Parallel Numerical Algorithms
Chapter 1 – Parallel Computing

Prof. Michael T. Heath

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
4. Communication
   - Message Routing
   - Communication Concurrency
   - Collective Communication
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
- 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

![Log of the number of components per integrated function vs. time](image)

**Microprocessor Transistor Counts 1971-2011 & Moore’s Law**

- **Fig. 2**: Number of components per integrated function for minimum cost per component extrapolated vs time.

- **Graph**: Transistor count doubling every two years.

- **Data Points**: Various microprocessors from 1971 to 2011.

- **Legend**: A list of processor names and manufacturers.

- **Axes**:
  - X-axis: Date of introduction
  - Y-axis: Transistor count

---

**Moore’s Law**

- **Motivation**: Improved performance and reduced cost of microprocessors.

- **Architectures**: Design strategies to leverage Moore’s Law.

- **Networks**: Communication systems in parallel computing.

---

Michael T. Heath

Parallel Numerical Algorithms
Smaller circuits are more efficient, so one can either

- maintain same power but increase clock speed (historical trend ~1975–2005)
- maintain same power and clock speed but increase functionality (current trend)
- maintain same clock speed but use less power (future trend?)
Consequences of Moore’s Law
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 more levels of cache)
- 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 parallel 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
  - 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 the 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

\[ \begin{align*}
M_0 & \quad M_1 & \cdots & \quad M_N \\
\quad P_0 & \quad P_1 & \cdots & \quad P_N \\
\text{network} & & & \text{network}
\end{align*} \]

distributed-memory multicomputer

shared-memory multiprocessor
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.

Topology of network affects algorithm design, implementation, and performance.
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.
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 width**: 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>2(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>2(n)</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>3(k/2)</td>
<td>(2k^2)</td>
<td>const</td>
</tr>
<tr>
<td>n-D torus</td>
<td>(k^n)</td>
<td>2(n)</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**: $\phi: V_s \rightarrow V_t$ 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
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

<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 exactly 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 elegant paradigm for low-diameter target network in designing parallel algorithms.
Simple model for time required to send message (move data) between adjacent nodes:

$$T_{\text{msg}} = t_s + t_w L$$

- $t_s = \text{startup time} = \text{latency}$ (i.e., time to send message of length zero)
- $t_w = \text{incremental transfer time}$ per word ($1/t_w = \text{bandwidth}$ in words per unit time)
- $L = \text{length}$ of message in words

For most real parallel systems, $t_s \gg t_w$
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 routing: entire message is received and stored at each node before being forwarded to next node on path, so

\[ T_{\text{msg}} = (t_s + t_w L)D, \text{ where } D = \text{distance in hops} \]

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

\[ T_{\text{msg}} = t_s + t_w L + t_h D, \text{ where } t_h = \text{incremental time per hop} \]
Cut-through routing greatly reduces distance effect, but aggregate bandwidth may still be significant constraint.
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

Depending on concurrency supported, time required for each step of communication algorithm is effectively multiplied by appropriate factor (e.g., degree of network graph)
Communication Concurrency

When multiple messages contend for network bandwidth, time required to send message modeled by

\[ T_{\text{msg}} = t_s + t_w S L \]

where \( S \) is number of messages sent concurrently over same communication link

In effect, each message uses \( 1/S \) of available bandwidth
Collective communication: multiple nodes communicating simultaneously in systematic pattern, such as

- broadcast: one-to-all
- reduction: all-to-one
- multinode broadcast: all-to-all
- scatter/gather: one-to-all/all-to-one
- total or complete exchange: personalized all-to-all
- scan or prefix
- circular shift
- barrier
Broadcast

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

Generic broadcast algorithm generates *spanning tree* with source node as root

```plaintext
if source \(\neq\) me then
    receive message
end

for each neighbor
    if neighbor has not already received message then
        send message to neighbor
    end
end
```
Broadcast

2-D mesh

hypercube
Broadcast

Cost of broadcast depends on network, for example

- 1-D mesh: $T = (p - 1) (t_s + t_w L)$
- 2-D mesh: $T = 2(\sqrt{p} - 1) (t_s + t_w L)$
- hypercube: $T = \log p (t_s + t_w L)$

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
**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.

