Lecture 8: Principles of Parallel Algorithm Design

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

Topics

- Introduction
- Programming on shared memory system (Chapter 7)
  - **OpenMP**

☛ Principles of parallel algorithm design (Chapter 3)

- Programming on large scale systems (Chapter 6)
  - **MPI (point to point and collectives)**
  - Introduction to PGAS languages, UPC and Chapel
- Analysis of parallel program executions (Chapter 5)
  - **Performance Metrics for Parallel Systems**
    - **Execution Time, Overhead, Speedup, Efficiency, Cost**
  - **Scalability of Parallel Systems**
  - **Use of performance tools**

Topics

- Programming on shared memory system (Chapter 7)
  - ***Cilk/Cilkplus and OpenMP Tasking***
  - ***PThread, mutual exclusion, locks, synchronizations***
- Parallel architectures and hardware
  - **Parallel computer architectures**
  - **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
  - Introduction to offloading model in OpenMP

“parallel and for” OpenMP Constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel shared (a, b)
{
      int id, i, Nthrds, istart, iend;
      id = omp_get_thread_num();
      Nthrds = omp_get_num_threads();
      istart = id * N / Nthrds;
      iend = (id+1) * N / Nthrds;
      for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a worksharing for construct

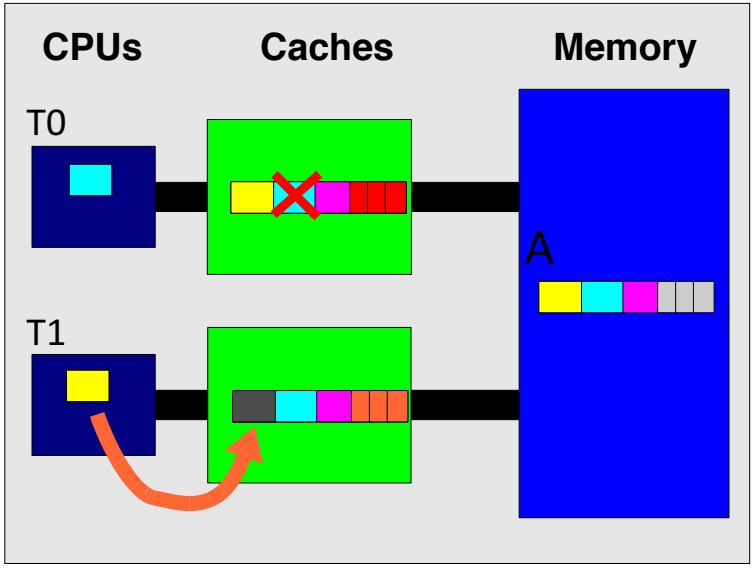
```
#pragma omp parallel shared (a, b) private (i)
#pragma omp for schedule(static)
      for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP Best Practices

```
#pragma omp parallel private(i)
{
  #pragma omp for nowait
  for(i=0;i<n;i++)
    a[i] +=b[i];
  #pragma omp for nowait
  for(i=0;i<n;i++)
    c[i] +=d[i];
  #pragma omp barrier
  #pragma omp for nowait reduction(+:sum)
  for(i=0;i<n;i++)
   sum += a[i] + c[i];
}
```

False-sharing in OpenMP and Solution

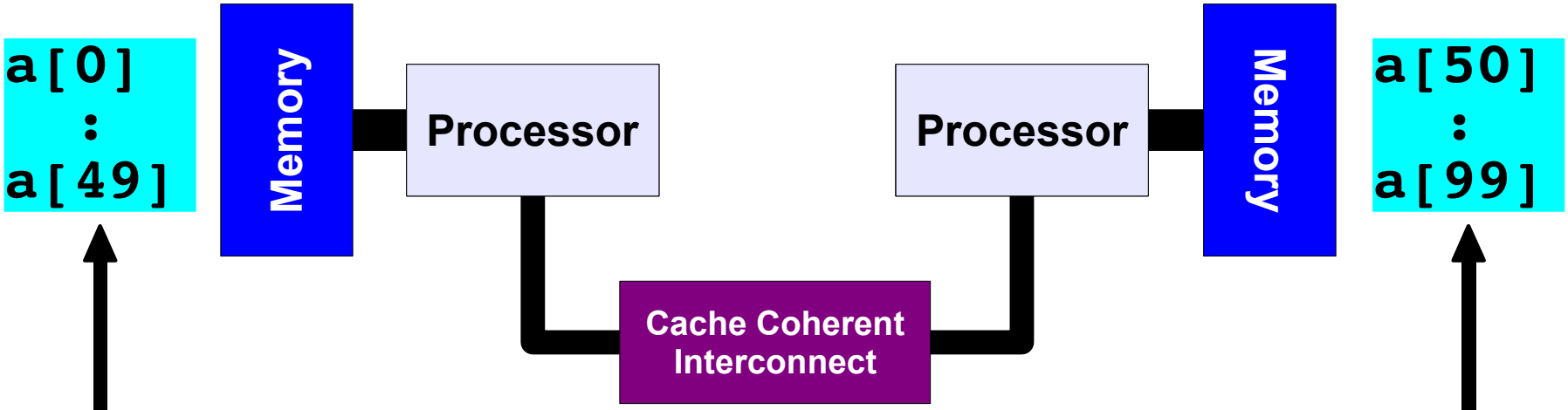
- False sharing
  - When at least one thread write to a cache line while others access it
    - Thread 0: = A[1] (read)
    - Thread 1: A[0] = … (write)
- Solution: use array padding




```
int a[max_threads];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
    a[i] +=i;
```

```
int a[max_threads][cache_line_size];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
    a[i][0] +=i;
```

NUMA First-touch




```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

SPMD Program Models in OpenMP

- **SPMD (Single Program, Multiple Data) for parallel regions**
  - All threads of the parallel region execute the same code
  - Each thread has unique ID

- Use the thread ID to diverge the execution of the threads
  - Different thread can follow different paths through the same code

```
if(my_id == x) {   }
else {    }
```

- SPMD is by far the most commonly used pattern for structuring parallel programs
  - MPI, OpenMP, CUDA, etc

Overview: Algorithms and Concurrency (part 1)

☛**Introduction to Parallel Algorithms**

- Tasks and Decomposition
- Processes and Mapping

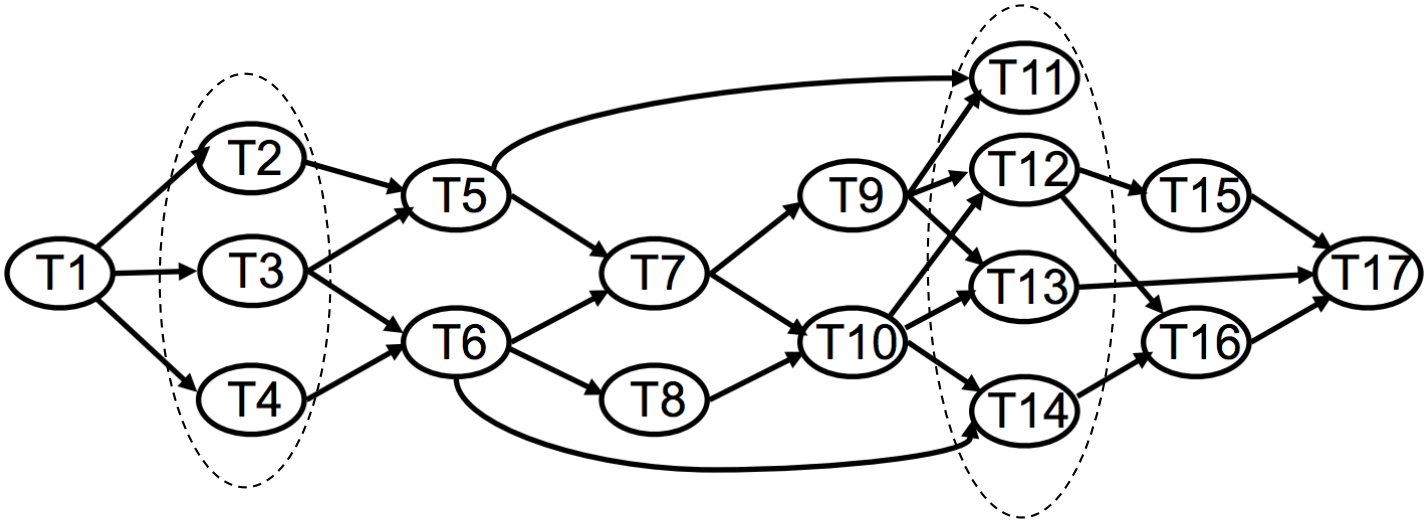
- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Hybrid Decomposition
- Characteristics of Tasks and Interactions
  - Task Generation, Granularity, and Context
  - Characteristics of Task Interactions.

