Outline

- Overview of some Serial Algorithms
- Parallel Algorithm vs Parallel Formulation
- Elements of a Parallel Algorithm/Formulation
  - concurrency extractor!
- Common Mapping Methods
  - parallel overhead reducer!
Some Serial Algorithms
Working Examples

- Dense Matrix-Matrix & Matrix-Vector Multiplication
- Sparse Matrix-Vector Multiplication
- Gaussian Elimination
- Floyd’s All-pairs Shortest Path
- Quicksort
- Minimum/Maximum Finding
- Heuristic Search—15-puzzle problem
Dense Matrix-Vector Multiplication

1. procedure MAT_VECT \((A, x, y)\)
2. begin
3.  \textbf{for} \(i\) := 0 \textbf{to} \(n - 1\) \textbf{do}
4.  begin
5.     \(y[i] := 0;\)
6.  \textbf{for} \(j\) := 0 \textbf{to} \(n - 1\) \textbf{do}
7.     \(y[i] := y[i] + A[i, j] \times x[j];\)
8.  endfor;
9. end MAT_VECT

\textbf{Algorithm 8.1} A serial algorithm for multiplying an \(n \times n\) matrix \(A\) with an \(n \times 1\) vector \(x\) to yield an \(n \times 1\) product vector \(y\).
Dense Matrix-Matrix Multiplication

1. procedure MAT_MULTI (A, B, C)
2. begin
3.  for i := 0 to n - 1 do
4.    for j := 0 to n - 1 do
5.      begin
6.        C[i, j] := 0;
7.        for k := 0 to n - 1 do
9.      endfor;
10.   end MAT_MULTI

Algorithm 8.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.
Sparse Matrix-Vector Multiplication

\[ y = Ab \]

\[ y[i] = \sum_{j=1}^{n} (A[i, j] \times b[j]) \]
Gaussian Elimination

\[ a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0, \]

\[ a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} = b_1, \]

\[ \vdots \]

\[ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}. \]

---

1. \textbf{procedure GAUSSIAN\_ELIMINATION} (\(A, b, y\))
2. \textbf{begin}
3. \hspace{1em} \textbf{for} \(k := 0 \text{ to } n - 1\) \textbf{do} /* Outer loop */
4. \hspace{2em} \textbf{begin}
5. \hspace{3em} \textbf{for} \(j := k + 1 \text{ to } n - 1\) \textbf{do}
7. \hspace{3em} \textbf{for} \(i := k + 1 \text{ to } n - 1\) \textbf{do}
8. \hspace{4em} \(A[i, k] := 1 ;
9. \hspace{3em} \textbf{begin}
10. \hspace{4em} \textbf{for} \(j := k + 1 \text{ to } n - 1\) \textbf{do}
12. \hspace{5em} \textbf{for} \(i := k + 1 \text{ to } n - 1\) \textbf{do}
13. \hspace{6em} \(b[i] := b[i] - A[i, k] \times y[k] ;
14. \hspace{5em} \textbf{endfor} /* Line 9 */
15. \hspace{4em} \textbf{endfor} /* Line 8 */
16. \hspace{3em} \textbf{endbegin} /* Line 7 */
17. \textbf{endend GAUSSIAN\_ELIMINATION}

---

Algorithm 8.4 A serial Gaussian elimination algorithm that converts the system of linear equations \(Ax = b\) to a unit upper-triangular system \(Ux = y\). The matrix \(U\) occupies the upper-triangular locations of \(A\). This algorithm assumes that \(A[k, k] \neq 0\) when it is used as a divisor on lines 6 and 7.

---

![Figure 3.28](image) A typical computation in Gaussian elimination and the active part of the coefficient matrix during the \(k\)th iteration of the outer loop.
Floyd’s All-Pairs Shortest Path

\[ d_{i,j}^{(k)} = \begin{cases} 
  w(v_i, v_j) & \text{if } k = 0 \\
  \min\left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right\} & \text{if } k \geq 1
\end{cases} \]

1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3. \( D^{(0)} = A; \)
4. for \( k := 1 \) to \( n \) do
5. \hspace{1em} for \( i := 1 \) to \( n \) do
6. \hspace{2em} for \( j := 1 \) to \( n \) do
7. \hspace{3em} \( d_{i,j}^{(k)} := \min\left( d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \right); \)
8. end FLOYD_ALL_PAIRS_SP

Algorithm 10.3 Floyd’s all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph \( G = (V, E) \) with adjacency matrix \( A \).
Quicksort

Algorithm 9.5  The sequential quicksort algorithm.

