Logistics

- Homework
  - Homework #1 graded, returned, and answers posted
  - Homework #2 answers posted later today
  - Homework #3 posted later today (due Monday, May 11, 5pm)

- Term project
  - Proposal feedback given
  - If you did not receive feedback, see Prof. Malony

- Term paper (graduate student only)
  - Proposal feedback given
  - If you did not receive feedback, see Prof. Malony

- Term exam on May 14
  - Next Tuesday’s lecture will be a review for the midterm

- Individual programming project
  - Handed out in the programming lab on Friday, May 8
  - Due in programming lab on Friday, May 22 (2 weeks)
Grading

- Undergraduate
  - 5% homework
  - 10% pattern programming labs
  - 15% individual programming assignment
  - 35% midterm exam
  - 35% project

- Graduate
  - 15% individual programming assignment
  - 30% midterm exam
  - 35% project
  - 20% research paper
Methodological Design

- Partition
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment

I. Foster, “Designing and Building Parallel Programs,” Addison-Wesley, 1995. Book is online, see webpage.
Partitioning

- Partitioning stage is intended to expose opportunities for parallel execution
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the computational tasks associated with a problem and the data on which the tasks operates
- Domain decomposition focuses on computation data
- Functional decomposition focuses on computation tasks
- Mixing domain/functional decomposition is possible
Domain and Functional Decomposition

- Domain decomposition of 2D / 3D grid

- Functional decomposition of a climate model
Partitioning Checklist

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, may lose design flexibility.

- Does your partition avoid redundant computation and storage requirements? If not, may not be scalable.

- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.

- Does the number of tasks scale with problem size? If not may not be able to solve larger problems with more processors.

- Have you identified several alternative partitions?
Communication (Interaction)

- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control

- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: coordination degree

- Granularity and frequency of communication
  - Size of data exchange

- Think of communication as interaction and control
  - Applicable to both shared and distributed memory parallelism
Types of Communication

- Point-to-point
- Group-based
- Hierarchical
- Collective

Diagram:

- **Broadcast**
  - Source
  - Multiple Receivers

- **Scatter**
  - Source
  - Multiple Receivers

- **Gather**
  - Multiple Sources
  - Target

- **Reduction**
  - Multiple Sources
  - Target
Communication Design Checklist

- Is the distribution of communications equal?
  - Unbalanced communication may limit scalability
- What is the communication locality?
  - Wider communication locales are more expensive
- What is the degree of communication concurrency?
  - Communication operations may be parallelized
- Is computation associated with different tasks able to proceed concurrently? Can communication be overlapped with computation?
  - Try to reorder computation and communication to expose opportunities for parallelism
**Agglomeration**

- Move from parallel abstractions to real implementation
- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to **agglomerate**?
  - What happens when tasks are combined?
- Is it useful to **replicate** data and/or computation?
- Changes important algorithm and performance ratios
  - **Surface-to-volume**: reduction in communication at the expense of decreasing parallelism
  - **Communication/computation**: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap
Types of Agglomeration

- Element to column
  - Better surface to volume

- Element to block
  - Better surface to volume

- Task merging

- Task reduction
  - Reduces communication
Agglomeration Design Checklist

- Has increased locality reduced communication costs?
- Is replicated computation worth it?
- Does data replication compromise scalability?
- Is the computation still balanced?
- Is scalability in problem size still possible?
- Is there still sufficient concurrency?
- Is there room for more agglomeration?
- Fine-grained vs. coarse-grained?
Mapping

- Specify where each task is to execute
  - Less of a concern on shared-memory systems

- Attempt to minimize execution time
  - Place concurrent tasks on different processors to enhance physical concurrency
  - Place communicating tasks on same processor, or on processors close to each other, to increase locality
  - Strategies can conflict!

- Mapping problem is NP-complete
  - Use problem classifications and heuristics

- Static and dynamic load balancing
Mapping Algorithms

- Load balancing (partitioning) algorithms
- Data-based algorithms
  - Think of computational load with respect to amount of data being operated on
  - Assign data (i.e., work) in some known manner to balance
  - Take into account data interactions
- Task-based (task scheduling) algorithms
  - Used when functional decomposition yields many tasks with weak locality requirements
  - Use task assignment to keep processors busy computing
  - Consider centralized and decentralize schemes
Mapping Design Checklist

- Is static mapping too restrictive and non-responsive?
- Is dynamic mapping too costly in overhead?
- Does centralized scheduling lead to bottlenecks?
- Do dynamic load-balancing schemes require too much coordination to re-balance the load?
- What is the tradeoff of dynamic scheduling complexity versus performance improvement?
- Are there enough tasks to achieve high levels of concurrency? If not, processors may idle.
Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - all processors do same thing on different data
  - Task parallelism
    - processors are assigned tasks that do different things