Overview: Concurrency and Mapping (part 2)

- Mapping Techniques for Load Balancing
  - Static and Dynamic Mapping
- Methods for Minimizing Interaction Overheads
  - Maximizing Data Locality
  - Minimizing Contention and Hot-Spots
  - Overlapping Communication and Computations
  - Replication vs. Communication
  - Group Communications vs. Point-to-Point Communication
- Parallel Algorithm Design Models
  - Data-Parallel, Work-Pool, Task Graph, Master-Slave, Pipeline, and Hybrid Models

Decomposition, Tasks, and Dependency Graphs

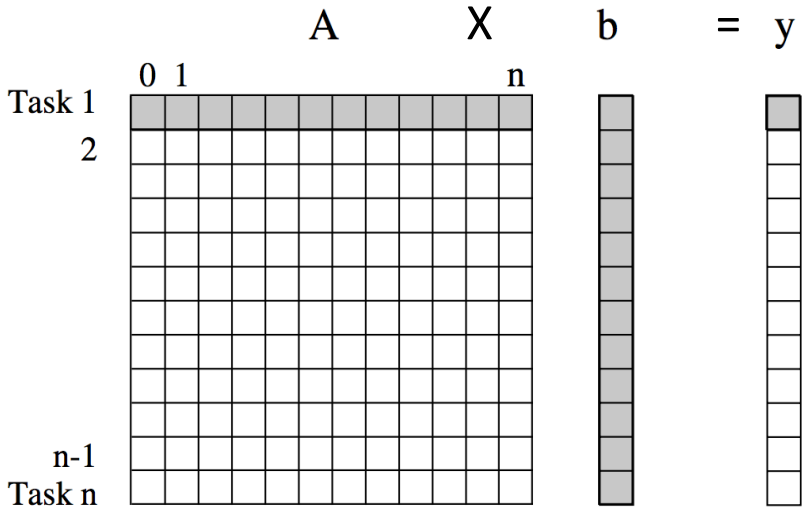
- Decompose work into tasks that can be executed concurrently
- Decomposition could be in many different ways.
- Tasks may be of same, different, or even indeterminate sizes.
- Task dependency graph:
  - node = task
  - edge = control dependence, output-input dependency
  - No dependency == parallelism



Example: *Dense* Matrix Vector Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax+by+cz+dw \\ ex+fy+gz+hw \\ ix+jy+kz+lw \\ mx+ny+oz+pw \end{bmatrix}$$

```
REAL A[n][n], b[n], y[n]; int i, j;
for (i = 0; i < n; i++) {
   sum = 0.0;
   for (j = 0; j < n; j++)
      sum += A[i][j] * b[j];
   c[i] = sum;
}
```



- Computation of each element of output vector y is independent
- Decomposed into n tasks, one per element in y → Easy
- Observations
  - Each task only reads one row of A, and writes one element of y
  - All tasks share vector b (the shared data)
  - No control dependencies between tasks
  - All tasks are of the same size in terms of number of operations.

Example: Database Query Processing

Consider the execution of the query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND
(COLOR = ``GREEN'' OR COLOR = ``WHITE”)

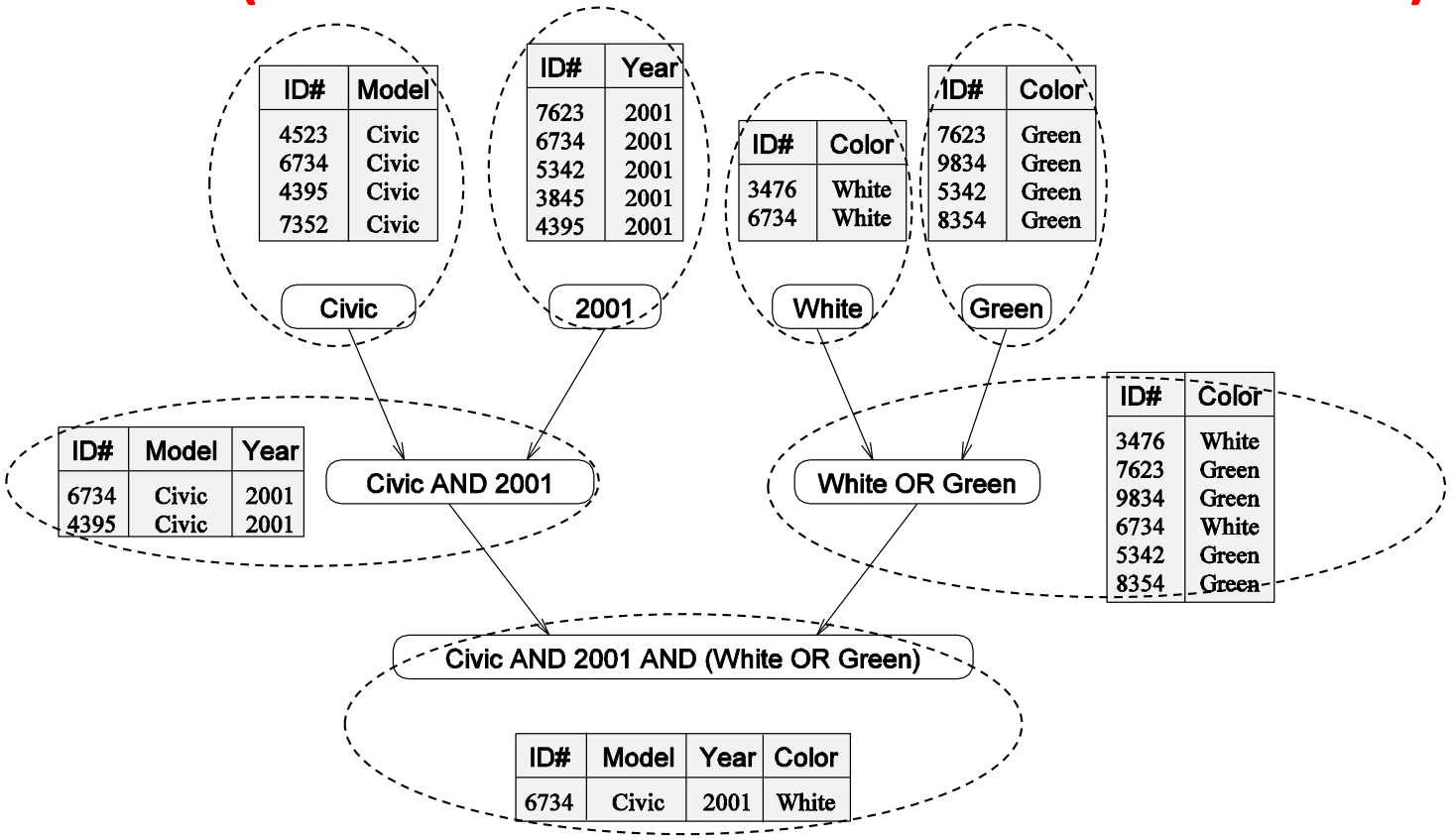
on the following table:

