Part III
Mesh-Based Architectures

<table>
<thead>
<tr>
<th>Part I: Fundamental Concepts</th>
<th>Background and Motivation</th>
<th>1. Introduction to Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complexity and Models</td>
<td>2. A Taste of Parallel Algorithms</td>
</tr>
<tr>
<td>Part II: Extreme Models</td>
<td>Abstract View of Shared Memory</td>
<td>3. Parallel Algorithm Complexity</td>
</tr>
<tr>
<td></td>
<td>Circuit Model of Parallel Systems</td>
<td>4. Models of Parallel Processing</td>
</tr>
<tr>
<td>Part III: Mesh-Based Architectures</td>
<td>Data Movement on 2D Arrays</td>
<td>5. PRAM and Basic Algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7. Sorting and Selection Networks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8. Other Circuit-Level Examples</td>
</tr>
<tr>
<td>Part IV: Low-Diameter Architectures</td>
<td>The Hypercube Architecture</td>
<td>9. Sorting on a 2D Mesh or Torus</td>
</tr>
<tr>
<td></td>
<td>Hypercubic and Other Networks</td>
<td>10. Routing on a 2D Mesh or Torus</td>
</tr>
<tr>
<td>Part V: Some Broad Topics</td>
<td>Coordination and Data Access</td>
<td>11. Numerical 2D Mesh Algorithms</td>
</tr>
<tr>
<td></td>
<td>Robustness and Ease of Use</td>
<td>12. Other Mesh-Related Architectures</td>
</tr>
<tr>
<td></td>
<td>Data Parallelism and Conclusion</td>
<td>14. Sorting and Routing on Hypercubes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15. Other Hypercubic Architectures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16. A Sampler of Other Networks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17. Emulation and Scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18. Data Storage, Input, and Output</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19. Reliable Parallel Processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20. System and Software Issues</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21. Shared-Memory MIMD Machines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22. Message-Passing MIMD Machines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23. Data-Parallel SIMD Machines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24. Past, Present, and Future</td>
</tr>
</tbody>
</table>
About This Presentation

This presentation is intended to support the use of the textbook *Introduction to Parallel Processing: Algorithms and Architectures* (Plenum Press, 1999, ISBN 0-306-45970-1). It was prepared by the author in connection with teaching the graduate-level course ECE 254B: Advanced Computer Architecture: Parallel Processing, at the University of California, Santa Barbara. Instructors can use these slides in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Spring 2005</td>
<td>Spring 2006</td>
<td>Fall 2008</td>
<td>Fall 2010</td>
</tr>
<tr>
<td></td>
<td>Winter 2019</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III Mesh-Type Architectures

Study mesh, torus, and related interconnection schemes:
• Many modern parallel machines are mesh/torus-based
• Scalability and speed due to short, regular wiring
• Enhanced meshes, variants, and derivative networks

Topics in This Part

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Sorting on a 2D Mesh or Torus</td>
</tr>
<tr>
<td>10</td>
<td>Routing on a 2D Mesh or Torus</td>
</tr>
<tr>
<td>11</td>
<td>Numerical 2D Mesh Algorithms</td>
</tr>
<tr>
<td>12</td>
<td>Mesh-Related Architectures</td>
</tr>
</tbody>
</table>
9 Sorting on a 2D Mesh or Torus

Introduce the mesh model (processors, links, communication):
  • Develop 2D mesh sorting algorithms
  • Learn about strengths and weaknesses of 2D meshes

Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Mesh-Connected Computers</td>
</tr>
<tr>
<td>9.2</td>
<td>The Shearsort Algorithm</td>
</tr>
<tr>
<td>9.3</td>
<td>Variants of Simple Shearsort</td>
</tr>
<tr>
<td>9.4</td>
<td>Recursive Sorting Algorithms</td>
</tr>
<tr>
<td>9.5</td>
<td>A Nontrivial Lower Bound</td>
</tr>
<tr>
<td>9.6</td>
<td>Achieving the Lower Bound</td>
</tr>
</tbody>
</table>
9.1 Mesh-Connected Computers

2D, four-neighbor (NEWS) mesh; other types in Chapter 12

Square $p^{1/2} \times p^{1/2}$ or rectangular $r \times p/r$

MIMD, SPMD, SIMD, Weak SIMD

Single/All-port model

Diameter-based or bisection-based lower bound: $O(p^{1/2})$

Fig. 9.1 Two-dimensional mesh-connected computer.
MIMD, SPMD, SIMD, or Weak SIMD Mesh

Some communication modes.

a. MIMD all-port

All-port: Processor can communicate with all its neighbors at once (in one cycle or time step)

Single-port: Processor can send/receive one message per time step

b. MIMD Single-port

MIMD: Processors choose their communication directions independently

SIMD: All processors directed to do the same

c. SIMD single-port

Weak SIMD: Same direction for all (uniaxis)

d. Weak SIMD
Fig. 9.2  A $5 \times 5$ torus folded along its columns. Folding this diagram along the rows will produce a layout with only short links.
**Processor Indexing in Mesh or Torus**

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td></td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

a. Row-major

b. Snakelike row-major

c. Shuffled row-major

d. Proximity order

---
---
---
---

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td></td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td></td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>12</td>
<td>13</td>
<td></td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td></td>
<td>15</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td></td>
<td>14</td>
<td>13</td>
<td>10</td>
<td>9</td>
<td></td>
<td>14</td>
<td>13</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

c. Shuffled row-major

d. Proximity order

---
---
---
---

Our focus will be on row-major and snakelike row-major indexing

**Fig. 9.3** Some linear indexing schemes for the processors in a 2D mesh.
Register-Based Communication

Fig. 9.4  Reading data from NEWS neighbors via virtual local registers.
9.2 The Shearsort Algorithm

Shearsort algorithm for a 2D mesh with r rows

\[
T_{\text{shearsort}} = \left[ \log_2 r \right] \left( \frac{p}{r} + r \right) + \frac{p}{r}
\]

On a square mesh:
\[
T_{\text{shearsort}} = p^{1/2} \left( \log_2 p + 1 \right)
\]

Diameter-based LB:
\[
T_{\text{sort}} \geq 2p^{1/2} - 2
\]

repeat \[ \left\lfloor \log_2 r \right\rfloor \] times

Sort the rows (snake-like)

\[ \begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\end{array} \]

then sort the columns (top-to-bottom)

\[ \begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\end{array} \]

endrepeat

Sort the rows

Snakelike or Row-Major

(depending on the desired final sorted order)

Fig. 9.5 Description of the shearsort algorithm on an r-row 2D mesh.
Proving Shearsort Correct

Fig. 9.6 A pair of dirty rows create at least one clean row in each shearsort iteration
Shearsort Proof (Continued)

Fig. 9.7 The number of dirty rows halves with each shearsort iteration.