- Parallel execution models
  - Data parallel
  - Pipelining (Producer-Consumer)
  - Task graph
  - Work pool
  - Master-Worker
Data Parallel

- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- Certainly may have interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
  - Makes data parallel algorithms more efficient
- Single Program Multiple Data (SPMD)
  - Convenient way to implement data parallel computation
  - More associated with distributed memory parallel execution
Matrix - Vector Multiplication

- \( A \times b = y \)
- Allocate tasks to rows of \( A \)
  \[ y[i] = \sum_{j} A[i,j] \cdot b[j] \]
- Dependencies?
- Speedup?
- Computing each element of \( y \) can be done independently
Matrix-Vector Multiplication (Limited Tasks)

- Suppose we only have 4 tasks
- Dependencies?
- Speedup?

![Diagram of matrix-vector multiplication with 4 tasks and dependencies]
Matrix Multiplication

- $A \times B = C$
- $A[i,:] \cdot B[:,j] = C[i,j]$

- **Row partitioning**
  - $N$ tasks

- **Block partitioning**
  - $N\times N/B$ tasks

- Shading shows data sharing in $B$ matrix
Granularity of Task and Data Decompositions

- Granularity can be with respect to tasks and data

- Task granularity
  - Equivalent to choosing the number of tasks
  - Fine-grained decomposition results in large number of tasks
  - Large-grained decomposition has smaller number of tasks
  - Translates to data granularity after number of tasks chosen
    - consider matrix multiplication

- Data granularity
  - Think of in terms of amount of data needed in operation
  - Relative to data as a whole
  - Decomposition decisions based on input, output, input-output, or intermediate data
Mesh Allocation to Processors

- Mesh model of Lake Superior
- How to assign mesh elements to processors

- Distribute onto 8 processors
  - randomly
  - graph partitioning for minimum edge cut
Pipeline Model

- Stream of data operated on by succession of tasks
  - Task 1  Task 2  Task 3  Task 4
  - Tasks are assigned to processors
- Consider $N$ data units
- Sequential

- Parallel (each task assigned to a processor)
  - 4-way parallel
  - 8-way parallel, but for longer time
Pipeline Performance

- $N$ data and $T$ tasks
- Each task takes unit time $t$
- Sequential time = $N*T*t$
- Parallel pipeline time = $start + finish + (N-2T)/T * t$
  \[= O(N/T) \quad \text{(for } N \gg T)\]
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Shorter tasks to do
- Pipeline computation is a special form of producer-consumer parallelism
  - Producer tasks output data input by consumer tasks
Tasks Graphs

- Computations in any parallel algorithms can be viewed as a task dependency graph.
- Task dependency graphs can be non-trivial:
  - Pipeline
  - Arbitrary (represents the algorithm dependencies)

Numbers are time taken to perform task.

(a) Example of a pipeline task dependency graph.

(b) Example of an arbitrary task dependency graph.
Task Graph Performance

- Determined by the *critical path (span)*
  - Sequence of dependent tasks that takes the longest time

\[
\begin{align*}
\text{Min time} &= 27 \\
\text{Min time} &= 34
\end{align*}
\]

- Critical path length bounds parallel execution time
**Task Assignment (Mapping) to Processors**

- Given a set of tasks and number of processors
- How to assign tasks to processors?
- Should take dependencies into account
- Task mapping will determine execution time

```
Total time = ?
```

```
Total time = ?
```

(a) Task Allocation Diagram

(b) Different Task Allocation Diagram
**Task Graphs in Action**

- Uintah task graph scheduler
  - C-SAFE: Center for Simulation of Accidental Fires and Explosions, University of Utah
  - Large granularity tasks

- PLASMA
  - DAG-based parallel linear algebra
  - DAGuE: A generic distributed DAG engine for HPC

---

DAG of QR for a $4 \times 4$ tiles matrix on a $2 \times 2$ grid of processors.
Bag o’ Tasks Model and Worker Pool

- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o’ Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisfied, it is moved to a ready to run queue
  - Scheduler assigns a task to a free processor
- Dynamic approach that is effective for load balancing
Master-Worker Parallelism

- One or more master processes generate work
- Masters allocate work to worker processes
- Workers idle if have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but must be told to do so
- Master may become the bottleneck if not careful
- What are the performance factors and expected performance behavior
  - Consider task granularity and asynchrony
  - How do they interact?
Master-Worker Execution Model (Li Li)

M-W Execution Trace (Li Li)

![Diagram showing M-W Execution Trace](image-url)
Search-Based (Exploratory) Decomposition