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

Example: Database Query Processing

- Tasks: Each task search the whole table for entries that satisfy one predicate
  - Results: A list of entries
- Edges: output of one task serves as input to the next

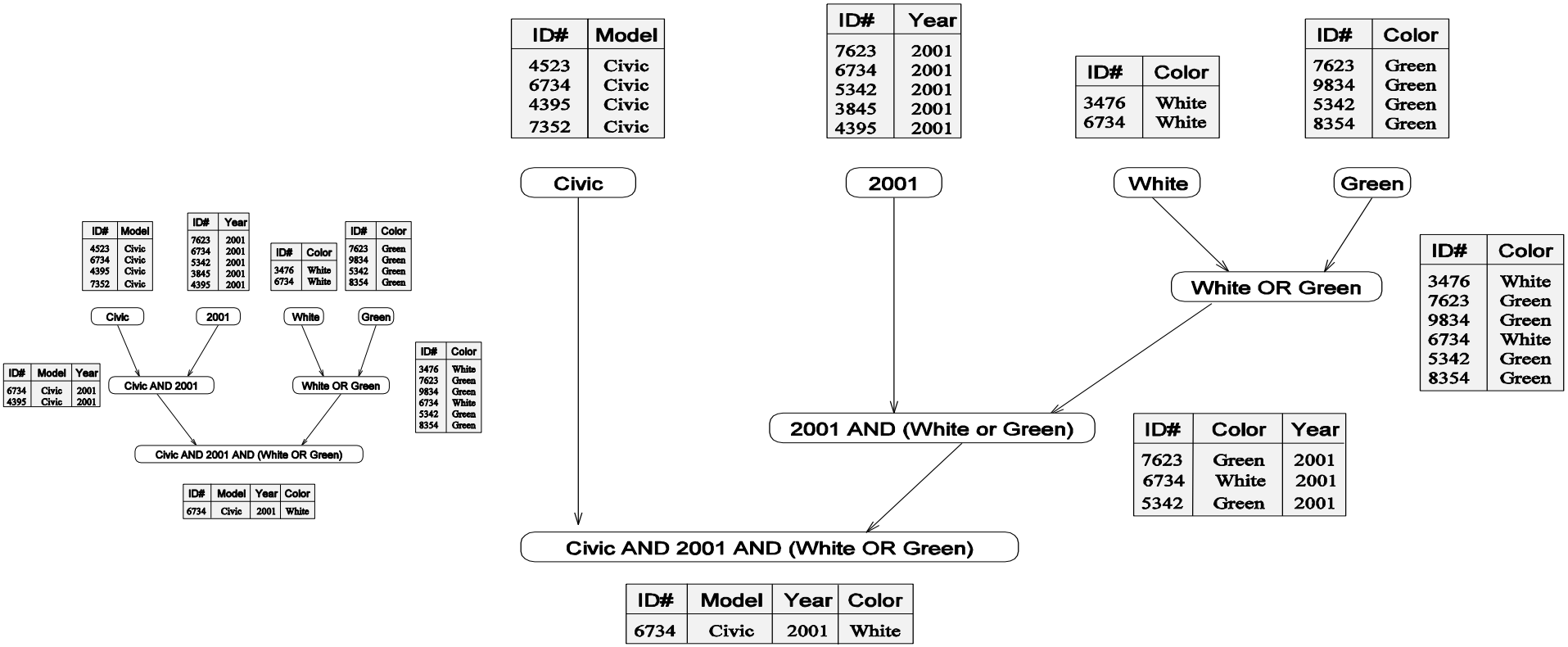
MODEL = ``CIVIC'' AND YEAR = 2001 AND (COLOR = ``GREEN'' OR COLOR = ``WHITE")



Example: Database Query Processing

- An alternate decomposition

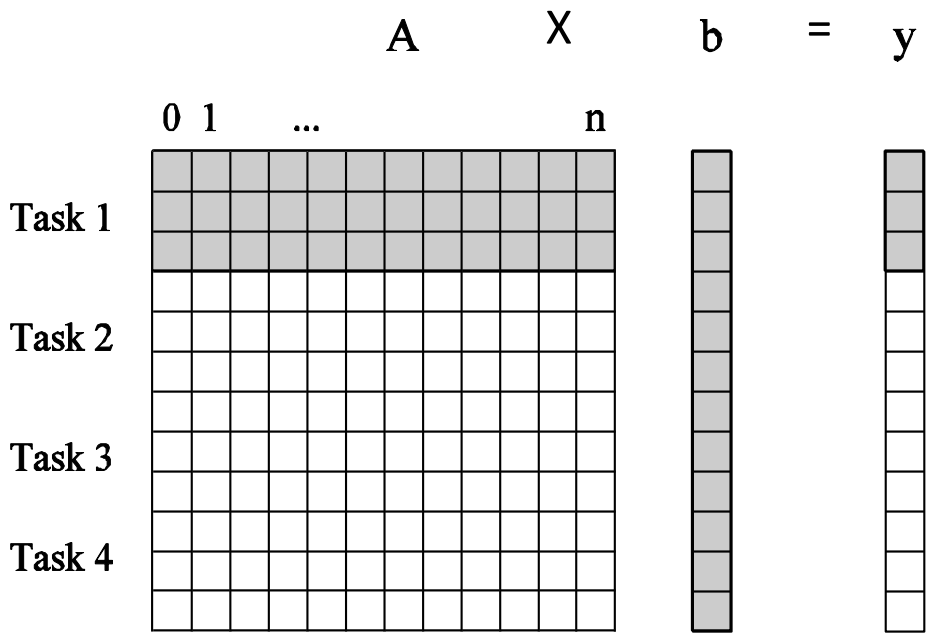
  MODEL = ``CIVIC'' AND YEAR = 2001 AND
  (COLOR = ``GREEN'' OR COLOR = ``WHITE)



- Different decompositions may yield different parallelism and performance

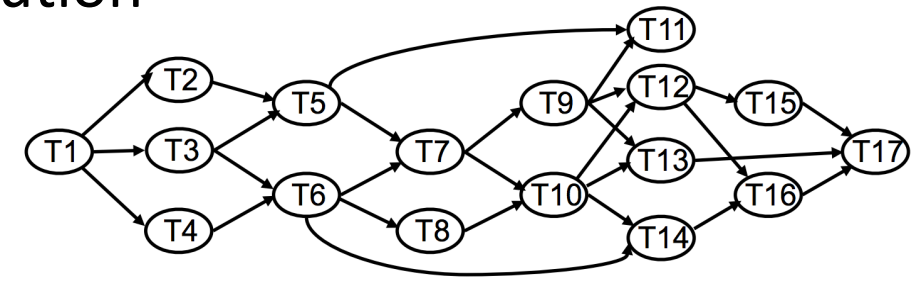
Granularity of Task Decompositions

- **Granularity is task size (amount of computation)**
  - Depending on the number of tasks for the same problem size
- Fine-grained decomposition = large number of tasks
- Coarse grained decomposition = small number of tasks
- Granularity for dense matrix-vector product
  - fine-grain: each task computes an individual element in y
  - coarser-grain: each task computes 3 elements in y