At most \([x/2]\) dirty rows

After \(\log_2 r\) iterations, only one dirty row remains
Shearsort Example

Fig. 9.8 Example of shearsort on a $4 \times 4$ mesh.
9.3 Variants of Simple Shearsort

Observation: On a linear array, odd-even transposition sort needs only $k$ steps if the “dirty” (unsorted) part of the array is of length $k$

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

Unsorted part

In shearsort, we do not have to sort columns completely, because only a portion of the column is unsorted (the portion shrinks in each phase)

\[T_{\text{opt shearsort}} = \frac{p}{r}\left(\left\lfloor \log_2 r \right\rfloor + 1\right) + \left(\frac{r}{2} + \frac{r}{4} + \ldots + r\right) = 2r - 2\]

Thus, $2r - 2$ replaces $r \log_2 r$ in simple shearsort

On a square mesh:

\[T_{\text{opt shearsort}} = \frac{p^{1/2}}{r}\left(\frac{1}{2} \log_2 r + 3\right) - 2\]
Shearsort with Multiple Items per Processor

Perform ordinary shearsort, but replace compare-exchange with merge-split \((n/p) \log_2(n/p)\) steps for the initial sort; the rest multiplied by \(n/p\).

Fig. 9.9  Example of shearsort on a 4 \(\times\) 4 mesh with two keys stored per processor.

**Keys**

<table>
<thead>
<tr>
<th>1</th>
<th>12</th>
<th>21</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>26</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>19</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>28</td>
<td>23</td>
<td>29</td>
<td>24</td>
</tr>
</tbody>
</table>

The final row sort (snake-like or row-major) is not shown.
### 9.4 Recursive Sorting Algorithms

**Snakelike sorting order on a square mesh**

\[ T(p^{1/2}) = T(p^{1/2}/2) + 5.5p^{1/2} \]

Note that row sort in phase 2 needs fewer steps.

**Snakelike sorting order on a square mesh**

\[ T_{\text{recursive}} \approx 11p^{1/2} \]

**Fig. 9.10** Graphical depiction of the first recursive algorithm for sorting on a 2D mesh based on four-way divide and conquer.
Proof of the $11p^{1/2}$-Time Sorting Algorithm

The proof of the first recursive sorting algorithm for 2D meshes.

$x \geq b + c + \lfloor (a - b)/2 \rfloor + \lfloor (d - c)/2 \rfloor$

A similar inequality applies to $x'$

\[
x + x' \geq b + c + \lfloor (a - b)/2 \rfloor + \lfloor (d - c)/2 \rfloor + a' + d' + \lfloor (b' - a')/2 \rfloor + \lfloor (c' - d')/2 \rfloor \\
\geq b + c + a' + d' + (a - b)/2 + (d - c)/2 + (b' - a')/2 + (c' - d')/2 - 4 \times 1/2 \\
= (a + a')/2 + (b + b')/2 + (c + c')/2 + (d + d')/2 - 2 \\
\geq p^{1/2} - 4
\]
Let $b$ (a power of 2) be the block length for snakelike sorting

\[ \text{snakelike-mesh-sort}(b) \]
\[ \text{snakelike-mesh-sort}(b/2) \]
\[ \text{snakelike-row-sort}(b) \]
\[ \text{column-sort}(b) \]
\[ \text{snake-odd-even-xpose}(4b) \]

\[ \text{snakelike-row-sort}(b) \]
for $k = 0$ to $b - 1$ Proc $(i, j)$, $j$ even, do

\[ \begin{align*}
\text{case } i, k & \\
\text{even, even: if } j \neq 0 \text{ mod } b \text{ AND } (R5) < (R3) \text{ then } R5 \leftrightarrow R3 \\
\text{even, odd: if } (R2) < (R5) \text{ then } R2 \leftrightarrow R5 \\
\text{. . . . . . . . . .} & 
\end{align*} \]
Another Recursive Sorting Algorithm

1. Sort quadrants
2. Shuffle row elements
3. Sort double columns in snakelike order
4. Apply $2\sqrt{p}$ steps of odd-even transposition along the overall snake

Distribute these $\sqrt{p}/2$ columns evenly

$$T(p^{1/2}) = T(p^{1/2}/2) + 4.5p^{1/2}$$

Note that the distribution in phase 2 needs $\frac{1}{2}p^{1/2}$ steps

$$T_{\text{recursive 2}} \approx 9p^{1/2}$$
Proof of the $9p^{1/2}$-Time Sorting Algorithm

Fig. 9.13 The proof of the second recursive sorting algorithm for 2D meshes.
Our Progress in Mesh Sorting Thus Far

**Lower bounds**: Theoretical arguments based on bisection width, and the like

**Upper bounds**: Deriving/analyzing algorithms and proving them correct

---

**Schnorr/Shamir algorithm**

- 2\(p^{1/2}\) Diameter
- 3\(p^{1/2}\) For snakelike order only (proved next)

**Problem 9.9**

- 9\(p^{1/2}\)
- 11\(p^{1/2}\)
- \(p^{1/2} \log_2 p\) Shearsort

---

Winter 2019  Parallel Processing, Mesh-Based Architectures  Slide 21
9.5 A Nontrivial Lower Bound

Fig. 9.14 The proof of the $3p^{1/2} - o(p^{1/2})$ lower bound for sorting in snakelike row-major order.

The proof is complete if we show that the highlighted element must move by $p^{1/2}$ steps in some cases.

Shortest path from the upper left triangle to the opposite corner in hops:

$2p^{1/2} - 2p^{1/4} - 2$

$x[t]$: Value held in this corner after $t$ steps.
Proving the Lower Bound

Any of the values 1-63 can be forced into any desired column in sorted order by mixing 0s and 64s in the shaded area.

Fig. 9.15  Illustrating the effect of fewer or more 0s in the shaded area.
Proving the Lower Bound

Any of the values 1-63 can be forced into any desired column in sorted order by mixing 0s and 64s in the shaded area.