- 15-puzzle problem
- 15 tiles numbered 1 through 15 placed in 4x4 grid
  - Blank tile located somewhere in grid
  - Initial configuration is out of order
  - Find shortest sequence of moves to put in order

- Sequential search across space of solutions
  - May involve some heuristics
Parallelizing the 15-Puzzle Problem

- Enumerate move choices at each stage
- Assign to processors
- May do pruning
- Wasted work
**Divide-and-Conquer Parallelism**

- Break problem up in orderly manner into smaller, more manageable chunks and solve
- Quicksort example
Dense Matrix Algorithms

- Great deal of activity in algorithms and software for solving linear algebra problems
  - Solution of linear systems (Ax = b)
  - Least-squares solution of over- or under-determined systems (min ||Ax-b||)
  - Computation of eigenvalues and eigenvectors (Ax=λx)
  - Driven by numerical problem solving in scientific computation

- Solutions involves various forms of matrix computations

- Focus on high-performance matrix algorithms
  - Key insight is to maximize computation to communication
Solving a System of Linear Equations

- $Ax = b$

\[ a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,n-1}x_{n-1} = b_0 \]
\[ a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,n-1}x_{n-1} = b_1 \]

... 

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

- Gaussian elimination (classic algorithm)
  - Forward elimination to $Ux = y$ ($U$ is upper triangular)
    - without or with partial pivoting
  - Back substitution to solve for $x$
  - Parallel algorithms based on partitioning of $A$
Sequential Gaussian Elimination

1. procedure GAUSSIAN ELIMINATION (A, b, y)
2. Begin
3. for \( k := 0 \) to \( n - 1 \) do /* Outer loop */
4. begin
5. for \( j := k + 1 \) to \( n - 1 \) do
7. \( y[k] := b[k] / A[k, k]; 
8. \( A[k, k] := 1; 
9. for \( i := k + 1 \) to \( n - 1 \) do
10. begin
11. for \( j := k + 1 \) to \( n - 1 \) do
13. \( b[i] := b[i] - A[i, k] \times y[k]; 
14. \( A[i, k] := 0; 
15. endfor; /*Line9*/
16. endfor; /*Line3*/
17. end GAUSSIAN ELIMINATION
Computation Step in Gaussian Elimination

\[
\begin{align*}
5x + 3y &= 22 \\
8x + 2y &= 13
\end{align*}
\]

\[
\begin{align*}
x &= \frac{22 - 3y}{5} \\
8\left(\frac{22 - 3y}{5}\right) + 2y &= 13 \\
x &= \frac{22 - 3y}{5} \\
y &= \frac{13 - 176/5}{24/5 + 2}
\end{align*}
\]
## Rowwise Partitioning on Eight Processes

| P<sub>0</sub> | 1   | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 |
| P<sub>1</sub> | 0   | 1   | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 |
| P<sub>2</sub> | 0   | 0   | 1   | 2.3 | 2.4 | 2.5 | 2.6 | 2.7 |
| P<sub>3</sub> | 0   | 0   | 0   | 3.3 | 3.4 | 3.5 | 3.6 | 3.7 |
| P<sub>4</sub> | 0   | 0   | 0   | 4.3 | 4.4 | 4.5 | 4.6 | 4.7 |
| P<sub>5</sub> | 0   | 0   | 0   | 5.3 | 5.4 | 5.5 | 5.6 | 5.7 |
| P<sub>6</sub> | 0   | 0   | 0   | 6.3 | 6.4 | 6.5 | 6.6 | 6.7 |
| P<sub>7</sub> | 0   | 0   | 0   | 7.3 | 7.4 | 7.5 | 7.6 | 7.7 |

### Computation:

1. \( A[k,j] := A[k,j]/A[k,k] \) for \( k < j < n \)
2. \( A[k,k] := 1 \)

### Communication:

One-to-all broadcast of row \( A[k,\ast] \)
### Rowwise Partitioning on Eight Processes

<table>
<thead>
<tr>
<th></th>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_5</th>
<th>P_6</th>
<th>P_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_0</td>
<td>1</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
<td>(0,4)</td>
<td>(0,5)</td>
<td>(0,6)</td>
<td>(0,7)</td>
</tr>
<tr>
<td>P_1</td>
<td>0</td>
<td>1</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
</tr>
<tr>
<td>P_2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
</tr>
<tr>
<td>P_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
</tr>
<tr>
<td>P_4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
</tr>
<tr>
<td>P_5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
</tr>
<tr>
<td>P_6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
</tr>
<tr>
<td>P_7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
</tr>
</tbody>
</table>

(c) Computation:

for \( k < i < n \) and \( k < j < n \)