1. procedure QUICKSORT (A, q, r)
2. begin
3.    if q < r then
4.     begin
5.       x := A[q];
6.       s := q;
7.       for i := q + 1 to r do
8.        if A[i] ≤ x then
9.         begin
10.        s := s + 1;
11.        swap(A[s], A[i]);
12.       end if
13.     swap(A[q], A[s]);
14.     QUICKSORT (A, q, s);
15.     QUICKSORT (A, s + 1, r);
16.    end if
17. end QUICKSORT

Figure 9.15  Example of the quicksort algorithm sorting a sequence of size n = 8.
Minimum Finding

---

1. **procedure** SERIAL_MIN \((A, n)\)
2. **begin**
3. \(min = A[0] ;\)
4. **for** \(i := 1 \text{ to } n - 1 \) **do**
5. \(\text{if } (A[i] < min) \text{ then } min := A[i] ;\)
6. **endfor**;
7. **return** \(min\);
8. **end** SERIAL_MIN

**Algorithm 3.1** A serial program for finding the minimum in an array of numbers \(A\) of length \(n\).
15—Puzzle Problem

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

(a)  

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

(b)  

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

(c)  

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

(d)  

**Figure 3.17** A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.
Parallel Algorithm vs Parallel Formulation

- **Parallel Formulation**
  - Refers to a *parallelization* of a serial algorithm.

- **Parallel Algorithm**
  - May represent an entirely different algorithm than the one used serially.

We primarily focus on “Parallel Formulations”

- Our goal today is to primarily discuss how to develop such parallel formulations.
- Of course, there will always be examples of “parallel algorithms” that were not derived from serial algorithms.
Elements of a Parallel Algorithm/Formulation

- Pieces of work that can be done concurrently
  - tasks
- Mapping of the tasks onto multiple processors
  - processes vs processors
- Distribution of input/output & intermediate data across the different processors
- Management the access of shared data
  - either input or intermediate
- Synchronization of the processors at various points of the parallel execution

Holy Grail:

Maximize concurrency and reduce overheads due to parallelization!
Maximize potential speedup!
Finding Concurrent Pieces of Work

- **Decomposition:**
  - The process of dividing the computation into smaller pieces of work i.e., *tasks*

- Tasks are programmer defined and are considered to be indivisible
Example: Dense Matrix-Vector Multiplication

Tasks can be of different size.
• granularity of a task

![Diagram](image)

*Figure 3.1* Decomposition of dense matrix-vector multiplication into $n$ tasks, where $n$ is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

*Figure 3.4* Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.
Example: Query Processing

<table>
<thead>
<tr>
<th>ID#</th>
<th>Model</th>
<th>Year</th>
<th>Color</th>
<th>Dealer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>4523</td>
<td>Civic</td>
<td>2002</td>
<td>Blue</td>
<td>MN</td>
<td>$18,000</td>
</tr>
<tr>
<td>3476</td>
<td>Corolla</td>
<td>1999</td>
<td>White</td>
<td>IL</td>
<td>$15,000</td>
</tr>
<tr>
<td>7623</td>
<td>Camry</td>
<td>2001</td>
<td>Green</td>
<td>NY</td>
<td>$21,000</td>
</tr>
<tr>
<td>9834</td>
<td>Prius</td>
<td>2001</td>
<td>Green</td>
<td>CA</td>
<td>$18,000</td>
</tr>
<tr>
<td>6734</td>
<td>Civic</td>
<td>2001</td>
<td>White</td>
<td>OR</td>
<td>$17,000</td>
</tr>
<tr>
<td>5342</td>
<td>Altima</td>
<td>2001</td>
<td>Green</td>
<td>FL</td>
<td>$19,000</td>
</tr>
<tr>
<td>3845</td>
<td>Maxima</td>
<td>2001</td>
<td>Blue</td>
<td>NY</td>
<td>$22,000</td>
</tr>
<tr>
<td>8354</td>
<td>Accord</td>
<td>2000</td>
<td>Green</td>
<td>VT</td>
<td>$18,000</td>
</tr>
<tr>
<td>4395</td>
<td>Civic</td>
<td>2001</td>
<td>Red</td>
<td>CA</td>
<td>$17,000</td>
</tr>
<tr>
<td>7352</td>
<td>Civic</td>
<td>2002</td>
<td>Red</td>
<td>WA</td>
<td>$18,000</td>
</tr>
</tbody>
</table>