Fig. 9.15 (Alternate version) Illustrating the effect of fewer or more 0s in the shaded area.
9.6 Achieving the Lower Bound

Schnorr-Shamir snakelike sorting

1. Sort each block in snakelike order
2. Permute columns such that the columns of each vertical slice are evenly distributed among all slices
3. Sort each block in snakelike order
4. Sort columns from top to bottom
5. Sort Blocks 0&1, 2&3, . . . of all vertical slices together in snakelike order; i.e., sort within $2p^{3/8} \times p^{3/8}$ submeshes
6. Sort Blocks 1&2, 3&4, . . . of all vertical slices together in snakelike order
7. Sort rows in snakelike order
8. Apply $2p^{3/8}$ steps of odd-even transposition to the snake

Fig. 9.16 Notation for the asymptotically optimal sorting algorithm.
Elaboration on the $3p^{1/2}$ Lower Bound

In deriving the $3p^{1/2}$ lower bound for snakelike sorting on a square mesh, we implicitly assumed that each processor holds one item at all times.

Without this assumption, the following algorithm leads to a running time of about $2.5p^{1/2}$.

**Phase 1:** Move all data to the center $p^{1/2}/2$ columns.

**Phase 2:** Perform 2-2 sorting in the half-wide center mesh.

**Phase 3:** Distribute data from center half of each row to the entire row.

$p^{1/2}/2$

$p^{1/2}/4$

$2p^{1/2}$

$p^{1/2}/4$
Routing is nonexistent in PRAM, hardwired in circuit model:
- Study point-to-point and collective communication
- Learn how to route multiple data packets to destinations

### Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Types of Data Routing Operations</td>
</tr>
<tr>
<td>10.2</td>
<td>Useful Elementary Operations</td>
</tr>
<tr>
<td>10.3</td>
<td>Data Routing on a 2D Array</td>
</tr>
<tr>
<td>10.4</td>
<td>Greedy Routing Algorithms</td>
</tr>
<tr>
<td>10.5</td>
<td>Other Classes of Routing Algorithms</td>
</tr>
<tr>
<td>10.6</td>
<td>Wormhole Routing</td>
</tr>
</tbody>
</table>
10.1 Types of Data Routing Operations

Point-to-point communication: one source, one destination

Collective communication

One-to-many: multicast, broadcast (one-to-all), scatter
Many-to-one: combine (fan-in), global combine, gather
Many-to-many: all-to-all broadcast (gossiping), scatter-gather
Types of Data Routing Algorithms

**Oblivious:** A source-destination pair leads to a unique path; non-fault-tolerant

**Adaptive:** One of the available paths is chosen dynamically; can avoid faulty nodes/links or route around congested areas

Degree of adaptivity leads to trade-offs between decision simplicity (e.g., hard to avoid infinite loops) and routing flexibility

**Optimal** (shortest-path): Only shortest paths considered; can be oblivious or adaptive

**Non-optimal** (non-shortest-path): Selection of shortest path is not guaranteed, although most algorithms tend to choose a shortest path if possible
Our First Encounter with Data Routing Issues

**Shared memory:** Processors can communicate by storing data into and reading data from the memory

**Circuit model:** Sending results from one part of the system to other parts is hardwired at design time

**Graph model:** We must specify the routing process explicitly
1-to-1 Communication (Point-to-Point Messages)

Packet sources:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
</table>

Packet destinations:

<table>
<thead>
<tr>
<th>d</th>
<th>c</th>
<th>f</th>
<th>e</th>
<th>a</th>
<th>h</th>
<th>b</th>
<th>g</th>
</tr>
</thead>
</table>

Routing paths:

Message sources, destinations, and routes:

Source nodes:

Destination nodes:
Routing Operations Specific to Meshes

Data compaction or packing
Move scattered data elements to the smallest possible submesh (e.g., for problem size reduction)

Random-access write (RAW)
Emulates one write step in PRAM (EREW vs CRCW)
Routing algorithm is critical

Random-access read (RAR)
Can be performed as two RAWs: Write source addresses to destinations; write data back to sources (emulates on PRAM memory read step)

Fig. 10.1 Example of data compaction or packing.
10.2 Useful Elementary Operations

Row/Column rotation
All-to-all broadcasting in a row or column

Semigroup computation

Fig. 10.2 Recursive semigroup computation in a 2D mesh.

Parallel prefix computation

Fig. 10.3 Recursive parallel prefix computation in a 2D mesh.

Sorting in various orders
Chapter 9
Routing on a Linear Array (Mesh Row or Column)

Fig. 10.4 Example of routing multiple packets on a linear array.
10.3 Data Routing on a 2D Array

Exclusive random-access write on a 2D mesh: *MeshRAW*

1. Sort packets in column-major order by destination column number; break ties by destination row number

2. Shift packets to the right, so that each item is in the correct column (no conflict; at most one element in a row headed for a given column)

3. Route the packets within each column

---

**Fig. 10.5** Example of random-access write on a 2D mesh.
Analysis of Sorting-Based Routing Algorithm

\[ T = 3p^{1/2} + o(p^{1/2}) \]
\[ + p^{1/2} \]
\[ + 2p^{1/2} - 2 \]
\[ = 6p^{1/2} + o(p^{1/2}) \]
\[ = 11p^{1/2} + o(p^{1/2}) \]

Not a shortest-path algorithm

Node buffer space requirement: 1 item at any given time
10.4 Greedy Routing Algorithms

**Greedy algorithm:** In each step, try to make the most progress toward the solution based on current conditions or information available.

This local or short-term optimization often does not lead to a globally optimal solution; but, problems with optimal greedy algorithms do exist.

![Diagram showing greedy row-first routing on a 2D mesh.](image)
Analysis of Row-First Greedy Routing

\[ T = 2p^{1/2} - 2 \]

This optimal time achieved if we give priority to messages that need to go further along a column.

Thus far, we have two mesh routing algorithms:

- 6\(p^{1/2}\)-step, 1 buffer per node
- 2\(p^{1/2}\)-step, time-optimal, but needs large buffers

**Question:** Is there a middle ground?

---

**Fig. 10.7** Demonstrating the worst-case buffer requirement with row-first routing.
An Intermediate Routing Algorithm

Sort \((p^{1/2}/q) \times (p^{1/2}/q)\)
submeshes in
column-major order

Perform greedy routing

Let there be \(r_k\) packets in
\(B_k\) headed for column \(j\)

Number of row-\(i\) packets
headed for column \(j\):

\[
\sum_{k=0}^{q-1} \left\lfloor \frac{r_k}{(p^{1/2}/q)} \right\rfloor
< \sum \left[ 1 + \frac{r_k}{(p^{1/2}/q)} \right]
\leq q + \left( \frac{q}{p^{1/2}} \right) \sum r_k
\leq 2q
\]

So, \(2q - 1\) buffers suffice

Fig. 10.8 Illustrating the structure of
the intermediate routing algorithm.
Analysis of the Intermediate Algorithm

Buffers: $2q - 1$, Intermediate between 1 and $O(p^{1/2})$

Sort time: $4p^{1/2}/q + o(p^{1/2}/q)$

Routing time: $2p^{1/2}$

Total time: $\approx 2p^{1/2} + 4p^{1/2}/q$

One extreme, $q = 1$:
Degenerates into sorting-based routing

Another extreme, large $q$:
Approaches the greedy routing algorithm

Fig. 10.8 Illustrating the structure of the intermediate routing algorithm.
10.5 Other Classes of Routing Algorithms

Row-first greedy routing has very good average-case performance, even if the node buffer size is restricted.