Degree of Concurrency

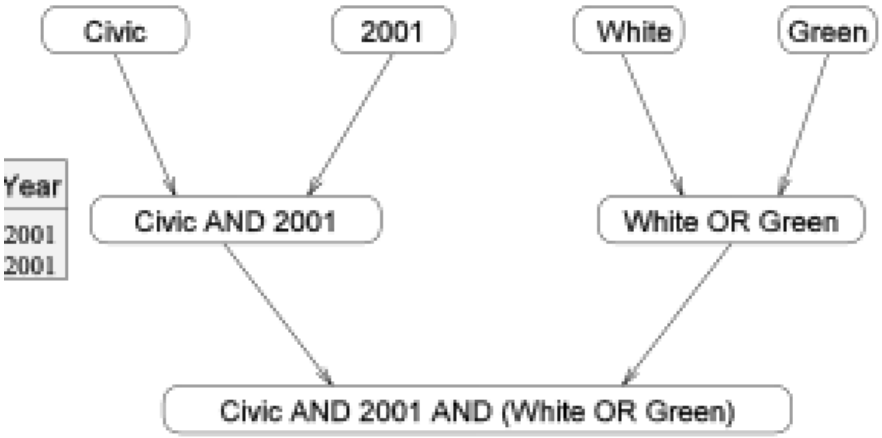
- Definition: the number of tasks that can be executed in parallel
- May change over program execution

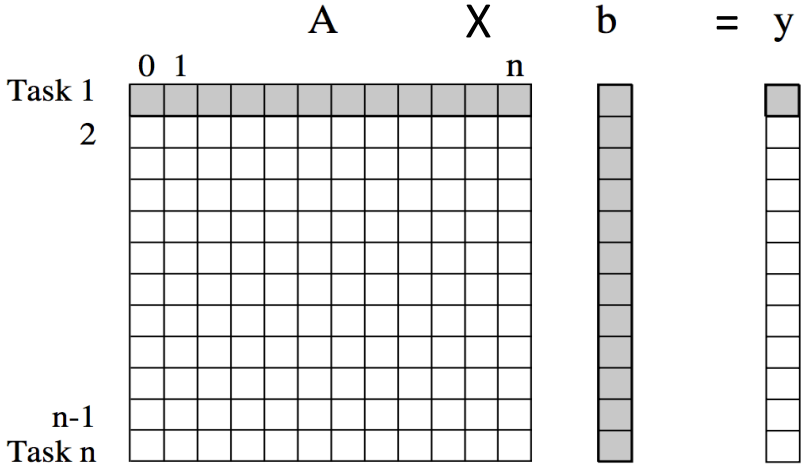


- Metrics
  - *Maximum degree of concurrency*
    - Maximum number of concurrent tasks at any point during execution.
  - *Average degree of concurrency*
    - The average number of tasks that can be processed in parallel over the execution of the program
    - **Speedup: serial_execution_time/parallel_execution_time**
- Inverse relationship of degree of concurrency and task granularity
  - Task granularity ↑(less tasks), degree of concurrency ↓
  - Task granularity ↓(more tasks), degree of concurrency ↑

Examples: Degree of Concurrency

- Maximum degree of concurrency
- Average degree of concurrency

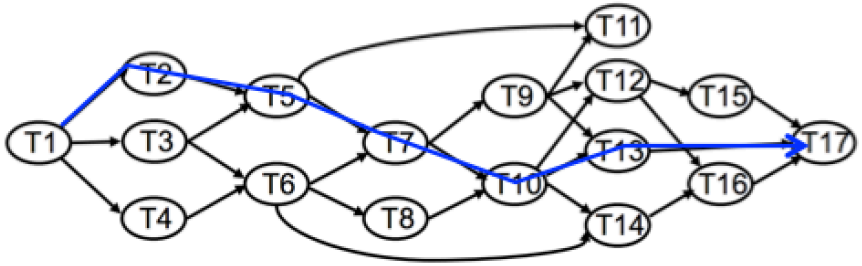




- Database query
  - Max: 4
  - Average: 7/3 (each task takes the same time)
- Matrix-vector multiplication
  - Max: n
  - Average: n

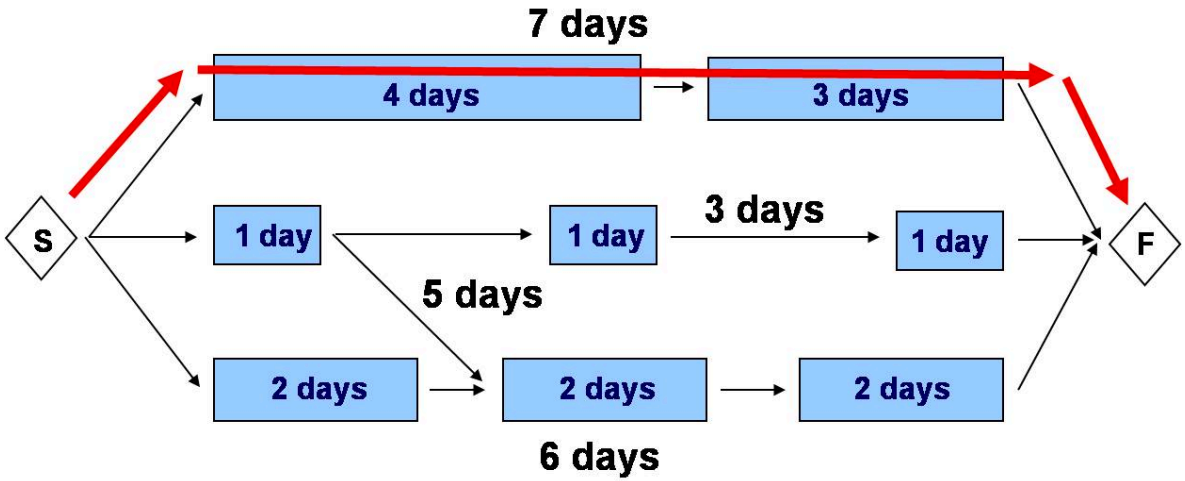
Critical Path Length

- **A directed path:** a sequence of tasks that must be serialized
  - Executed one after another



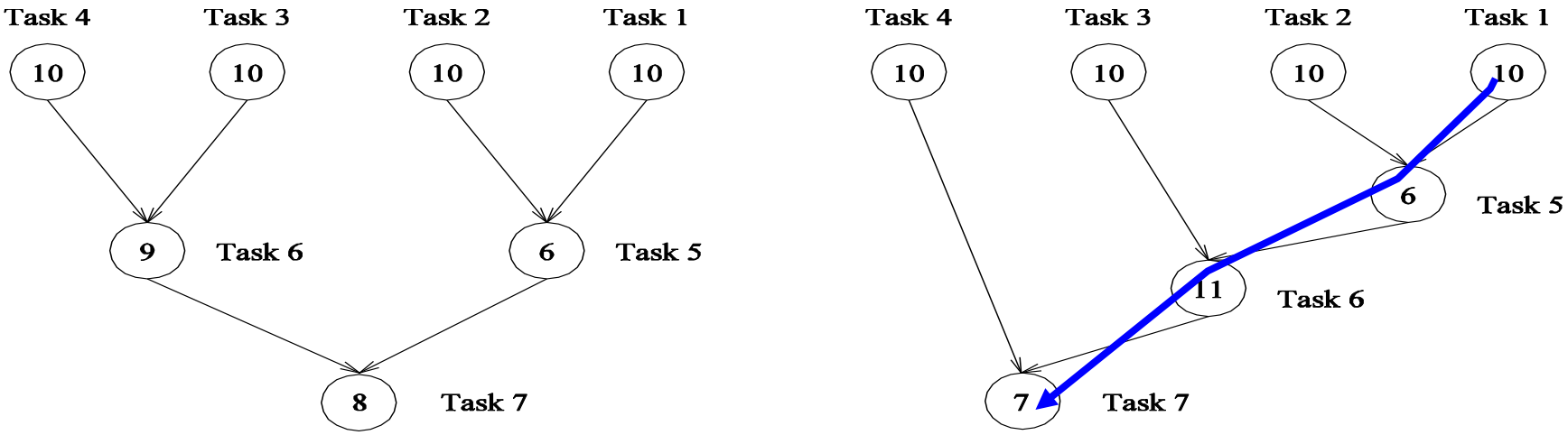
- Critical path:
  - The longest weighted path throughout the graph
- Critical path length: shortest time in which the program can be finished
  - Lower bound on parallel execution time

A building project



Critical Path Length and Degree of Concurrency

Database query task dependency graph



Questions:

What are the tasks on the critical path for each dependency graph?

