CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Introduction to Parallel Algorithms

Instructor: Haidar M. Harmanani
Spring 2016

Methodological Design

- **Partition**
  - Task/data decomposition

- **Communication**
  - Task execution coordination

- **Agglomeration**
  - Evaluation of the structure

- **Mapping**
  - Resource assignment

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
**Divide and Conquer**

- Divide a problem into sub-problems that are of the same form as the larger problem
- Further divisions into still smaller sub-problems are usually done by recursion

**M-ary Divide and Conquer**

- Divide and conquer can also be applied where a task is divided into more than two parts at each stage
- For example, if the task is broken into four parts, the sequential recursive definition would be

```c
int add(int *s) /* add list of numbers, s */
{
    if (number(s) <= 4) return(n1 + n2 + n3 + n4);
    else {
        Divide (s,s1,s2,s3,s4); /* divide s into s1,s2,s3,s4*/
        part_sum1 = add(s1);  /*recursive calls to add sublists*/
        part_sum2 = add(s2);
        part_sum3 = add(s3);
        part_sum4 = add(s4);
        return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
    }
}
```
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
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
- 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 \( A \)
  - \( y[i] = \sum A[i,j] \times b[j] \)
- Dependencies?
- Speedup?
- Computing each element of \( y \) can be done independently
Matrix-Vector Multiplication (Limited Tasks)

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

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 # tasks
  - Large-grained decomposition has smaller # tasks
  - Translates to data granularity after # 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

Centralized work pool
Decentralized Dynamic Load Balancing Distributed Work Pool

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
Pipelined Computations

- Pipelined program divided into a series of tasks that have to be completed one after the other.
- Each task executed by a separate pipeline stage
- Data streamed from stage to stage to form computation

\[ f, e, d, c, b, a \rightarrow P1 \rightarrow P2 \rightarrow P3 \rightarrow P4 \rightarrow P5 \]
Pipeline Performance

- N data and T tasks
- Each task takes unit time t
- Sequential time = N*T*t
- Parallel pipeline time = start + finish + \(\frac{(N-2T)}{T} \cdot t\)
  \[= O\left(\frac{N}{T}\right)\] (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

Example

- Frequency filter - Objective to remove specific frequencies (f0, f1, f2, f3, etc.) from a digitized signal, f(t).
- Signal enters pipeline from left:

\[\text{Signal without frequency } f_0\rightarrow f_\text{in} \rightarrow f_\text{out}\]
\[\text{Signal without frequency } f_1\rightarrow f_\text{in} \rightarrow f_\text{out}\]
\[\text{Signal without frequency } f_2\rightarrow f_\text{in} \rightarrow f_\text{out}\]
\[\text{Signal without frequency } f_3\rightarrow f_\text{in} \rightarrow f_\text{out}\]
\[\text{Filtered signal } \]

\[f(t)\rightarrow f_\text{in} \rightarrow f_\text{out}\]
\[f_\text{in} \rightarrow f_\text{out}\]
\[f_\text{in} \rightarrow f_\text{out}\]
\[f_\text{in} \rightarrow f_\text{out}\]
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

Task Graph Performance

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

Min time = 27
- Critical path length bounds parallel execution time.

Min time = 34
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

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

The task graphs illustrate the assignment of tasks to processors. The total time for each configuration is to be determined.
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 most 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?
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

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

- Seq (a) (b) (c) (d)
  - 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 ($Ax = \lambda x$)
  - Driven by numerical problem solving in scientific computation
- Solutions involve 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

- \( Ax = b \)
  
  \[
  \begin{align*}
  a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n-1}x_{n-1} &= b_0 \\
  a_{1,0}x_0 + a_{1,1}x_1 + \ldots + a_{1,n-1}x_{n-1} &= b_1 \\
  \vdots \\
  a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} &= b_{n-1}
  \end{align*}
  \]

- 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.   endfor; /*Line 9*/
10. for \( i := k + 1 \) to \( n - 1 \) do
11.   begin
12.     for \( j := k + 1 \) to \( n \) do
14.       \( b[i] := b[i] - A[i, k] \times y[k]; \)
15.     endfor; /*Line 9*/
16.   endfor; /*Line 3*/
17. end GAUSSIAN ELIMINATION
Computation Step in Gaussian Elimination

\[
5x + 3y = 22 \quad \Rightarrow \quad x = \frac{(22 - 3y)}{5}
\]
\[
8x + 2y = 13 \quad \Rightarrow \quad y = \frac{(13 - 176/5)}{(24/5 + 2)}
\]

Rowwise Partitioning on Eight Processes

(a) Computation:
(i) \(A[k,j] := A[k,j]/A[k,k]\) for \(k < j <\)
(ii) \(A[k,k] := 1\)