**Idea:** Convert any routing problem to two random instances by picking a random intermediate node for each message.

Regardless of the routing algorithm used, concurrent writes can degrade the performance.

Priority or combining scheme can be built into the routing algorithm so that congestion close to the common destination is avoided.

**Fig. 10.9** Combining of write requests headed for the same destination.
Types of Routing Problems or Algorithms

Static: Packets to be routed all available at $t = 0$
Dynamic: Packets “born” in the course of computation
Off-line: Routes precomputed, stored in tables
On-line: Routing decisions made on the fly
Oblivious: Path depends only on source and destination
Adaptive: Path may vary by link and node conditions
Deflection: Any received packet leaves immediately, even if this means misrouting (via detour path); also known as hot-potato routing
10.6 Wormhole Routing

**Circuit switching:** A circuit is established between source and destination before message is sent (as in old telephone networks)

- Advantage: Fast transmission after the initial overhead

**Packet switching:** Packets are sent independently over possibly different paths

- Advantage: Efficient use of channels due to sharing

**Wormhole switching:** Combines the advantages of circuit and packet switching

---

Fig. 10.10 Worms and deadlock in wormhole routing.
Route Selection in Wormhole Switching

Routing algorithm must be simple to make the route selection quick.

Example: row-first routing, with 2-byte header for row & column offsets

But ... care must be taken to avoid excessive blocking and deadlock.

(a) Two worms en route to their respective destinations

(b) Deadlock due to circular waiting of four blocked worms

Each worm is blocked at the point of attempted right turn.
Dealing with Conflicts

Buffer

Block

Drop

Deflect

Fig. 10.11 Various ways of dealing with conflicts in wormhole routing.
Deadlock in Wormhole Switching

Each worm is blocked at the point of attempted right turn

Two strategies for dealing with deadlocks:
1. Avoidance
2. Detection and recovery

Deadlock avoidance requires a more complicated routing algorithm and/or more conservative routing decisions

... nontrivial performance penalties
Deadlock Avoidance via Dependence Analysis

A sufficient condition for lack of deadlocks is to have a link dependence graph that is cycle-free.

Less restrictive models are also possible; e.g., the turn model allows three of four possible turns for each worm.

Fig. 10.12 Use of dependence graph to check for the possibility of deadlock.
Deadlock Avoidance via Virtual Channels

Fig. 10.13 Use of virtual channels for avoiding deadlocks.

Deadlock Avoidance via Routing Restrictions

Allow only three of the four possible turns
11 Numerical 2D Mesh Algorithms

Become more familiar with mesh/torus architectures by:
- Developing a number of useful numerical algorithms
- Studying seminumerical applications (graphs, images)

### Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Matrix Multiplication</td>
</tr>
<tr>
<td>11.2</td>
<td>Triangular System of Linear Equations</td>
</tr>
<tr>
<td>11.3</td>
<td>Tridiagonal System of Linear Equations</td>
</tr>
<tr>
<td>11.4</td>
<td>Arbitrary System of Linear Equations</td>
</tr>
<tr>
<td>11.5</td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>11.6</td>
<td>Image-Processing Algorithms</td>
</tr>
</tbody>
</table>
11.1 Matrix Multiplication

\[ y = Ax \quad \text{or} \]
\[ y_i = \sum_{j=0}^{m-1} a_{ij} x_j \]

With \( p = m \) processors,
\[ T = 2m - 1 = 2p - 1 \]

Fig. 11.1 Matrix–vector multiplication on a linear array.
Another View of Matrix-Vector Multiplication

$m$-processor linear array for multiplying an $m$-vector by an $m \times m$ matrix.
**Mesh Matrix Multiplication**

**Fig. 11.2** Matrix-matrix multiplication on a 2D mesh.

\[ C = AB \quad \text{or} \quad c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj} \]

\[ p = m^2, \quad T = 3m - 2 \]
Matrix-Vector Multiplication on a Ring

\[ y = Ax \quad \text{or} \quad y_i = \sum_{j=0}^{m-1} a_{ij} x_j \]

With \( p = m \) processors, \( T = m = p \)

Fig. 11.3  Matrix-vector multiplication on a ring.
Torus Matrix Multiplication

\[ C = AB \text{ or } c_{ij} = \sum_{k=0}^{m-1} a_{ik} b_{kj} \]

For \( m > \rho^{1/2} \), use block matrix multiplication

Can gain efficiency from overlapping communication with computation
11.2 Triangular System of Linear Equations

Solution: Use forward (lower) or back (upper) substitution

\[
\begin{align*}
    a_{00}x_0 &= b_0 \\
    a_{10}x_0 + a_{11}x_1 &= b_1 \\
    a_{20}x_0 + a_{21}x_1 + a_{22}x_2 &= b_2 \\
    \vdots & \vdots \\
    a_{m-1,0}x_0 + a_{m-1,1}x_1 + \ldots + a_{m-1,m-1}x_{m-1} &= b_{m-1}
\end{align*}
\]

Lower triangular: Find \(x_0\) from the first equation, substitute in the second equation to find \(x_1\), etc.

