tori without dilation

- bandwidth-efficient collectives can be achieved by instead pipelining along spanning trees

Collectives on Mesh and Torus Networks

Butterfly protocols cannot be mapped to tori without dilation

- bandwidth-efficient collectives can be achieved by instead pipelining along spanning trees
- if height of spanning tree is H (e.g. $H \approx 2\sqrt{p}$ for 2D mesh), then cost of one-to-all and all-to-one collectives is

$$T_{\text{one-to-all}}(s, p, H) = \Theta(\alpha H + \beta s)$$

Collectives on Mesh and Torus Networks

Butterfly protocols cannot be mapped to tori without dilation

- bandwidth-efficient collectives can be achieved by instead pipelining along spanning trees
- if height of spanning tree is H (e.g. $H \approx 2\sqrt{p}$ for 2D mesh), then cost of one-to-all and all-to-one collectives is

$$T_{\text{one-to-all}}(s, p, H) = \Theta(\alpha H + \beta s)$$

- hypercube (general) cost is recovered with $H = \log_2(p)$

Collectives on Mesh and Torus Networks

Butterfly protocols cannot be mapped to tori without dilation

- bandwidth-efficient collectives can be achieved by instead pipelining along spanning trees
- if height of spanning tree is H (e.g. $H \approx 2\sqrt{p}$ for 2D mesh), then cost of one-to-all and all-to-one collectives is

$$T_{\text{one-to-all}}(s, p, H) = \Theta(\alpha H + \beta s)$$

- hypercube (general) cost is recovered with $H = \log_2(p)$
- use of more than one disjoint spanning trees (rectangular collectives) is beneficial if processors can send and receive messages along multiple links concurrently

Collectives on Mesh and Torus Networks

Butterfly protocols cannot be mapped to tori without dilation

- bandwidth-efficient collectives can be achieved by instead pipelining along spanning trees
- if height of spanning tree is H (e.g. $H \approx 2\sqrt{p}$ for 2D mesh), then cost of one-to-all and all-to-one collectives is

$$T_{\text{one-to-all}}(s, p, H) = \Theta(\alpha H + \beta s)$$

- hypercube (general) cost is recovered with $H = \log_2(p)$
- use of more than one disjoint spanning trees (rectangular collectives) is beneficial if processors can send and receive messages along multiple links concurrently
- all-to-all cost generally depends on the bisection bandwidth of the network (proportional to $p^{(d-1)/d}$ for d-dimensional torus/mesh)

References – Moore's Law

- M. T. Heath, A tale of two laws, *International Journal of High Performance Computing Applications*, 29(3):320-330, 2015
- C. A. Mack, Fifty years of Moore's law, *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202-207, 2011

References – Parallel Computing

- G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd ed., Benjamin/Cummings, 1994
- J. Dongarra, et al., eds., *Sourcebook of Parallel Computing*, Morgan Kaufmann, 2003
- A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd. ed., Addison-Wesley, 2003
- G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, Chapman & Hall, 2011
- K. Hwang and Z. Xu, *Scalable Parallel Computing*, McGraw-Hill, 1998
- A. Y. Zomaya, ed., *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996

References – Parallel Architectures

- W. C. Athas and C. L. Seitz, Multicomputers: message-passing concurrent computers, *IEEE Computer* 21(8):9-24, 1988
- D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1998
- M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*, Cambridge University Press, 2012
- R. Duncan, A survey of parallel computer architectures, *IEEE Computer* 23(2):5-16, 1990
- F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1992

References – Interconnection Networks

- L. N. Bhuyan, Q. Yang, and D. P. Agarwal, Performance of multiprocessor interconnection networks, *IEEE Computer* 22(2):25-37, 1989
- W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann, 2004
- T. Y. Feng, A survey of interconnection networks, *IEEE Computer* 14(12):12-27, 1981
- I. D. Scherson and A. S. Youssef, eds., *Interconnection Networks for High-Performance Parallel Computers*, IEEE Computer Society Press, 1994
- H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing*, D. C. Heath, 1985
- C.-L. Wu and T.-Y. Feng, eds., *Interconnection Networks for Parallel and Distributed Processing*, IEEE Computer Society Press, 1984

References – Hypercubes

- D. P. Bertsekas *et al.*, Optimal communication algorithms for hypercubes, *J. Parallel Distrib. Comput.* 11:263-275, 1991
- S. L. Johnsson and C.-T. Ho, Optimum broadcasting and personalized communication in hypercubes, *IEEE Trans. Comput.* 38:1249-1268, 1989
- O. McBryan and E. F. Van de Velde, Hypercube algorithms and implementations, *SIAM J. Sci. Stat. Comput.* 8:s227-s287, 1987
- S. Ranka, Y. Won, and S. Sahni, Programming a hypercube multicomputer, *IEEE Software* 69-77, September 1988
- Y. Saad and M. H. Schultz, Topological properties of hypercubes, *IEEE Trans. Comput.* 37:867-872, 1988
- Y. Saad and M. H. Schultz, Data communication in hypercubes, *J. Parallel Distrib. Comput.* 6:115-135, 1989
- C. L. Seitz, The cosmic cube, *Comm. ACM* 28:22-33, 1985