Generic reduction algorithm uses spanning tree, but with data flow in opposite direction, from leaves to root.

```plaintext
for each child of mine in spanning tree
    receive value from child
    my_value = my_value $\oplus$ value
end
if root $\neq$ me then
    send my_value to parent
end
```
Reduction

2-D mesh

hypercube
Subsequent broadcast required if all nodes need result of reduction

Cost of reduction depends on network, for example

- 1-D mesh: \( T = (p - 1) (t_s + (t_w + t_c) L) \)
- 2-D mesh: \( T = 2(\sqrt{p} - 1) (t_s + (t_w + t_c) L) \)
- hypercube: \( T = \log p (t_s + (t_w + t_c) L) \)

where \( t_c = \text{time per word for associative reduction operation} \)

Often \( t_c \ll t_w \), so \( t_c \) is sometimes omitted from performance analyses
Multinode broadcast: each of \( p \) nodes sends message to all other nodes (all-to-all)

- Logically equivalent to \( p \) broadcasts, one from each node, but efficiency can often be enhanced by overlapping broadcasts
- Total time for multinode broadcast depends strongly on concurrency supported by communication system
- Multinode broadcast need be no more costly than standard broadcast if aggressive overlapping of communication is supported
Multinode Broadcast

Implementation of multinode broadcast in specific networks

- 1-D torus (ring): initiate broadcast from each node simultaneously in same direction around ring; completes after $p - 1$ steps at same cost as single-node broadcast
- 2-D or 3-D torus: apply ring algorithm successively in each dimension
- hypercube: exchange messages pairwise in each of $\log p$ dimensions, with messages concatenated at each stage

Multinode broadcast implements reduction if messages are combined using associative operation instead of concatenation; avoids subsequent broadcast when result needed by all nodes
Multinode reduction: each of $p$ nodes is destination of reduction from all other nodes

Algorithms for multinode reduction are essentially reverse of corresponding algorithms for multinode broadcast
**Personalized collective communication**: each node sends (or receives) *distinct* message to (or from) each other node

**Examples**

- **scatter**: analogous to broadcast, but root sends *different* message to each other node
- **gather**: analogous to reduction, but data received by root are concatenated rather than combined using associative operation
- **total exchange**: analogous to multinode broadcast, but each node exchanges *different* message with each other node
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 $s_0, s_1, \ldots, s_{p-1}$, where

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

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

Scan can be implemented similarly to multinode broadcast, except intermediate results received by each node are selectively combined, depending on sending node’s numbering, before being forwarded.
Circular \textit{k-shift}: for $0 < k < p$, node $i$ sends data to node $(i + k) \mod p$

Circular shift implemented naturally in ring network, and by embedding ring in other networks
**Barrier**: synchronization point that all processes must reach before any process is allowed to proceed beyond it

- For distributed-memory systems, barrier usually implemented by message passing, using algorithm similar to all-to-all
  - Some systems have special networks for fast barriers

- For shared-memory systems, barrier usually implemented using mechanism for enforcing *mutual exclusion*, such as test-and-set or semaphore, or with atomic memory operations
Graphical Depiction of Collective Communication

**broadcast**

```
0 ← 1 → 2 ← 3 → 4 → 5 → 6 → 7 → 8
```

**reduction**

```
0 → 1 → 2 → 3 → 4 ← 5 → 6 ← 7 ← 8
```

**multinode broadcast**

```
0 ← 1 ← 2 ← 3 ← 4 ← 5 ← 6 ← 7 ← 8
```

Arrows indicate general direction of data flow, but not necessarily specific implementation.

May not be convenient (or possible) to indicate root, in which case two-headed arrows used.
References – Moore’s Law


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