Fig. 11.5 Lower/upper triangular square matrix; if \(a_{ii} = 0\) for all \(i\), then it is strictly lower/upper triangular.
Forward Substitution on a Linear Array

\[ a_{00}x_0 = b_0 \]
\[ a_{10}x_0 + a_{11}x_1 = b_1 \]
\[ \vdots \]
\[ a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \]

\[ b_0 \]
\[ x_0 \]
\[ x_1 \]
\[ x_2 \]
\[ x_3 \]

\[ \text{Inputs} \]
\[ \text{Outputs} \]

Col 0 of \( A \)

\[ b - ax \]
\[ x \]

\[ x_0 \]
\[ x_1 \]
\[ x_2 \]
\[ x_3 \]

\[ x_0 \]
\[ x_1 \]
\[ x_2 \]
\[ x_3 \]

Fig. 11.6 Solving a triangular system of linear equations on a linear array.
Triangular Matrix Inversion: Algorithm

\[
\begin{pmatrix}
0 \\
\vdots \\
a_{ij} \\
i \geq j
\end{pmatrix} \times
\begin{pmatrix}
\vdots \\
\vdots \\
1 \\
0
\end{pmatrix} =
\begin{pmatrix}
1 \\
\vdots \\
1 \\
0
\end{pmatrix}
\]

A multiplied by \(i\)th column of \(X\) yields \(i\)th column of the identity matrix \(I\)
(solve \(m\) such triangular systems to invert \(A\))

Fig. 11.7 Inverting a triangular matrix by solving triangular systems of linear equations.
Triangular Matrix Inversion on a Mesh

$T = 3m - 2$

Can invert two matrices using one more step

Fig. 11.8 Inverting a lower triangular matrix on a 2D mesh.
11.3 Tridiagonal System of Linear Equations

\[ l_0 x_{-1} + d_0 x_0 + u_0 x_1 = b_0 \]
\[ l_1 x_0 + d_1 x_1 + u_1 x_2 = b_1 \]
\[ l_2 x_1 + d_2 x_2 + u_2 x_3 = b_2 \]
\[ \ldots \]
\[ l_{m-1} x_{m-2} + d_{m-2} x_{m-1} + u_{m-2} x_m = b_{m-1} \]

Special case of a band matrix

\[
\begin{pmatrix}
  d_0 & u_0 & 0 & & \\
  l_1 & d_1 & u_1 & 0 & \\
  l_2 & d_2 & u_2 & \ddots & \\
  \vdots & \ddots & \ddots & \ddots & 0 \\
  0 & \ddots & \ddots & l_{m-2} & d_{m-2} & u_{m-2} \\
  l_{m-1} & d_{m-1} & u_{m-1} & \cdots & \cdots & l_m & d_m & u_m & \cdots & 0 \\
\end{pmatrix}
\begin{pmatrix}
  x_0 \\
  x_1 \\
  x_2 \\
  \vdots \\
  x_{m-1} \\
\end{pmatrix}
= 
\begin{pmatrix}
  b_0 \\
  b_1 \\
  b_2 \\
  \vdots \\
  b_{m-1} \\
\end{pmatrix}
\]

Fig. 11.9 A tridiagonal system of linear equations.
Other Types of Diagonal Matrices

Tridiagonal, pentadiagonal, . . . matrices arise in the solution of differential equations using finite difference methods.

Matrices with more than three diagonals can be viewed as tridiagonal blocked matrices.

A pentadiagonal matrix.
Odd-Even Reduction

\[ l_0 x_{-1} + d_0 x_0 + u_0 x_1 = b_0 \]
\[ l_1 x_0 + d_1 x_1 + u_1 x_2 = b_1 \]
\[ l_2 x_1 + d_2 x_2 + u_2 x_3 = b_2 \]
\[ l_3 x_2 + d_3 x_3 + u_3 x_4 = b_3 \]
\[
\]

Use odd equations to find odd-indexed variables in terms of even-indexed ones

\[ d_1 x_1 = b_1 - l_1 x_0 - u_1 x_2 \]
\[ d_3 x_3 = b_3 - l_3 x_2 - u_3 x_4 \]
\[
\]

Substitute in even equations to get a tridiagonal system of half the size

\[ L_0 x_{-2} + D_0 x_0 + U_0 x_2 = B_0 \]
\[ L_2 x_0 + D_2 x_2 + U_2 x_4 = B_2 \]
\[ L_4 x_2 + D_4 x_4 + U_4 x_6 = B_4 \]
\[
\]

The six divides are replaceable with one reciprocation per equation, to find \(1/d_j\) for odd \(j\), and six multiplies

\[ L_i = -l_i l_{i-1}/d_{i-1} \]
\[ D_i = d_i - l_i u_{i-1}/d_{i-1} - u_i l_{i+1}/d_{i+1} \]
\[ U_i = -u_i u_{i+1}/d_{i+1} \]
\[ B_i = b_i - l_i b_{i-1}/d_{i-1} - u_i b_{i+1}/d_{i+1} \]

Sequential solution:

\[ T(m) = T(m/2) + cm = 2cm \]
Architecture for Odd-Even Reduction

* Find $x_1$ in terms of $x_0$ and $x_2$ from Eqn. 1; substitute in Eqns. 0 and 2.

Parallel solution:

$$T(m) = T(m/2) + c = c \log_2 m$$

Because we ignored communication, our analysis is valid for PRAM or for an architecture whose topology matches that of Fig. 11.10.

Fig. 11.10  The structure of odd-even reduction for solving a tridiagonal system of equations.
Odd-Even Reduction on a Linear Array

Architecture of Fig. 11.10 can be modified to binary X-tree and then simplified to 2D multigrid

Communication time on linear array:

\[ T(m) = 2(1 + 2 + \ldots + m/2) = 2m - 2 \]
Odd-Even Reduction on a 2D Mesh

Communication time on 2D mesh:
\[ T(m) \approx 2[2(1 + 2 + \ldots + m^{1/2}/2)] \]
\[ \approx 2m^{1/2} \]
11.4 Arbitrary System of Linear Equations

\[ \begin{align*}
2x_0 + 4x_1 - 7x_2 &= 3 \\
3x_0 + 6x_1 - 10x_2 &= 4 \\
-x_0 + 3x_1 - 4x_2 &= 6
\end{align*} \]

Extended matrix \( A' = \begin{pmatrix} 2 & 4 & -7 \\ 3 & 6 & -10 \\ -1 & 3 & -4 \end{pmatrix} \)

Divide row 0 by 2; subtract 3 times from row 1 (pivoting oper)

Gaussian elimination

\[ \begin{align*}
Ax &= b \\
\begin{pmatrix} A \\ b \end{pmatrix} & \text{for system 1} \\
\begin{pmatrix} A \\ b \end{pmatrix} & \text{for system 2}
\end{align*} \]

Repeat until identity matrix appears in first \( n \) columns; read solutions from remaining columns

Extended matrix \( A' = \begin{pmatrix} 1 & 2 & -3.5 & 1.5 & 3.5 \\ 0 & 0 & 0.5 & -0.5 & -2.5 \\ 0 & 5 & -7.5 & 7.5 & 2.5 \end{pmatrix} \)