What is the shortest parallel execution time?

How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

What is the average parallelism (average degree of concurrency)?

Total amount of work/(critical path length)

2.33 (63/27) and 1.88 (64/34)

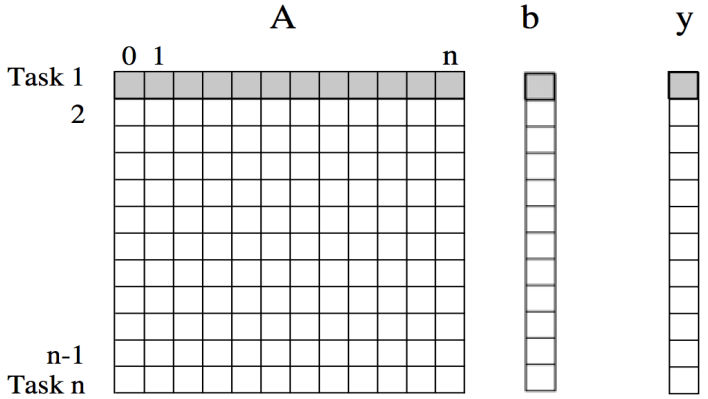
Limits on Parallel Performance

- What bounds parallel execution time?
  - minimum task granularity
    - e.g. dense matrix-vector multiplication ≤ n^2 concurrent tasks
  - fraction of application work that can't be parallelized
    - more about Amdahl's law in a later lecture …
  - **dependencies between tasks**
  - **parallelization overheads**
    - e.g., cost of communication between tasks

- Metrics of parallel performance
  - T1: sequential execution time, Tp parallel execution time on p processors/cores/threads
  - **speedup = T1/Tp**
  - **parallel efficiency = T1/(p*Tp)**

Task Interaction Graphs

- Tasks generally exchange data with others
  - example: dense matrix-vector multiply
    - If b is not replicated in all tasks: each task has some, but not all
    - Tasks will have to communicate elements of b

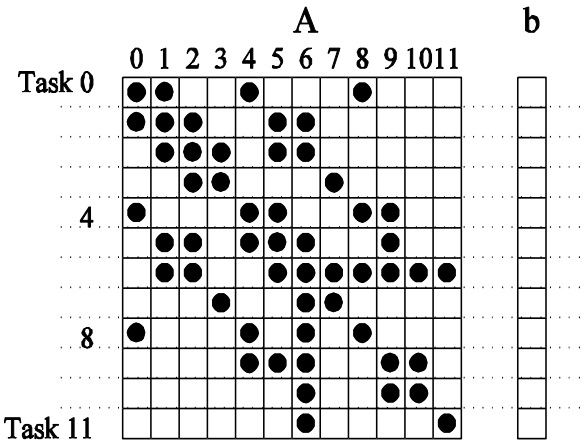


- Task interaction graph
  - node = task
  - edge = interaction or data exchange
- Task interaction graphs vs. task dependency graphs
  - Task dependency graphs represent *control dependences*
  - Task interaction graphs represent *data dependences*

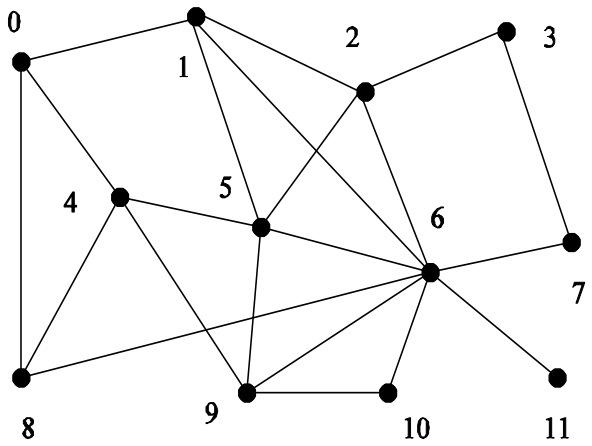
Task Interaction Graphs: An Example

Sparse matrix vector multiplication: *A* x *b*

- Computation of each result element = an independent task.
- Only non-zero elements of sparse matrix ***A*** participate in computation.
- **If partition *b* across tasks, i.e. task Ti has b[i] only**
  - The task interaction graph of the computation = graph of the matrix ***A***
    - A is the adjacent matrix of the graph



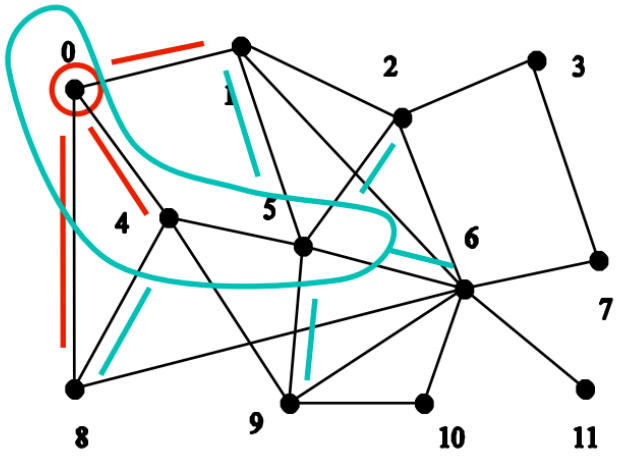
(a)

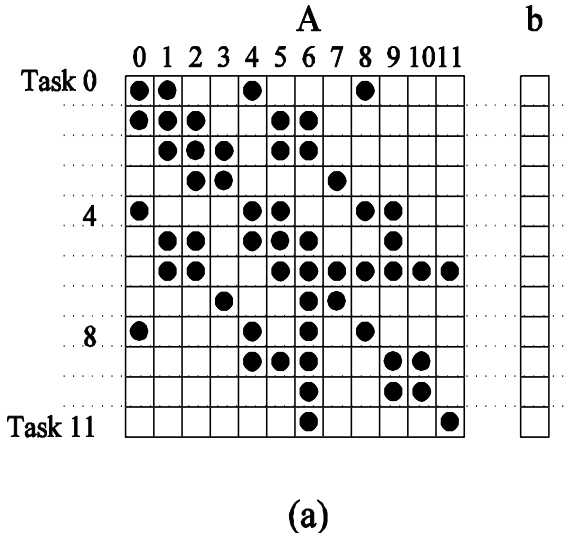


(b)

Task Interaction Graphs, Granularity, and Communication

- Finer task granularity → more overhead of task interactions
  - Overhead as a ratio of useful work of a task
- Example: sparse matrix-vector product interaction graph





(a)

- Assumptions:
  - each dot (A[i][j]*b[j]) takes unit time to process
  - each communication (edge) causes an overhead of a unit time
- If node 0 is a task: communication = 3; computation = 4
- If nodes 0, 4, and 5 are a task: communication = 5; computation = 15
  - **coarser-grain decomposition → smaller communication/computation ratio (3/4 vs 5/15)**

Processes and Mapping

- Generally
  - # of tasks >= # processing elements (PEs) available
  - parallel algorithm must map tasks to ***processes***
- Mapping: aggregate tasks into processes
  - Process/thread = processing or computing agent that performs work
  - assign collection of tasks and associated data to a process
- An PE, e.g. a core/thread, has its system abstraction
  - Not easy to bind tasks to physical PEs, thus, more layers at least conceptually from PE to task mapping
  - Process in MPI, thread in OpenMP/pthread, etc
    - The overloaded terms of processes and threads
  - Task → processes → OS processes → CPU → cores
  - **For the sake of simplicity, processes = PEs/cores**