(ii) \( A[i,k] := 0 \) for \( k < i < n \)
### 2D Mesh Partitioning on 64 Processes

<table>
<thead>
<tr>
<th>1</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 1 (2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (3,3)</td>
<td>(3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Rowwise broadcast of $A[i,k]$ for $(k-1) < i < n$

<table>
<thead>
<tr>
<th>1</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 1 (2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 1 (3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) $A[k,j] := A[k,j]/A[k,k]$ for $k < j < n$

<table>
<thead>
<tr>
<th>1</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 1 (2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 1 (3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) Columnwise broadcast of $A[k,j]$ for $k < j < n$

<table>
<thead>
<tr>
<th>1</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th>(0,4)</th>
<th>(0,5)</th>
<th>(0,6)</th>
<th>(0,7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (1,2)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,5)</td>
<td>(1,6)</td>
<td>(1,7)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 1 (2,3)</td>
<td>(2,4)</td>
<td>(2,5)</td>
<td>(2,6)</td>
<td>(2,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 1 (3,4)</td>
<td>(3,5)</td>
<td>(3,6)</td>
<td>(3,7)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (4,3)</td>
<td>(4,4)</td>
<td>(4,5)</td>
<td>(4,6)</td>
<td>(4,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (5,3)</td>
<td>(5,4)</td>
<td>(5,5)</td>
<td>(5,6)</td>
<td>(5,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (6,3)</td>
<td>(6,4)</td>
<td>(6,5)</td>
<td>(6,6)</td>
<td>(6,7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 0 (7,3)</td>
<td>(7,4)</td>
<td>(7,5)</td>
<td>(7,6)</td>
<td>(7,7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Back Substitution to Find Solution

1. procedure BACK SUBSTITUTION \((U, x, y)\)
2. begin
3. for \(k := n - 1\) downto 0 do /* Main loop */
4. begin
5. \(x[k] := y[k];\)
6. for \(i := k - 1\) downto 0 do
7. \(y[i] := y[i] - x[k] \times U[i, k];\)
8. endfor;
9. end BACK SUBSTITUTION
Dense Linear Algebra (www.netlib.gov)

- Basic Linear Algebra Subroutines (BLAS)
  - Level 1 (*vector-vector*): vectorization
  - Level 2 (*matrix-vector*): vectorization, parallelization
  - Level 3 (*matrix-matrix*): parallelization

- LINPACK (Fortran)
  - Linear equations and linear least-squares

- EISPACK (Fortran)
  - Eigenvalues and eigenvectors for matrix classes

- LAPACK (Fortran, C) (LINPACK + EISPACK)
  - Use BLAS internally

- ScaLAPACK (Fortran, C, MPI) (scalable LAPACK)
Numerical Libraries

- **PETSc**
  - Data structures / routines for partial differential equations
  - MPI based
- **SuperLU**
  - Large sparse nonsymmetric linear systems
- **Hypre**
  - Large sparse linear systems
- **TAO**
  - Toolkit for Advanced Optimization
- **DOE ACTS**
  - Advanced CompuTational Software
Sorting Algorithms

- Task of arranging unordered collection into order
- Permutation of a sequence of elements
- Internal versus external sorting
  - External sorting uses auxiliary storage
- Comparison-based
  - Compare pairs of elements and exchange
    - $O(n \log n)$
- Noncomparison-based
  - Use known properties of elements
    - $O(n)$
Sorting on Parallel Computers

- Where are the elements stored?
  - Need to be distributed across processes
  - Sorted order will be with respect to process order

- How are comparisons performed?
  - One element per process
    - compare-exchange
    - interprocess communication will dominate execution time
  - More than one element per process
    - compare-split

- Sorting networks
  - Based on comparison network model

- Contrast with shared memory sorting algorithms
Single vs. Multi Element Comparison

- One element per processor
  
  $a_i \xrightarrow{\quad} a_j \xrightarrow{\quad} a_i, a_j \xrightarrow{\quad} a_j, a_i \xrightarrow{\quad} \min\{a_i, a_j\} \xrightarrow{\quad} \max\{a_i, a_j\}$

  \begin{align*}
  P_i & \quad P_j \\
  \text{Step 1} & \quad \text{Step 2} & \quad \text{Step 3}
  \end{align*}

- Multiple elements per processor
  
  $\begin{array}{c}
  1 & 6 & 8 & 13 \\
  P_i & \quad P_j
  \end{array} \xrightarrow{\quad} \begin{array}{c}
  2 & 7 & 9 & 10 & 12 \\
  P_i & \quad P_j
  \end{array} \xrightarrow{\quad} \begin{array}{c}
  1 & 6 & 8 & 13 \\
  P_i & \quad P_j
  \end{array}$

  \begin{align*}
  P_i & \quad P_j \\
  \text{Step 1} & \quad \text{Step 2}
  \end{align*}