Extended matrix \( A' = \begin{pmatrix} 1 & 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 & -7 \\ 0 & 0 & 1 & -1 & -5 \end{pmatrix} \)
Performing One Step of Gaussian Elimination

Fig. 11.12 A linear array performing the first phase of Gaussian elimination.
Gaussian Elimination on a 2D Mesh

Fig. 11.13 Implementation of Gaussian elimination on a 2D array.
Matrix Inversion on a 2D Mesh

Fig. 11.14 Matrix inversion by Gaussian elimination.
Jacobi Methods

\[ Ax = b \]

\[
\begin{align*}
2x_0 + 4x_1 - 7x_2 &= 3 \\
3x_0 + 6x_1 - 10x_2 &= 4 \\
-x_0 + 3x_1 - 4x_2 &= 6
\end{align*}
\]

Solution: \( x_0 = -2 \) \hspace{1cm} x_1 = 0 \hspace{1cm} x_2 = -1

Use each equation to find one of the variables in terms of all others

\[
\begin{align*}
x_0 &= -2.000x_1 + 3.500x_2 + 1.500 \\
x_1 &= -0.500x_0 + 1.667x_2 + 0.667 \\
x_2 &= -0.250x_0 + 0.750x_1 - 1.500
\end{align*}
\]

Iterate: Plug in estimates for the unknowns on the right-hand side to find new estimates on the left-hand side

Example: Estimate \( x_0 = 1, \ x_1 = 1, \ x_2 = 1 \)

\[
\begin{align*}
x_0 &= -2.000 + 3.500 + 1.500 = 3.000 \\
x_1 &= -0.500 + 1.667 + 0.667 = 1.834 \\
x_2 &= -0.250 + 0.750 - 1.500 = -1.000
\end{align*}
\]
Jacobi Relaxation and Overrelaxation

**Jacobi relaxation:** Assuming $a_{ii} \neq 0$, solve the $i$th equation for $x_i$, yielding $m$ equations from which new (better) approximations to the answers can be obtained.

$$x_i^{(t+1)} = \frac{1}{a_{ii}} [b_i - \sum_{j \neq i} a_{ij} x_j^{(t)}] \quad x_i^{(0)} = \text{initial approximation for } x_i$$

On an $m$-processor linear array, each iteration takes $O(m)$ steps. The number of iterations needed is $O(\log m)$ if certain conditions are satisfied, leading to $O(m \log m)$ average time.

A variant: **Jacobi overrelaxation**

$$x_i^{(t+1)} = (1 - \gamma) x_i^{(t)} + \left( \frac{\gamma}{a_{ii}} \right) [b_i - \sum_{j \neq i} a_{ij} x_j^{(t)}] \quad 0 < \gamma \leq 1$$

For $\gamma = 1$, the method is the same as Jacobi relaxation.

For smaller $\gamma$, overrelaxation may offer better performance.
11.5 Graph Algorithms

Fig. 11.15  Matrix representation of directed graphs.

\[
A = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
0 & 2 & 2 & \infty & 2 \\
1 & 0 & 2 & \infty & \infty \\
\infty & \infty & 0 & -3 & \infty \\
\infty & \infty & \infty & 0 & 0 \\
1 & \infty & \infty & \infty & 0 \\
\end{bmatrix}
\]
Transitive Closure of a Graph

\[ A^0 = I \quad \text{Paths of length 0 (identity matrix)} \]
\[ A^1 = A \quad \text{Paths of length 1} \]
\[ A^2 = A \times A \quad \text{Paths of length 2} \]
\[ A^3 = A^2 \times A \quad \text{Paths of length 3 etc.} \]

Compute “powers” of \( A \) via matrix multiplication, but use AND/OR in lieu of multiplication/addition

Transitive closure of \( G \) has the adjacency matrix

\[ A^* = A^0 + A^1 + A^2 + \ldots \]

\( A_{ij}^* = 1 \) iff node \( j \) is reachable from node \( i \)

Powers need to be computed up to \( A^{n-1} \) (why?)
Transitive Closure Algorithm

Initialization: Insert the edges \((i, i), 0 \leq i \leq n - 1\), into the graph

Phase 0    Insert the edge \((i, j)\) into the graph if \((i, 0)\) and \((0, j)\) are in the graph

Phase 1    Insert the edge \((i, j)\) into the graph if \((i, 1)\) and \((1, j)\) are in the graph

\ldots

Phase \(k\)   Insert the edge \((i, j)\) into the graph if \((i, k)\) and \((k, j)\) are in the graph

[Graph \(A^{(k)}\) then has an edge \((i, j)\) iff there is a path from \(i\) to \(j\) that goes only through nodes \(\{1, 2, \ldots, k\}\) as intermediate hops]

\ldots

Phase \(n - 1\)   Graph \(A^{(n-1)}\) is the answer \(A^*\)

\[
A = \begin{pmatrix}
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

Graph \(G\) with adjacency matrix \(A\)
Transitive Closure on a 2D Mesh

The key to the algorithm is to ensure that each phase takes constant time; overall $O(n)$ steps. This would be optimal on an $n \times n$ mesh because the best sequential algorithm needs $O(n^3)$ time.

Graph $G$ with adjacency matrix $A$

Fig. 11.16  Transitive closure algorithm on a 2D mesh.
Elimination of Broadcasting via Retiming

Example of systolic retiming by delaying the inputs to $C_L$ and advancing the outputs from $C_L$ by $d$ units [Fig. 12.8 in Computer Arithmetic: Algorithms and Hardware Designs, by Parhami, Oxford, 2000]
Systolic Retiming for Transitive Closure

Add $2n - 2 = 6$ units of delay to edges crossing cut 1
Move 6 units of delay from inputs to outputs of node $(0, 0)$

Fig. 11.17
Systolic retiming to eliminate broadcasting.
11.6 Image Processing Algorithms

Labeling connected components in a binary image (matrix of pixels)

The reason for considering diagonally adjacent pixels parts of the same component.

Worst-case component showing that a naïve “propagation” algorithm may require $O(p)$ time.


Recursive Component Labeling on a 2D Mesh

<table>
<thead>
<tr>
<th>$C_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_{49}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 11.18  Connected components in an 8 $\times$ 8 binary image.

Fig. 11.19  Finding the connected components via divide and conquer.

$$T(p) = T(p/4) + O(p^{1/2}) = O(p^{1/2})$$
Levialdi’s Algorithm

0 1 1 1 0 is changed to 1
1 0 1 0 if N = W = 1
0 0 0 0 1 is changed to 0
1 0 1 0 if N = W = NW = 0

Figure 11.20
Transformation or rewriting rules for Levialdi’s algorithm in the shrinkage phase (no other pixel changes).