Processes and Mapping

- Mapping of tasks to processes is critical to the parallel performance of an algorithm.
- On what basis should one choose mappings?
  - Task dependency graph
  - Task interaction graph
- Task dependency graphs
  - To ensure equal spread of work across all processes at any point
    - minimum idling
    - optimal load balance
- Task interaction graphs
  - To minimize interactions
    - Minimize communication and synchronization

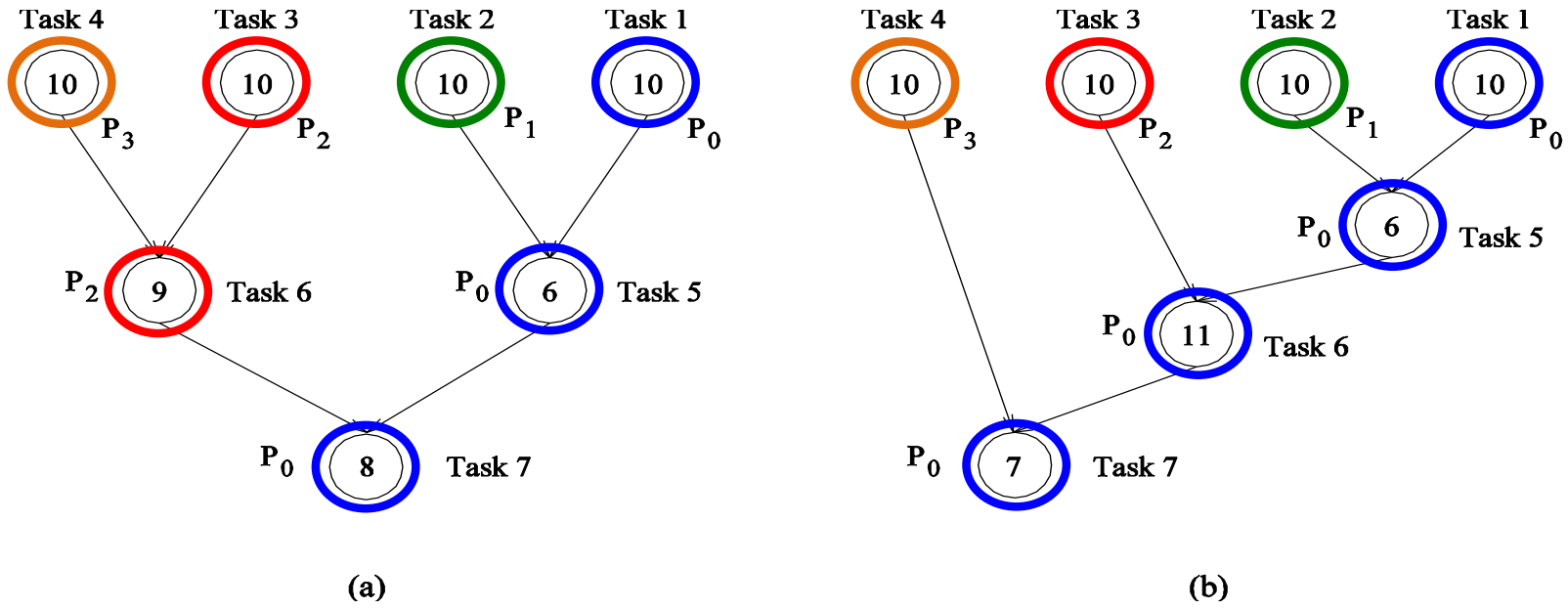
Processes and Mapping

A good mapping must minimize parallel execution time by:

- Mapping independent tasks to different processes
  - Maximize concurrency
- Tasks on critical path have high priority of being assigned to processes
- Minimizing interaction between processes
  - mapping tasks with dense interactions to the same process.

- Difficulty: these criteria often conflict with each other
  - E.g. No decomposition, i.e. one task, minimizes interaction but no speedup at all!

Processes and Mapping: Example



Example: mapping database queries to processes

- Consider the dependency graphs in levels
  - no nodes in a level depend upon one another
    - Assign all tasks within a level to different processes

Overview: Algorithms and Concurrency

- Introduction to Parallel Algorithms
  - **Tasks and Decomposition**
  - **Processes and Mapping**

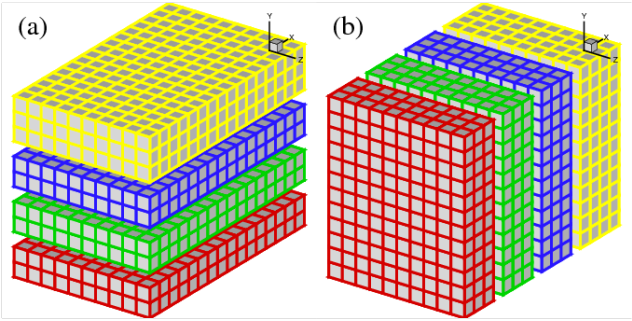
☛ **Decomposition Techniques**

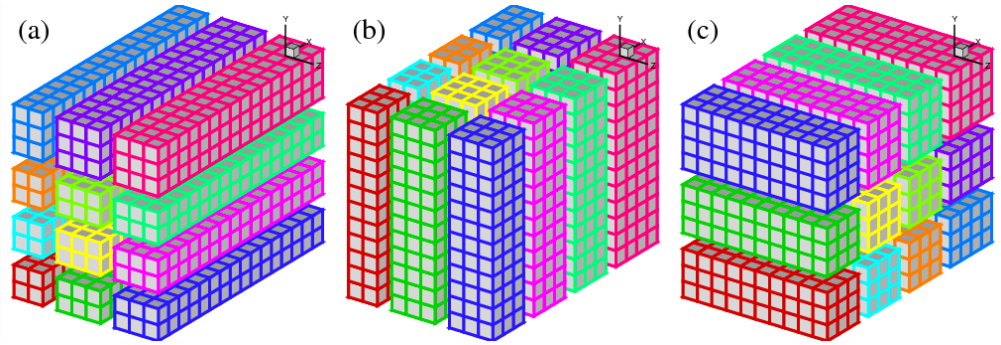
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Hybrid Decomposition

- Characteristics of Tasks and Interactions
  - Task Generation, Granularity, and Context
  - Characteristics of Task Interactions.

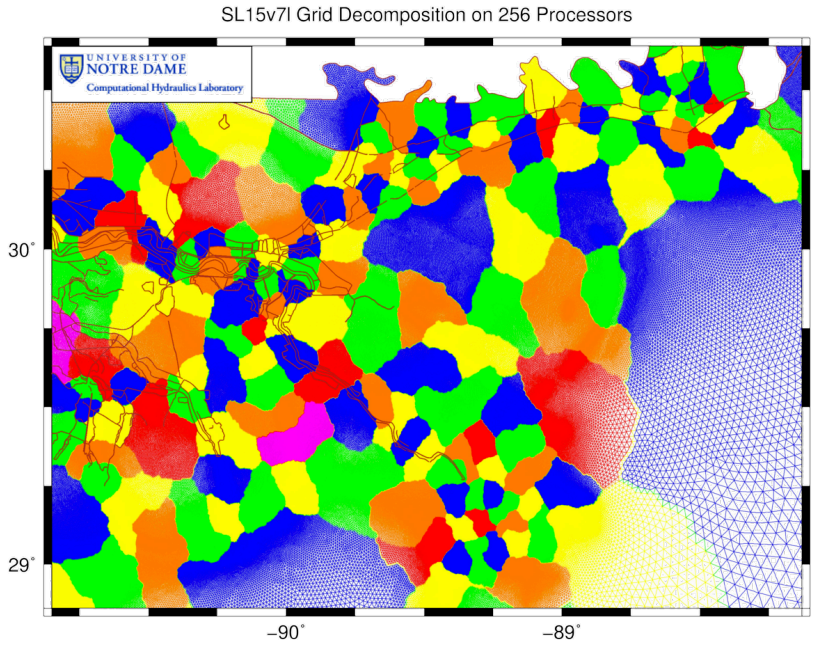
Decomposition Techniques

So how does one decompose a task into various subtasks?