  $\begin{array}{c}
  1 & 2 & 6 & 7 & 8 & 9 & 10 & 12 & 13 \\
  P_i & \quad P_j
  \end{array} \xrightarrow{\quad} \begin{array}{c}
  1 & 2 & 6 & 7 & 8 & 9 & 10 & 12 & 13 \\
  P_i & \quad P_j
  \end{array} \xrightarrow{\quad} \begin{array}{c}
  1 & 2 & 6 & 7 & 8 \\
  P_i & \quad P_j
  \end{array}$

  \begin{align*}
  P_i & \quad P_j \\
  \text{Step 3} & \quad \text{Step 4}
  \end{align*}
**Sorting Networks**

- Networks to sort \( n \) elements in less than \( O(n \log n) \)
- Key component in network is a comparator
  - Increasing or decreasing comparator

![Comparator Diagram](image)

- Comparators connected in parallel and permute elements
Sorting Network Design

- Multiple comparator stages (# stages, # comparators)
- Connected together by interconnection network
- Output of last stage is the sorted list
- $O(\log_2 n)$ sorting time
- Convert any sorting network to sequential algorithm
Bitonic Sort

- Create a *bitonic sequence* then sort the sequence
- Bitonic sequence
  - sequence of elements $<a_0, a_1, \ldots, a_{n-1}>$
  - $<a_0, a_1, \ldots, a_i>$ is monotonically increasing
  - $<a_i, a_{i+1}, \ldots, a_{n-1}>$ is monotonically decreasing
- Sorting using *bitonic splits* is called *bitonic merge*
- *Bitonic merge network* is a network of comparators
  - Implement bitonic merge
- Bitonic sequence is formed from unordered sequence
  - Bitonic sort creates a bitonic sequence
  - Start with sequence of size two (default bitonic)
Bitonic Sort Network

Unordered sequence

Bitonic sequence

Wires
0000 10 10 5 3
0001 20 20 9 5
0010 5 9 10 8
0011 9 5 20 9
0100 3 3 14 10
0101 8 8 12 12
0110 12 14 8 14
0111 14 12 3 20
1000 90 0 0 95
1001 0 90 40 90
1010 60 60 60 60
1011 40 40 90 40
1100 23 23 95 35
1101 35 35 35 23
1110 95 95 23 18
1111 18 18 18 0

decrease

increase
**Bitonic Merge Network**

Bitonic sequence

Sorted sequence

<table>
<thead>
<tr>
<th>Wires</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>20</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>20</td>
<td>95</td>
<td>90</td>
<td>60</td>
<td>40</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>5</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>9</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>18</td>
<td>23</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0011</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>18</td>
<td>23</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0101</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0110</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>0111</td>
<td>20</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>20</td>
<td>35</td>
<td>23</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>1000</td>
<td>95</td>
<td>95</td>
<td>35</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>1001</td>
<td>90</td>
<td>90</td>
<td>23</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>1010</td>
<td>60</td>
<td>60</td>
<td>18</td>
<td>35</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>1011</td>
<td>40</td>
<td>40</td>
<td>20</td>
<td>23</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>1100</td>
<td>35</td>
<td>35</td>
<td>95</td>
<td>60</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>1101</td>
<td>23</td>
<td>23</td>
<td>90</td>
<td>40</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>1110</td>
<td>18</td>
<td>18</td>
<td>60</td>
<td>95</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>20</td>
<td>40</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>
Parallel Bitonic Sort on a Hypercube

1. procedure BITONIC SORT(label, d)
2. begin
3. for $i := 0$ to $d - 1$ do
4. for $j := i$ downto 0 do
5. if $(i + 1)$st bit of label = $j$th bit of label then
6. comp exchange max($j$);
7. else
8. comp exchange min($j$);
9. end BITONIC SORT
Parallel Bitonic Sort on a Hypercube

Step 1

Step 2

Last stage

Step 3

Step 4
Bubble Sort and Variants

- Can easily parallelize sorting algorithms of $O(n^2)$
- *Bubble sort* compares and exchanges adjacent elements
  - $O(n)$ each pass
  - $O(n)$ passes
  - Available parallelism?