Figure 11.21 Example of the shrinkage phase of Levialdi’s component labeling algorithm.
Analysis and Proof of Levialdi’s Algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0 is changed to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>if N = W = 1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1 is changed to 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>if N = W = NW = 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11.20
Transformation or rewriting rules for Levialdi’s algorithm in the shrinkage phase (no other pixel changes).

Latency of Levialdi’s algorithm

\[ T(n) = 2n^{1/2} - 1 \text{ (shrinkage)} + 2n^{1/2} - 1 \text{ (expansion)} \]

Component do not merge in shrinkage phase
Consider a 0 that is about to become a 1
If any \( y \) is 1, then already connected
If \( z \) is 1 then it will change to 0 unless at least one neighboring \( y \) is 1
12 Mesh-Related Architectures

Study variants of simple mesh and torus architectures:
- Variants motivated by performance or cost factors
- Related architectures: pyramids and meshes of trees

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Three or More Dimensions</td>
</tr>
<tr>
<td>12.2 Stronger and Weaker Connectivities</td>
</tr>
<tr>
<td>12.3 Meshes Augmented with Nonlocal Links</td>
</tr>
<tr>
<td>12.4 Meshes with Dynamic Links</td>
</tr>
<tr>
<td>12.5 Pyramid and Multigrid Systems</td>
</tr>
<tr>
<td>12.6 Meshes of Trees</td>
</tr>
</tbody>
</table>
12.1 Three or More Dimensions

3D vs 2D mesh: \[ D = 3p^{1/3} - 3 \] vs \[ 2p^{1/2} - 2 \]; \[ B = p^{2/3} \] vs \[ p^{1/2} \]

Example: 3D \( 8 \times 8 \times 8 \) mesh \( p = 512, D = 21, B = 64 \)
2D \( 22 \times 23 \) mesh \( p = 506, D = 43, B = 23 \)

Fig. 12.1 3D and 2.5D physical realizations of a 3D mesh.
More than Three Dimensions?

2.5D and 3D packaging technologies

4D, 5D, . . . meshes/tori: optical links?

$qD$ mesh with $m$ processors along each dimension: $p = m^q$
- Node degree $d = 2q$
- Diameter $D = q(m - 1) = q(p^{1/q} - 1)$
- Bisection width: $B = p^{1-1/q}$ when $m = p^{1/q}$ is even

$qD$ torus with $m$ processors along each dimension = $m$-ary $q$-cube

(a) 2D or 2.5D packaging now common
(b) 3D packaging of the future
Node Indexing in $q$-D Meshes

<table>
<thead>
<tr>
<th>zyx order</th>
<th>Node Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>002</td>
<td>2</td>
</tr>
<tr>
<td>003</td>
<td>3</td>
</tr>
<tr>
<td>010</td>
<td>4</td>
</tr>
<tr>
<td>011</td>
<td>5</td>
</tr>
<tr>
<td>012</td>
<td>6</td>
</tr>
<tr>
<td>013</td>
<td>7</td>
</tr>
<tr>
<td>020</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>101</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>200</td>
<td>32</td>
</tr>
<tr>
<td>201</td>
<td>33</td>
</tr>
</tbody>
</table>
Sorting on a 3D Mesh

Time for Kunde’s algorithm
\[ = 4 \times (2D\text{-}sort\ time) + 2 \]
\[ \approx 16p^{1/3} \text{ steps} \]

Defining the zyx processor ordering

A variant of shearsort is available, but Kunde’s algorithm is faster and simpler

Sorting on 3D mesh (zyx order; reverse of node index)

Phase 1: Sort elements on each zx plane into zx order
Phase 2: Sort elements on each yz plane into zy order
Phase 3: Sort elements on each xy layer into yx order (odd layers sorted in reverse order)
Phase 4: Apply 2 steps of odd-even transposition along z
Phase 5: Sort elements on each xy layer into yx order
Greedy zyx (layer-first, row last) routing algorithm

Phase 1: Sort into zyx order by destination addresses
Phase 2: Route along z dimension to correct xy layer
Phase 3: Route along y dimension to correct column
Phase 4: Route along x dimension to destination

As in 2D case, partial sorting can be used

Simple greedy algorithm does fine usually, but sorting first reduces buffer requirements

Time for sort-based routing = Sort time + Diameter ≈ 19p^{1/3} steps
Matrix Multiplication on a 3D Mesh

A total of \((m^{1/4})^3 = m^{3/4}\) block multiplications are needed.

Matrix blocking for multiplication on a 3D mesh

Assume the use of an \(m^{3/4} \times m^{3/4} \times m^{3/4}\) mesh with \(p = m^{9/4}\) processors.

Each \(m^{3/4} \times m^{3/4}\) layer of the mesh is assigned to one of the \(m^{3/4} \times m^{3/4}\) matrix multiplications (\(m^{3/4}\) multiply-add steps).

The rest of the process can take time that is of lower order.

Optimal: Matches sequential work and diameter-based lower bound.
Low- vs High-Dimensional Meshes

There is a good match between the structure of a 3D mesh and communication requirements of physical modeling problems.

6 × 6 mesh emulating 3 × 3 × 3 mesh (not optimal)

<table>
<thead>
<tr>
<th>Layer</th>
<th>Middle layer</th>
<th>Upper layer</th>
<th>Lower layer</th>
</tr>
</thead>
</table>

A low-dimensional mesh can efficiently emulate a high-dimensional one.

**Question:** Is it more cost effective, e.g., to have 4-port processors in a 2D mesh architecture or 6-port processors in a 3D mesh architecture, given that for the 4-port processors, fewer ports and ease of layout allow us to make each channel wider?
12.2 Stronger and Weaker Connectivities

As in higher-dimensional meshes, greater connectivity does not automatically translate into greater performance.

Area and signal-propagation delay penalties must be factored in.
Simplification via Link Orientation

Two in- and out-channels per node, instead of four

With even side lengths, the diameter does not change

Some shortest paths become longer, however

Can be more cost-effective than 2D mesh

Figure 12.3  A $4 \times 4$ Manhattan street network.
Pruning a high-dimensional mesh or torus can yield an architecture with the same diameter but much lower implementation cost.
Simplification via Link Sharing

Fig. 12.5 Eight-neighbor mesh with shared links and example data paths.