Query: `MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")`
Example: Query Processing

Finding concurrent tasks…

Figure 3.2 The different tables and their dependencies in a query processing operation.

Figure 3.3 An alternate data-dependency graph for the query processing operation.
Task-Dependency Graph

- In most cases, there are dependencies between the different tasks
  - certain task(s) can only start once some other task(s) have finished
    - e.g., producer-consumer relationships

- These dependencies are represented using a DAG called \textit{task-dependency graph}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{task-dependency-graph.png}
\caption{Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.}
\end{figure}
Key Concepts Derived from the Task-Dependency Graph

- **Degree of Concurrency**
  - The number of tasks that can be concurrently executed
  - We usually care about the *average* degree of concurrency

- **Critical Path**
  - The longest vertex-weighted path in the graph
  - The weights represent task size

- Task granularity affects both of the above characteristics
Task-Interaction Graph

- Captures the pattern of interaction between tasks
  - This graph usually contains the task-dependency graph as a **subgraph**
    - i.e., there may be interactions between tasks even if there are no dependencies
    - these interactions usually occur due to accesses on shared data

![Figure 3.6](image)

A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition, Task $i$ computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i,j]b[j]$. 
Task Dependency/Interaction Graphs

- These graphs are important in developing effectively mapping the tasks onto the different processors
  - Maximize concurrency and minimize overheads

More on this later…
Common Decomposition Methods

- Data Decomposition
- Recursive Decomposition
- Exploratory Decomposition
- Speculative Decomposition
- Hybrid Decomposition
Recursive Decomposition

- Suitable for problems that can be solved using the divide-and-conquer paradigm
- Each of the *subproblems* generated by the *divide* step becomes a task
Example: Quicksort

Figure 3.8  The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.
Example: Finding the Minimum

Note that we can obtain divide-and-conquer algorithms for problems that are traditionally solved using non-divide-and-conquer approaches.

Algorithm 3.2  A recursive program for finding the minimum in an array of numbers $A$ of length $n$. 

```
procedure RECURSIVE_MIN (A, n)
begin
  if ($n = 1$) then
    $min := A[0]$;
  else
    $lmin :=$ RECURSIVE_MIN ($A$, $n/2$);
    $rmin :=$ RECURSIVE_MIN ($A[n/2]$), $n - n$;
    if ($lmin < rmin$) then
      $min := lmin$;
    else
      $min := rmin$;
  endelse;
return $min$;
end RECURSIVE_MIN
```
Recursive Decomposition

- How good are the decompositions that it produces?
  - average concurrency?
  - critical path?

- How do the quicksort and min-finding decompositions measure-up?
Data Decomposition

- Used to derive concurrency for problems that operate on large amounts of data
- The idea is to derive the tasks by focusing on the multiplicity of data
- Data decomposition is often performed in two steps
  - Step 1: Partition the data
  - Step 2: Induce a computational partitioning from the data partitioning
- Which data should we partition?
  - Input/Output/Intermediate?
    - Well… all of the above—leading to different data decomposition methods
- How do induce a computational partitioning?
  - Owner-computes rule
Example: Matrix-Matrix Multiplication

- Partitioning the output data

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix} \cdot \begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix} \rightarrow \begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

(a)

Task 1: \[ C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \]
Task 2: \[ C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \]
Task 3: \[ C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \]
Task 4: \[ C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \]