- No single recipe that works for all problems
- Practically used techniques
  - **Recursive decomposition**
  - **Data decomposition**
  - Exploratory decomposition
  - Speculative decomposition

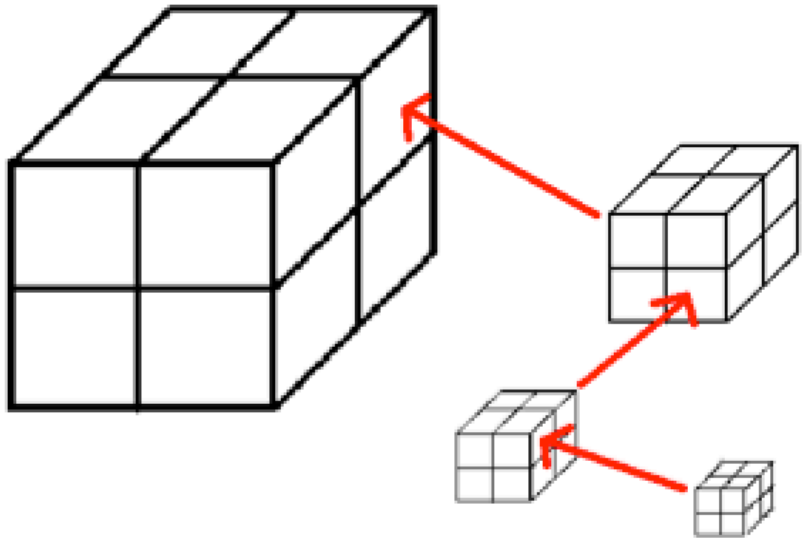


Recursive Decomposition

Generally suited to problems solvable using the divide-and-conquer strategy

Steps:

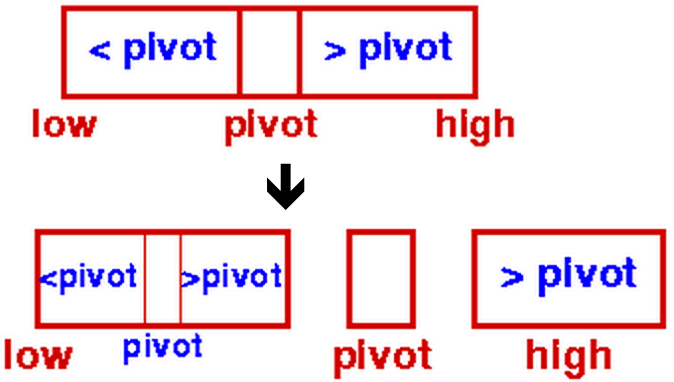
1. decompose a problem into a set of sub-problems
2. recursively decompose each sub-problem
3. stop decomposition when minimum desired granularity reached

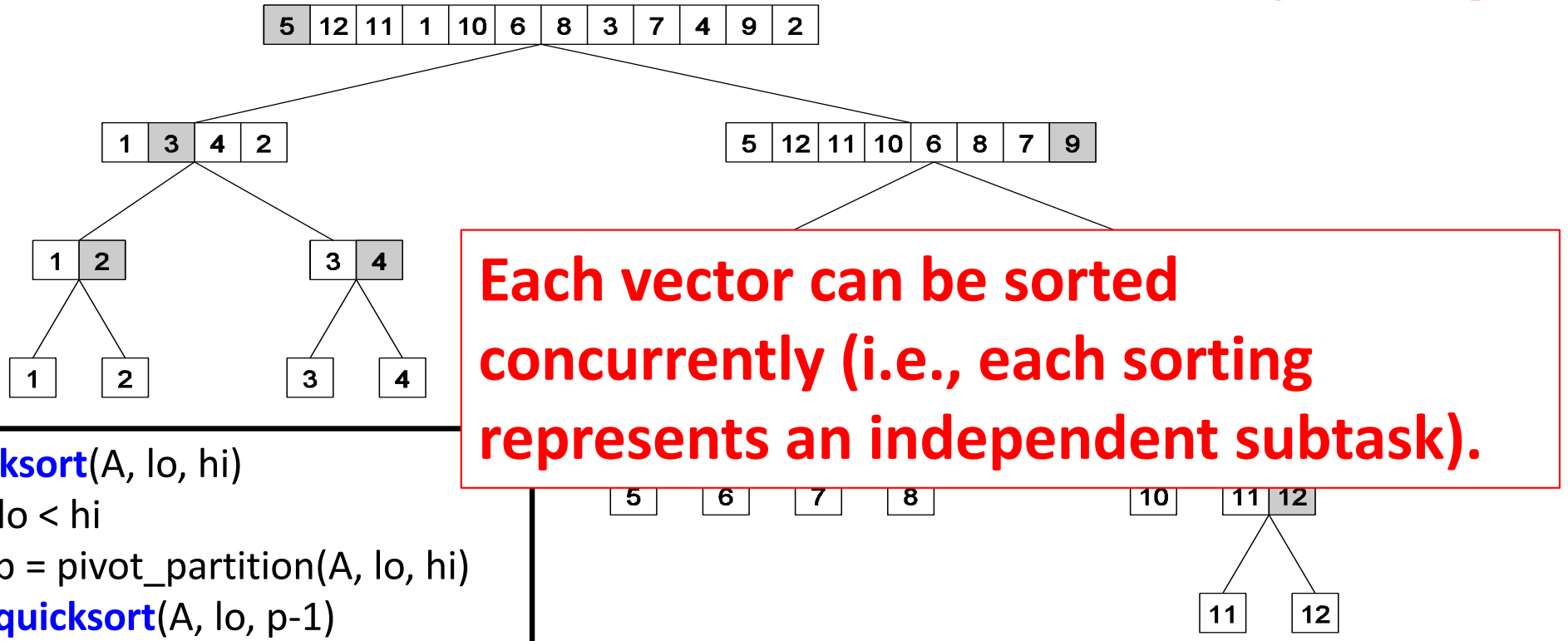


Recursive Decomposition: Quicksort

At each level and for each vector

1. Select a pivot
2. Partition set around pivot
3. Recursively sort each subvector





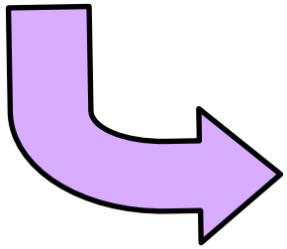
Each vector can be sorted concurrently (i.e., each sorting represents an independent subtask).

```
quicksort(A, lo, hi)
   if lo < hi
      p = pivot_partition(A, lo, hi)
      quicksort(A, lo, p-1)
      quicksort(A, p+1, hi)
```

Recursive Decomposition: Min

Finding the minimum in a vector using divide-and-conquer

```
procedure SERIAL_MIN (A, n)
    min = A[0];
    for i := 1 to n − 1 do
        if (A[i] < min) min := A[i];
    return min;
```



```
procedure RECURSIVE_MIN (A, n)
    if ( n = 1 ) then min := A [0] ;
    else
        lmin := RECURSIVE_MIN (A, n/2 );
        rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
        if (lmin < rmin) then min := lmin;
        else min := rmin;
    return min;
```