- *Odd-even transposition sort*
  - Compares and exchanges odd and even pairs
  - After $n$ phases, elements are sorted
  - Available parallelism?
Odd-Even Transposition Sort

Unsorted

3  2  3  8  5  6  4  1

Phase 1 (odd)

2  3  3  8  5  6  1  4

Phase 2 (even)

2  3  3  5  8  1  6  4

Phase 3 (odd)

2  3  3  5  1  8  4  6

Phase 4 (even)

2  3  3  1  5  4  8  6

Phase 5 (odd)

2  3  1  3  4  5  6  8

Phase 6 (even)

2  1  3  3  4  5  6  8

Phase 7 (odd)

1  2  3  3  4  5  6  8

Phase 8 (even)

Sorted

1  2  3  3  4  5  6  8
Parallel Odd-Even Transposition Sort

1. procedure ODD-EVEN PAR\( (n) \)
2. begin
3. \( id := \) process’ s label
4. for \( i := 1 \) to \( n \) do
5. begin
6. if \( i \) is odd then
7. \hspace{1em} if \( id \) is odd then
8. \hspace{2em} compare-exchange \( \min(id + 1) \);
9. \hspace{1em} else
10. \hspace{2em} compare-exchange \( \max(id - 1) \);
11. if \( i \) is even then
12. \hspace{1em} if \( id \) is even then
13. \hspace{2em} compare-exchange \( \min(id + 1) \);
14. \hspace{1em} else
15. \hspace{2em} compare-exchange \( \max(id - 1) \);
16. end for
17. end ODD-EVEN PAR
Quicksort

- Quicksort has average complexity of $O(n \log n)$
- Divide-and-conquer algorithm
  - Divide into subsequences where every element in first is less than or equal to every element in the second
  - Pivot is used to split the sequence
  - Conquer step recursively applies quicksort algorithm
- Available parallelism?
Sequential Quicksort

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
Parallel Shared Address Space Quicksort

First Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>13</td>
<td>18</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>19</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

pivot = 7

after local rearrangement

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>18</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>20</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>12</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>

after global rearrangement

Second Step

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>16</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>13</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>15</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

pivot = 5

pivot = 17

after local rearrangement

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>14</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>15</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

after global rearrangement
Parallel Shared Address Space Quicksort

Third Step

Fourth Step

Solution
Bucket Sort and Sample Sort

- *Bucket sort* is popular when elements (values) are uniformly distributed over an interval
  - Create $m$ buckets and place elements in appropriate bucket
  - $O(n \log(n/m))$
  - If $m=n$, can use value as index to achieve $O(n)$ time

- *Sample sort* is used when uniformly distributed assumption is not true
  - Distributed to $m$ buckets and sort each with quicksort
  - Draw sample of size $s$
  - Sort samples and choose $m-1$ elements to be *splitters*
  - Split into $m$ buckets and proceed with bucket sort
Parallel Sample Sort

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment
Graph Algorithms

- Graph theory important in computer science
- Many complex problems are graph problems

- $G = (V, E)$
  - $V$ finite set of points
  - $E$ finite set of edges
  - $e \in E$ is an pair $(u,v)$, where $u, v \in V$
  - Unordered and ordered graphs
Graph Terminology

- Vertex adjacency if \((u,v)\) is an edge
- Path from \(u\) to \(v\) if there is an edge sequence starting at \(u\) and ending at \(v\)
- If there exists a path, \(v\) is reachable from \(u\)
- A graph is connected if all pairs of vertices are connected by a path
- A weighted graph associates weights with each edge
- Adjacency matrix is an \(n \times n\) array \(A\) such that
  - \(A_{i,j} = 1\) if \((v_i,v_j) \in E\); 0 otherwise
  - Can be modified for weighted graphs (\(\infty\) is no edge)
  - Can represent as adjacency lists
Graph Representations

- Adjacency matrix

- Adjacency list
Minimum Spanning Tree

- A spanning tree of an undirected graph $G$ is a subgraph of $G$ that is a tree containing all the vertices of $G$
- The minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight
- Prim’s algorithm can be used
  - Greedy algorithm
  - Selects an arbitrary starting vertex
  - Chooses new vertex guaranteed to be in MST
  - $O(n^2)$
  - Prim’s algorithm is iterative
Prim’s Minimum Spanning Tree Algorithm

1. procedure PRIM MST(\(V, E, w, r\))
2. begin
3. \(VT := \{r\}\);
4. \(d[r] := 0\);
5. for all \(v \in (V - VT)\) do
6. if edge \((r, v)\) exists set \(d[v] := w(r, v)\);
7. else set \(d[v] := \infty\);
8. while \(VT \neq V\) do
9. begin
10. find a vertex \(u\) such that \(d[u] := \min\{d[v] | v \in (V - VT)\}\);
11. \(VT := VT \cup \{u\}\);
12. for all \(v \in (V - VT)\) do
13. \(d[v] := \min\{d[v], w(u, v)\}\)\)
14. endwhile
15. end PRIM MST
Example: Prim’s MST Algorithm