Factor-of-2 reduction in ports and links, with no performance degradation for uniaxis communication (weak SIMD model)
12.3 Meshes Augmented with Nonlocal Links

Motivation: Reduce the wide diameter (which is a weakness of meshes)

Fig. 12.6 Three examples of bypass links along the rows of a 2D mesh.

Increases max node degree and hurts the wiring locality and regularity

Road analogy for bypass connections

One-way street

Freeway
Using a Single Global Bus

Fig. 12.7 Mesh with a global bus and semigroup computation on it.

Semigroup computation on 2D mesh with a global bus

Phase 1: Find partial results in $p^{1/3} \times p^{1/3}$ submeshes in $O(p^{1/3})$ steps; results stored in the upper left corner of each submesh.

Phase 2: Combine partial results in $O(p^{1/3})$ steps, using a sequential algorithm in one node and the global bus for data transfers.

Phase 3: Broadcast the result to all nodes (one step).

The single bus increases the bisection width by 1, so it does not help much with sorting or other tasks that need extensive data movement.
Mesh with Row and Column Buses

The bisection width doubles, so row and column buses do not fundamentally change the performance of sorting or other tasks that need extensive data movement.

**Fig. 12.8** Mesh with row/column buses and semigroup computation on it.

**Semigroup computation on 2D mesh with row and column buses**

- **Phase 1:** Find partial results in $p^{1/6} \times p^{1/6}$ submeshes in $O(p^{1/6})$ steps
- **Phase 2:** Distribute $p^{1/3}$ row values left among the $p^{1/6}$ rows in same slice
- **Phase 3:** Combine row values in $p^{1/6}$ steps using the row buses
- **Phase 4:** Distribute column-0 values to $p^{1/3}$ columns using the row buses
- **Phase 5:** Combine column values in $p^{1/6}$ steps using the column buses
- **Phase 6:** Distribute $p^{1/3}$ values on row 0 among $p^{1/6}$ rows of row slice 0
- **Phase 7:** Combine row values in $p^{1/6}$ steps
- **Phase 8:** Broadcast the result to all nodes (2 steps)
12.4 Meshes with Dynamic Links

Semigroup computation in $O(\log p)$ steps; both 1D and 2D meshes

Various subsets of processors (not just rows and columns) can be configured, to communicate over shared buses

Fig. 12.9 Linear array with a separable bus using reconfiguration switches.

Fig. 12.10 Some processor states in a reconfigurable mesh.
Programmable Connectivity in FPGAs

Interconnection switch with 8 ports and four connection choices for each port:

- 0 – No connection
- 1 – Straight through
- 2 – Right turn
- 3 – Left turn

8 control bits (why?)
An Array Reconfiguration Scheme

3-state $2 \times 2$ switch
Reconfiguration of Faulty Arrays

Question: How do we know which cells/nodes must be bypassed?

Must devise a scheme in which healthy nodes set the switches
12.5 Pyramid and Multigrid Systems

Faster than mesh for semigroup computation, but not for sorting or arbitrary routing

Fig. 12.11 Pyramid with 3 levels and $4 \times 4$ base along with its 2D layout.

Originally developed for image processing applications
Roughly $\frac{3}{4}$ of the processors belong to the base
For an $l$-level pyramid: $D = 2l - 2 \quad d = 9 \quad B = 2^l$
Pyramid and 2D Multigrid Architectures

Fig. 12.12 The relationship between pyramid and 2D multigrid architectures.

Multigrid architecture is less costly and can emulate the pyramid architecture quite efficiently.

Pyramid is to 2D multigrid what X-tree is to 1D multigrid.
12.6 Meshes of Trees

2m trees, each with m leaves, sharing leaves in the base

Row and column roots can be combined into m degree-4 nodes

Row tree (one per row)

Column tree (one per col.)

$m \times m$ base

Fig. 12.13 Mesh of trees architecture with 3 levels and a $4 \times 4$ base.
Alternate Views of a Mesh of Trees

Fig. 12.14 Alternate views of the mesh of trees architecture with a $4 \times 4$ base.

2D layout for mesh of trees network with a $4 \times 4$ base; root nodes are in the middle row and column.
Simple Algorithms for Mesh of Trees

Semigroup computation: row/column combining

Parallel prefix computation: similar

Routing \( m^2 \) packets, one per processor on the \( m \times m \) base: requires \( \Omega(m) = \Omega(p^{1/2}) \) steps

In the view of Fig. 12.14, with only \( m \) packets to be routed from one side of the network to the other, \( 2 \log_2 m \) steps are required, provided destination nodes are distinct

Sorting \( m^2 \) keys, one per processor on the \( m \times m \) base: emulate any mesh sorting algorithm

Sorting \( m \) keys stored in merged roots: broadcast \( x_i \) to row \( i \) and column \( i \), compare \( x_i \) to \( x_j \) in leaf \((i, j)\) to set a flag, add flags in column trees to find the rank of \( x_i \), route \( x_i \) to node \( rank[x_i] \)
Some Numerical Algorithms for Mesh of Trees

Matrix-vector multiplication $Ax = y$ (A stored on the base and vector $x$ in the column roots, say; result vector $y$ is obtained in the row roots): broadcast $x_j$ in the $j$th column tree, compute $a_{ij}x_j$ in base processor $(i, j)$, sum over row trees

Convolution of two vectors: similar
Minimal-Weight Spanning Tree Algorithm

Greedy algorithm: in each of at most $\log_2 n$ phases, add the minimal-weight edge that connects a component to a neighbor.

Sequential algorithms, for an $n$-node, $e$-edge graph:
- Kruskal’s: $O(e \log e)$
- Prim’s (binary heap): $O((e + n) \log n)$

Both of these algorithms are $O(n^2 \log n)$ for dense graphs, with $e = O(n^2)$.

Prim’s (Fibonacci heap): $O(e + n \log n)$, or $O(n^2)$ for dense graphs.

Fig. 12.16 Example for min-weight spanning tree algorithm.
MWST Algorithm on a Mesh of Trees

The key to parallel version of the algorithm is showing that each phase can be done in $O(\log^2 n)$ steps; $O(\log^3 n)$ overall.

Leaf $(i, j)$ holds the weight $W(i, j)$ of edge $(i, j)$ and “knows” whether the edge is in the spanning tree, and if so, in which supernode. In each phase, we must:

a. Find the min-weight edge incident to each supernode
b. Merge supernodes for next phase

Subphase a takes $O(\log n)$ steps
Subphase b takes $O(\log^2 n)$ steps