(b)
Example: Matrix-Matrix Multiplication

Partitioning the intermediate data

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix} \cdot \begin{pmatrix}
B_{1,1} \\
B_{2,1}
\end{pmatrix} \rightarrow \begin{pmatrix}
D_{1,1,1} & D_{1,1,2} \\
D_{1,2,1} & D_{1,2,2}
\end{pmatrix}
\]

\[
\begin{pmatrix}
D_{1,1,1} & D_{1,1,2} \\
D_{1,2,1} & D_{1,2,2}
\end{pmatrix} + \begin{pmatrix}
D_{2,1,1} & D_{2,1,2} \\
D_{2,2,1} & D_{2,2,2}
\end{pmatrix} \rightarrow \begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
\]

A decomposition induced by a partitioning of \( D \)

- Task 01: \( D_{1,1,1} = A_{1,1}B_{1,1} \)
- Task 02: \( D_{2,1,1} = A_{1,2}B_{2,1} \)
- Task 03: \( D_{1,1,2} = A_{1,1}B_{2,1} \)
- Task 04: \( D_{2,1,2} = A_{1,2}B_{2,2} \)
- Task 05: \( D_{1,2,1} = A_{2,1}B_{1,1} \)
- Task 06: \( D_{2,2,1} = A_{2,2}B_{2,1} \)
- Task 07: \( D_{1,2,2} = A_{2,1}B_{2,2} \)
- Task 08: \( D_{2,2,2} = A_{2,2}B_{2,2} \)
- Task 09: \( C_{1,1} = D_{1,1,1} + D_{1,2,1} \)
- Task 10: \( C_{1,2} = D_{1,1,2} + D_{2,1,2} \)
- Task 11: \( C_{2,1} = D_{1,2,1} + D_{2,2,1} \)
- Task 12: \( C_{2,2} = D_{1,2,2} + D_{2,2,2} \)

**Figure 3.15** A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

**Figure 3.14** Multiplication of matrices \( A \) and \( B \) with partitioning of the three-dimensional intermediate matrix \( D \).

**Figure 3.16** The task-dependency graph of the decomposition shown in Figure 3.15.
Data Decomposition

- Is the most widely-used decomposition technique
  - after all parallel processing is often applied to problems that have a lot of data
  - splitting the work based on this data is the natural way to extract high-degree of concurrency
- It is used by itself or in conjunction with other decomposition methods
  - Hybrid decomposition

![Diagram of Hybrid Decomposition](image)

Figure 3.21 Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.
Exploratory Decomposition

- Used to decompose computations that correspond to a search of a space of solutions
Example: 15-puzzle Problem
Exploratory Decomposition

- It is not as general purpose
- It can result in speedup anomalies
  - *engineered* slow-down or superlinear speedup

![Diagram](Figure 3.19) An illustration of anomalous speedups resulting from exploratory decomposition.
Speculative Decomposition

- Used to extract concurrency in problems in which the next step is one of many possible actions that can only be determined when the current tasks finishes

- This decomposition assumes a certain outcome of the currently executed task and executes some of the next steps
  - Just like speculative execution at the microprocessor level
Example: Discrete Event Simulation

Figure 3.20  A simple network for discrete event simulation.
Speculative Execution

- If predictions are wrong...
  - work is wasted
  - work may need to be *undone*
    - state-restoring overhead
    - memory/computations

- However, it may be the only way to extract concurrency!
Mapping the Tasks

- Why do we care about task mapping?
  - Can I just randomly assign them to the available processors?

- Proper mapping is critical as it needs to minimize the parallel processing overheads
  - If $T_p$ is the parallel runtime on $p$ processors and $T_s$ is the serial runtime, then the total overhead $T_o$ is $p*T_p - T_s$
    - The work done by the parallel system beyond that required by the serial system

- Overhead sources:
  - Load imbalance
  - Inter-process communication
    - coordination/synchronization/data-sharing

remember the holy grail… they can be at odds with each other
Why Mapping can be Complicated?