(a) Original graph

(b) After the first edge has been selected
Example: Prim’s MST Algorithm

(c) After the second edge has been selected

(d) Final minimum spanning tree
Parallel Formulation of Prim’s Algorithm

- Difficult to perform different iterations of the while loop in parallel because $d[v]$ may change each time
- Can parallelize each iteration though
- Partition vertices into $p$ subsets $V_i$, $i=0,...,p-1$
- Each process $P_i$ computes
  \[ d_i[u] = \min\{d_i[v] \mid v \in (V-V_T) \cap V_i\} \]
- Global minimum is obtained using all-to-one reduction
- New vertex is added to $V_T$ and broadcast to all processes
- New values of $d[v]$ are computed for local vertex
- $O(n^2/p) + O(n \log p)$ (computation + communication)
Partitioning in Prim’s Algorithm

\[ d[1..n] \]

| \( \frac{n}{p} \) |

(a)

\[ A \]

(b)

\[ \text{Processors} \quad 0 \quad 1 \quad i \quad p-1 \]
Single-Source Shortest Paths

- Find shortest path from a vertex \( v \) to all other vertices
- The shortest path in a weighted graph is the edge with the minimum weight
- Weights may represent time, cost, loss, or any other quantity that accumulates additively along a path
- Dijkstra’s algorithm finds shortest paths from vertex \( s \)
  - Similar to Prim’s MST algorithm
    - MST with vertex \( v \) as starting vertex
  - Incrementally finds shortest paths in greedy manner
  - Keep track of minimum cost to reach a vertex from \( s \)
  - \( O(n^2) \)
Dijkstra’s Single-Source Shortest Path

1. procedure DIJKSTRA SINGLE SOURCE SP(V, E, w, s)
2. begin
3. $V_T := \{s\};$
4. for all $v \in (V - V_T)$ do
5. if $(s, v)$ exists set $l[v] := w(s, v);$ 
6. else set $l[v] := \infty;$
7. while $V_T \neq V$ do
8. begin
9. find a vertex $u$ such that $l[u] := \min\{l[v]|v \in (V - V_T)\};$
10. $V T := V_T \cup \{u\};$
11. for all $v \in (V - V_T)$ do
12. $l[v] := \min\{l[v], l[u] + w(u, v)\};$
13. endwhile
14. end DIJKSTRA SINGLE SOURCE SP
Parallel Formulation of Dijkstra’s Algorithm

- Very similar to Prim’s MST parallel formulation
- Use 1D block mapping as before
- All processes perform computation and communication similar to that performed in Prim’s algorithm

- Parallel performance is the same
  - $O(n^2/p) + O(n \log p)$
  - Scalability
    - $O(n^2)$ is the sequential time
    - $O(n^2) / [O(n^2/p) + O(n \log p)]$
**All Pairs Shortest Path**

- Find the shortest path between all pairs of vertices
- Outcome is a $n \times n$ matrix $D = \{d_{i,j}\}$ such that $d_{i,j}$ is the cost of the shortest path from vertex $v_i$ to vertex $v_j$
- **Dijkstra’s algorithm**
  - Execute single-source algorithm on each process
  - $O(n^3)$
  - Source-partitioned formulation (use sequential algorithm)
  - Source-parallel formulation (use parallel algorithm)
- **Floyd’s algorithm**
  - Builds up distance matrix from the bottom up
Floyd’s All-Pairs Shortest Paths Algorithm

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} \(d^{(k)}_{i, j} := \min d^{(k-1)}_{i, j}, d^{(k-1)}_{i, k} + d^{(k-1)}_{k, j};\)
8. end FLOYD ALL PAIRS SP
Parallel Floyd’s Algorithm

1. procedure FLOYD ALL PAIRS PARALLEL (A)
2. begin
3. \( D^{(0)} = A; \)
4. for \( k := 1 \) to \( n \) do
5. \hspace{1em} forall \( P_{i,j} \), where \( i, j \leq n \), do in parallel
6. \( d^{(k)}_{i,j} := \min d^{(k-1)}_{i,j} , d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j} ; \)
7. end FLOYD ALL PAIRS PARALLEL
Parallel Graph Algorithm Library – Boost

- Parallel Boost Graph Library
  - Andrew Lumsdaine, Indiana University
  - Generic C++ library for high-performance parallel and distributed graph computation
  - Builds on the Boost Graph Library (BGL)
    - offers similar data structures, algorithms, and syntax
  - Targets very large graphs (millions of nodes)
  - Distributed-memory parallel processing on clusters
Original BGL: Algorithms