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

<table>
<thead>
<tr>
<th>P₀</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>P₁</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₂</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₄</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>
</tr>
<tr>
<td>P₅</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>
</tr>
<tr>
<td>P₆</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>
</tr>
<tr>
<td>P₇</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>
</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

(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</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
</tbody>
</table>

(b) \( A[k,j] := (A[i,k] \times A[k,j]) \)
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</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</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</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
<tr>
<td>0</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>
</tr>
</tbody>
</table>

(d) \( A[i,j] := (A[i,k] \times A[k,j]) \)
for \( k < i < n \) and \( k < j < n \)
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] xU[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

Gravitational N-Body Problem

The N-body problem: Given n bodies in 3D space, determine the gravitational force $F$ between them at any given point in time.

$$ F = \frac{G m_a m_b}{r^2} $$

where $G$ is the gravitational constant, $r$ is the distance between the bodies, $m_a$ and $m_b$ are the masses of the bodies.
Exact N-body serial pseudo-code

- At each time \( t \), velocity \( v \) and position \( x \) of body \( i \) may change
- Real problem a bit more complicated than this

```c
For (t=0; t<max; t++)
    For (i=0; i<N; i++) {
        F= Force_routine(i);
        v[i]_new = v[i]+F*dt;
        x[i]_new=x[i]+v[i]_new*dt;
    }
For (i=0; i<nmax; i++) {
    x[i] = x[i]_new;
    v[i]=v[i]_new;
}
```

Parallel Code

- The algorithm is an \( O(N^2) \) algorithm (for one iteration) as each of the \( N \) bodies is influenced by each of the other \( N - 1 \) bodies.
- It is not feasible to use this direct algorithm for most interesting \( N \)-body problems where \( N \) is very large.
- The time complexity can be reduced using the observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster:

![Figure 4.18 Clustering distant bodies.](image)
**Exact N-body and Static Partitioning**

- Can parallelize n-body by tagging velocity and position for each body and updating bodies using correctly tagged information.

- This can be implemented as a data parallel algorithm. What is the worst-case complexity of complexity for a single iteration?

- How should we partition this?
  - Static partitioning can be a bad strategy for n-body problem.
  - Load can be very unbalanced for some configurations

---

**Improving the N-Body Code Complexity**

- Complexity of serial n-body algorithm very large: $O(n^2)$ for each iteration.

- Communication structure not local – each body must gather data from all other bodies.

- Most interesting problems are when $n$ is large – not feasible to use exact method for this

- Barnes-Hut algorithm is well-known approximation to exact n-body problem and can be efficiently parallelized
**Barnes-Hut Approximation**

- Barnes-Hut algorithm based on the observation that a cluster of distant bodies can be approximated as a single distant body
  - Total mass = aggregate of bodies in cluster
  - Distance to cluster = distance to center of mass of the cluster
- This clustering idea can be applied recursively

**Barnes-Hut Idea**

- Dynamic divide and conquer approach:
  - Each region (cube) of space divided into 8 subcubes
  - If subcube contains more than 1 body, it is recursively subdivided
  - If subcube contains no bodies, it is removed from consideration
- 2D example on right – each 2D region divided into 4 subregions
**Barnes-Hut idea**

- For 2D decomposition, result is a quadtree, pictured below.
- For 3D decomposition, result is an octtree

---

**Barnes Hut 3D Problem Pseudo-code**

For (t=0; t< tmax; t++) {
    Build octtree;
    Compute total mass and center;
    Traverse the tree, computing the forces
    Update the position and velocity of all bodies
}

**Notes:**
- Total mass and center of mass of each subcube stored at its root
- Tree traversal stops at a node when the clustering approximation can be used for a particular body
  - Need criteria for determining when bodies are in the same cluster
Barnes-Hut Complexity

- Partitioning is dynamic: Whole octtree must be reconstructed for each time step because bodies will have moved.
- Constructing tree can be done in $O(n \log n)$
- Computing forces can be done in $O(n \log n)$
- Barnes-Hut for one iteration is $O(n \log n)$ [compare to $O(n^2)$ for one iteration with exact solution]

Generalizing the Barnes-Hut approach

- Approach can be used for applications which repeatedly perform some calculation on particles/bodies/data indexed by position.

- Recursive Bisection:
  - Divide region in half so that particles are balanced each time
  - Map rectangular regions onto processors so that load is balanced
Barnes-Hut Algorithm

Orthogonal Recursive Bisection

- Example for a two-dimensional square area.
  - First, a vertical line is found that divides the area into two areas each with an equal number of bodies.
  - For each area, a horizontal line is found that divides it into two areas each with an equal number of bodies.
  - This is repeated until there are as many areas as processors, and then one processor is assigned to each area.
Recursive Bisection Programming Issues

- How do we keep track of the regions mapped to each processor?
- What should the density of each region be? [granularity!]
- What is the complexity of performing the partitioning? How often should we repartition to optimize the load balance?
- How can locality of communication or processor configuration be leveraged?