- Proper mapping needs to take into account the task-dependency and interaction graphs
  - Are the tasks available a priori?
    - Static vs dynamic task generation
  - How about their computational requirements?
    - Are they uniform or non-uniform?
    - Do we know them a priori?
  - How much data is associated with each task?
  - How about the interaction patterns between the tasks?
    - Are they static or dynamic?
    - Do we know them a priori?
    - Are they data instance dependent?
    - Are they regular or irregular?
    - Are they read-only or read-write?

- Depending on the above characteristics different mapping techniques are required of different complexity and cost
Example: Simple & Complex Task Interaction

Figure 3.22 The regular two-dimensional task-interaction graph for image dithering. The pixels with dotted outline require color values from the boundary pixels of the neighboring tasks.

Figure 3.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task $i$ computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i,j] \cdot b[j]$. 
Mapping Techniques for Load Balancing

Be aware…

The assignment of tasks whose aggregate computational requirements are the same does not automatically ensure load balance.

Each processor is assigned three tasks but (a) is better than (b)!

Figure 3.23 Two mappings of a hypothetical decomposition with a synchronization.
Load Balancing Techniques

- **Static**
  - The tasks are distributed among the processors prior to the execution
  - Applicable for tasks that are
    - generated statically
    - known and/or uniform computational requirements

- **Dynamic**
  - The tasks are distributed among the processors during the execution of the algorithm
    - i.e., tasks & data are migrated
  - Applicable for tasks that are
    - generated dynamically
    - unknown computational requirements
Static Mapping—Array Distribution

- Suitable for algorithms that
  - use data decomposition
  - their underlying input/output/intermediate data are in the form of arrays

- Block Distribution
- Cyclic Distribution
- Block-Cyclic Distribution
- Randomized Block Distributions

1D/2D/3D
Examples: Block Distributions

**Figure 3.24** Examples of one-dimensional partitioning of an array among eight processes.

**Figure 3.25** Examples of two-dimensional distributions of an array, (a) on a $4 \times 4$ process grid, and (b) on a $2 \times 8$ process grid.
Examples: Block Distributions

Figure 3.26  Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices $A$ and $B$ are required by the process that computes the shaded portion of the output matrix $C$. 
Example: Block-Cyclic Distributions

Gaussian Elimination

The active portion of the array shrinks as the computations progress.

Figure 3.30 Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size $4 \times 4$, and it is mapped onto a $2 \times 2$ grid of processes in a wraparound fashion.
Random Block Distributions

- Sometimes the computations are performed only at certain portions of an array
  - sparse matrix-matrix multiplication

**Figure 3.31** Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.
Random Block Distributions

Better load balance can be achieved via a random block distribution

\[ V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] \]
\[ \text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10] \]

mapping = 8 2 6 0 3 7 11 1 9 5 4 10

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
</table>

![Figure 3.32](image) A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., \( \alpha = 3 \)).

![Figure 3.33](image) Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).
Graph Partitioning

A mapping can be achieved by directly partitioning the task interaction graph.

EG: Finite element mesh-based computations

Figure 3.34   A mesh used to model Lake Superior.
Directly partitioning this graph

Figure 3.35  A random distribution of the mesh elements to eight processes.

Figure 3.36  A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.
Example: Sparse Matrix-Vector

Another *instance* of graph partitioning

![Graph partitioning example](image)

**Figure 3.38** A mapping for sparse matrix-vector multiplication onto three processes. The list $C_i$ contains the indices of $b$ that Process $i$ needs to access from other processes.
Dynamic Load Balancing Schemes

- There is a huge body of research
  - Centralized Schemes
    - A certain processor is responsible for giving out work
      - master-slave paradigm
    - Issue:
      - task granularity
  - Distributed Schemes
    - Work can be transferred between any pairs of processors.
    - Issues:
      - How do the processors get paired?
      - Who initiates the work transfer? push vs pull
      - How much work is transferred?
Mapping to Minimize Interaction Overheads

- Maximize data locality
- Minimize volume of data-exchange
- Minimize frequency of interactions
- Minimize contention and hot spots
- Overlap computation with interactions
- Selective data and computation replication

Achieving the above is usually an interplay of decomposition and mapping and is usually done iteratively.