- Searches (breadth-first, depth-first, A*)
- Single-source shortest paths (Dijkstra, Bellman-Ford, DAG)
- All-pairs shortest paths (Johnson, Floyd-Warshall)
- Minimum spanning tree (Kruskal, Prim)
- Components (connected, strongly connected, biconnected)
- Maximum cardinality matching
- Max-flow (Edmonds-Karp, push-relabel)
- Sparse matrix ordering (Cuthill-McKee, King, Sloan, minimum degree)
- Layout (Kamada-Kawai, Fruchterman-Reingold, Gursoy-Atun)
- Betweenness centrality
- PageRank
- Isomorphism
- Vertex coloring
- Transitive closure
- Dominator tree
**Original BGL Summary**

- Original BGL is large, stable, efficient
  - Lots of algorithms, graph types
  - Peer-reviewed code with many users, nightly regression testing, and so on
  - Performance comparable to FORTRAN.

- Who should use the BGL?
  - Programmers comfortable with C++
  - Users with graph sizes from tens of vertices to millions of vertices
Parallel BGL

- A version of C++ BGL for computational clusters
  - Distributed memory for huge graphs
  - Parallel processing for improved performance
- An active research project
- Closely related to the original BGL
  - Parallelizing BGL programs should be “easy”

A simple, directed graph… distributed across 3 processors
Parallel Graph Algorithms

- Breadth-first search
- Eager Dijkstra’s single-source shortest paths
- Crauser et al. single-source shortest paths
- Depth-first search
- Minimum spanning tree (Boruvka, Dehne & Götz)
- Connected components
- Strongly connected components
- Biconnected components
- PageRank
- Graph coloring
- Fruchterman-Reingold layout
- Max-flow (Dinic’s)
Big-Data and Map-Reduce

- Big-data deals with processing large data sets
- Nature of data processing problem makes it amenable to parallelism
  - Looking for features in the data
  - Extracting certain characteristics
  - Analyzing properties with complex data mining algorithms
- Data size makes it opportunistic for partitioning into large # of sub-sets and processing these in parallel
- We need new algorithms, data structures, and programming models to deal with problems
A Simple Big-Data Problem

- Consider a large data collection of text documents
- Suppose we want to find how often a particular word occurs and determine a probability distribution for all word occurrences

Sequential algorithm

<table>
<thead>
<tr>
<th>Data collection</th>
<th>Get next document</th>
<th>Find and count words</th>
<th>Count words and update statistics</th>
<th>Generate probability distributions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>web</td>
<td>2</td>
<td>weed</td>
<td>1</td>
<td>green</td>
</tr>
</tbody>
</table>
Parallelization Approach

- **Map**: partition the data collection into subsets of documents and process each subset in parallel
- **Reduce**: assemble the partial frequency tables to derive final probability distribution

**Parallel algorithm**

1. Get next document
2. Find and count words
3. Count words and update statistics
4. Check if more documents
5. Generate probability distributions
**Parallelization Approach**

- **Map**: partition the data collection into subsets of documents and process each subset in parallel
- **Reduce**: assemble the partial frequency tables to derive final probability distribution

**Parallel algorithm**

```
Data collection

Get next document

Find and count words

Count words and update statistics

Generate probability distributions

Check if more documents
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>web</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weed</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>green</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sun</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moon</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>land</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>part</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Actually, it is not easy to parallel….

Fundamental issues
Scheduling, data distribution, synchronization, inter-process communication, robustness, fault tolerance, …

Architectural issues
Flynn’s taxonomy (SIMD, MIMD, etc.), network topology, bisection bandwidth, cache coherence, …

Common problems
Livelock, deadlock, data starvation, priority inversion, …dining philosophers, sleeping barbers, cigarette smokers, …

Actually, Programmer’s Nightmare….
Map-Reduce Parallel Programming

- Become an important distributed parallel programming paradigm for large-scale applications
  - Also applies to shared-memory parallelism
  - Becomes one of the core technologies powering big IT companies, like Google, IBM, Yahoo and Facebook.

- Framework runs on a cluster of machines and automatically partitions jobs into number of small tasks and processes them in parallel

- Can capture in combining Map and Reduce parallel patterns
**Map-Reduce Example**

MAP: Input data $\rightarrow$ <key, value> pair

- **Data Collection: split1**
- **Data Collection: split 2**
- **Data Collection: split n**

Split the data to
Supply multiple processors

CIS 410/510: Parallel Computing, University of Oregon, Spring 2015
**MapReduce**

MAP: Input data → <key, value> pair
REDUCE: <key, value> pair → <result>

Split the data to Supply multiple processors

Data Collection: split 1

Data Collection: split 2

Data Collection: split n