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
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
- 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

- Curve shows transistor count doubling every two years.
**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.
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
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**: 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
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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Distributed Memory</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>Data mapping</td>
<td>harder</td>
<td>easier</td>
</tr>
<tr>
<td>Data integrity</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>Performance optimization</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>Incremental parallelization</td>
<td>harder</td>
<td>easier</td>
</tr>
<tr>
<td>Automatic parallelization</td>
<td>harder</td>
<td>easier</td>
</tr>
</tbody>
</table>

Hybrid systems are common, with memory shared locally within SMP (symmetric multiprocessor) nodes but distributed globally across nodes.
### Distributed Memory vs. Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>Distributed Memory</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalability</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>data mapping</td>
<td>harder</td>
<td>easier</td>
</tr>
<tr>
<td>data integrity</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>performance optimization</td>
<td>easier</td>
<td>harder</td>
</tr>
<tr>
<td>incremental parallelization</td>
<td>harder</td>
<td>easier</td>
</tr>
<tr>
<td>automatic parallelization</td>
<td>harder</td>
<td>easier</td>
</tr>
</tbody>
</table>

Hybrid systems are common, with memory shared locally within SMP (symmetric multiprocessor) nodes but distributed globally across nodes.
Access to remote data requires communication

Direct connections would require $O(p^2)$ wires and communication ports, which is infeasible for large $p$

Limited connectivity necessitates routing data through intermediate processors or switches
Some Common Network Topologies

- **1-D mesh**
- **1-D torus (ring)**
- **2-D mesh**
- **2-D torus**
- **bus**
- **star**
- **crossbar**
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
- **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 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.
- **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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bus/star</td>
<td>$k + 1$</td>
<td>$k$</td>
<td>2</td>
<td>1</td>
<td>var</td>
</tr>
<tr>
<td>crossbar</td>
<td>$k^2 + 2k$</td>
<td>4</td>
<td>$2(k + 1)$</td>
<td>$k$</td>
<td>var</td>
</tr>
<tr>
<td>1-D mesh</td>
<td>$k$</td>
<td>2</td>
<td>$k - 1$</td>
<td>1</td>
<td>const</td>
</tr>
<tr>
<td>2-D mesh</td>
<td>$k^2$</td>
<td>4</td>
<td>$2(k - 1)$</td>
<td>$k$</td>
<td>const</td>
</tr>
<tr>
<td>3-D mesh</td>
<td>$k^3$</td>
<td>6</td>
<td>$3(k - 1)$</td>
<td>$k^2$</td>
<td>const</td>
</tr>
<tr>
<td>n-D mesh</td>
<td>$k^n$</td>
<td>$2n$</td>
<td>$n(k - 1)$</td>
<td>$k^{n-1}$</td>
<td>var</td>
</tr>
<tr>
<td>1-D torus</td>
<td>$k$</td>
<td>2</td>
<td>$k/2$</td>
<td>2</td>
<td>const</td>
</tr>
<tr>
<td>2-D torus</td>
<td>$k^2$</td>
<td>4</td>
<td>$k$</td>
<td>$2k$</td>
<td>const</td>
</tr>
<tr>
<td>3-D torus</td>
<td>$k^3$</td>
<td>6</td>
<td>$3k/2$</td>
<td>$2k^2$</td>
<td>const</td>
</tr>
<tr>
<td>n-D torus</td>
<td>$k^n$</td>
<td>$2n$</td>
<td>$nk/2$</td>
<td>$2k^{n-1}$</td>
<td>var</td>
</tr>
<tr>
<td>binary tree</td>
<td>$2^k - 1$</td>
<td>3</td>
<td>$2(k - 1)$</td>
<td>1</td>
<td>var</td>
</tr>
<tr>
<td>hypercube</td>
<td>$2^k$</td>
<td>$k$</td>
<td>$k$</td>
<td>$2^{k-1}$</td>
<td>var</td>
</tr>
<tr>
<td>butterfly</td>
<td>$(k + 1)2^k$</td>
<td>4</td>
<td>$2k$</td>
<td>$2^k$</td>
<td>var</td>
</tr>
</tbody>
</table>
**Graph embedding**: \( \phi : V_s \to V_t \) maps nodes in source graph \( G_s = (V_s, E_s) \) to nodes in target graph \( G_t = (V_t, E_t) \). Edges in \( G_s \) 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
- 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
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

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
</tbody>
</table>
Computing Binary Reflected Gray Code

```c
/* Gray code */
int gray(int i) {
    return ((i >> 1) ^ i);
}

/* inverse Gray code */
int inv_gray(int i) {
    int k; k = i;
    while (k > 0) {
        k >>= 1;
        i ^= k;
    }
    return i;
}
```
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 a 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 elegant paradigm for low-diameter target network in designing parallel algorithms.
Motivation
Architectures
Networks
Communication

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
  - 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 \( d \)
  \[
  p \leq 1 + d \sum_{i=1}^{r-1} (d - 1)^i \quad \left( \text{asymptotically, } p = O(d^r) \right)
  \]
- 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.
- 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 optimizations are typically not performance-portable.
- Nevertheless, topologies provide a convenient visual model for design of parallel algorithms.
An algorithm designed for a sparsely-connected network is typically as efficient on more densely-connected ones.

An algorithm designed for a densely-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:

\[ T_{\text{msg}} = \alpha + \beta s \]

- \( \alpha = \text{startup time} = \text{latency} \) (i.e., time to send message of length zero)
- \( \beta = \text{incremental transfer time} \) per word (\( \frac{1}{\beta} = \text{bandwidth} \) in words per unit time)
- \( s = \text{length} \) 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 \textit{communication volume} (total amount of data sent)
- However, the execution time depends on whether we send the messages concurrently or in sequence
- The \textit{communication time} models execution time in terms of per-message costs
  - if the messages are sent \textit{simultaneously},
    \[
    T_{\text{sim-ring}}(s) = T_{\text{msg}}(s) = \alpha + s \cdot \beta
    \]
  - if the messages are sent \textit{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.
- *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.
Most regular network topologies admit simple routing schemes that are static, deterministic, and minimal.
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

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

**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

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

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

\[ T_{\text{route}} = \alpha D + \beta s \]
Store-and-Forward vs. Wormhole 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 the same communication link
- send message on one link while receiving simultaneously on a 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: 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

- **Motivation**
- **Architectures**
- **Networks**
- **Communication**
  - Message Routing
  - Communication Concurrency
  - Collective Communication
**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: \( T = (p - 1)(\alpha + \beta s) \)
- 2-D mesh: \( T = 2(\sqrt{p} - 1)(\alpha + \beta s) \)
- hypercube: \( T = \log p (\alpha + \beta s) \)

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
All collective-communication can be done near-optimally with \textit{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.

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

\[
T_{\text{broadcast}} = T_{\text{scatter}} + T_{\text{allgather}} = 2T_{\text{allgather}}
\]
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 Reduce-Scatter Diagram](image-url)
**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 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

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

Its possible to do All-to-All in less bandwidth cost (as low as \( \beta s \) by sending directly to targets) at the cost of more messages (as high as \( \alpha P \) 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
- 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 – Parallel Architectures

References – Interconnection Networks

- C.-L. Wu and T.-Y. Feng, eds., *Interconnection Networks for Parallel and Distributed Processing*, IEEE Computer Society Press, 1984
References – Hypercubes