Applicable to other associative operations, e.g. sum, AND …

Known as reduction operation

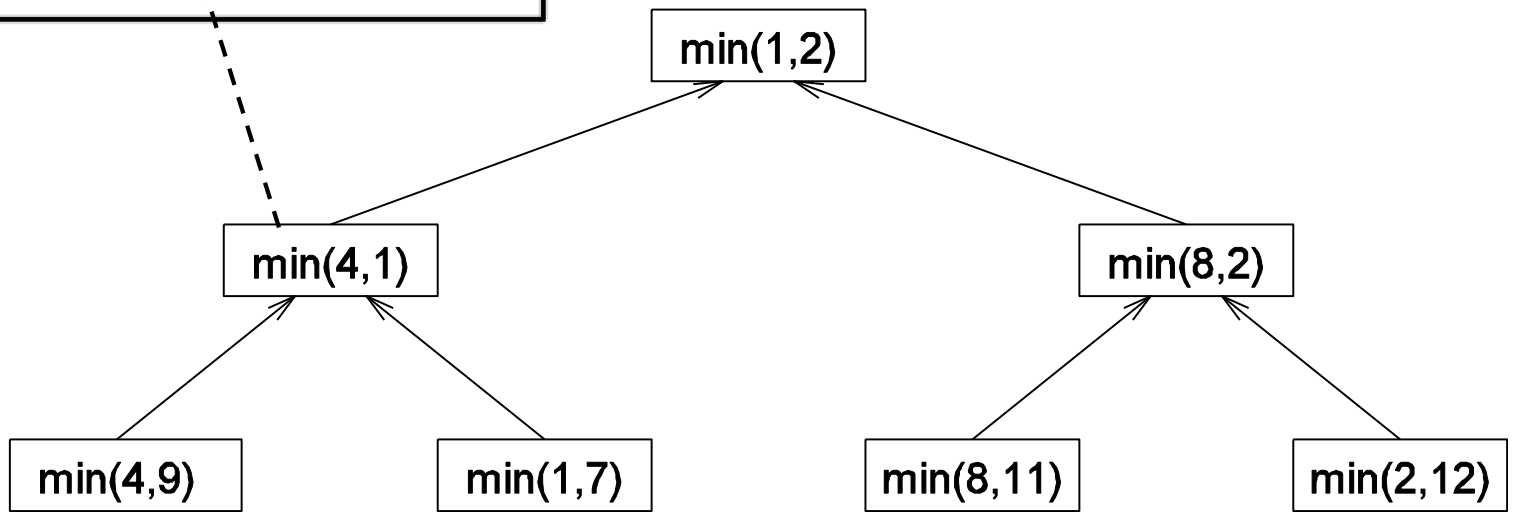
Recursive Decomposition: Min

finding the minimum in set {4, 9, 1, 7, 8, 11, 2, 12}.

```
procedure RECURSIVE_MIN (A, n)
    if ( n = 1 ) then min := A [0] ;
    else
        lmin := RECURSIVE_MIN (A, n/2 );
        rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
        if (lmin < rmin) then min := lmin;
        else min := rmin;
    return min;
```

Task dependency graph:

- **RECURSIVE_MIN** forms the binary tree
- min finishes and closes
- Parallel in the same level



Fib with OpenMP Tasking

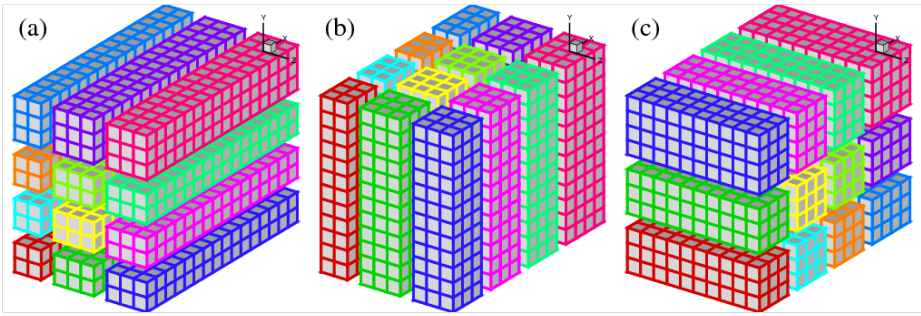
- **Task completion** occurs when the task reaches the end of the task region code
- Multiple tasks joined to complete through the use of **task synchronization constructs**
  - **taskwait**
  - **barrier** construct

- **taskwait** constructs:
  - #pragma omp taskwait
  - !$omp taskwait

```
int fib(int n) {
   int x, y;
   if (n < 2)  return n;
   else {
        #pragma omp task shared(x)
        x = fib(n-1);
        #pragma omp task shared(y)
        y = fib(n-2);
        #pragma omp taskwait
        return x + y;
   }
}
```

Data Decomposition
-- The most commonly used approach

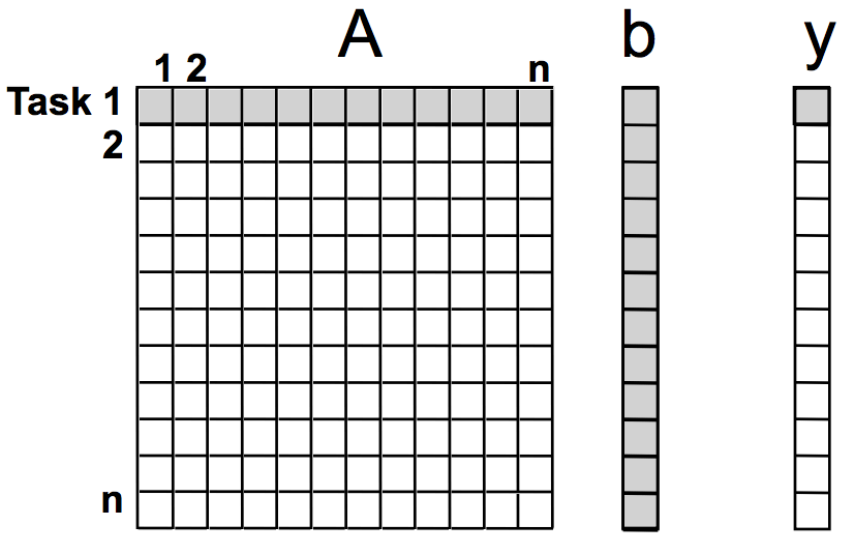
- Steps:
  1. Identify the data on which computations are performed.
  2. Partition this data across various tasks.
     - Partitioning induces a decomposition of the problem, i.e. computation is partitioned
- Data can be partitioned in various ways
  - Critical for parallel performance
- Decomposition based on
  - output data
  - input data
  - input + output data
  - intermediate data



Output Data Decomposition

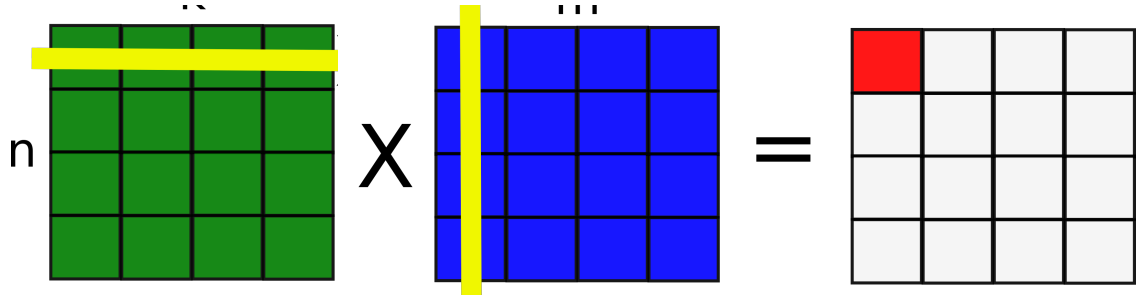
- Each element of the output can be computed independently of others
  - simply as a function of the input.
- A natural problem decomposition

Example:
dense matrix-vector multiply



Output Data Decomposition: Matrix Multiplication

multiplying two *n* x *n* matrices *A* and *B* to yield matrix *C*



The output matrix ***C*** can be partitioned into four tasks:

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

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: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Other task decompositions possible

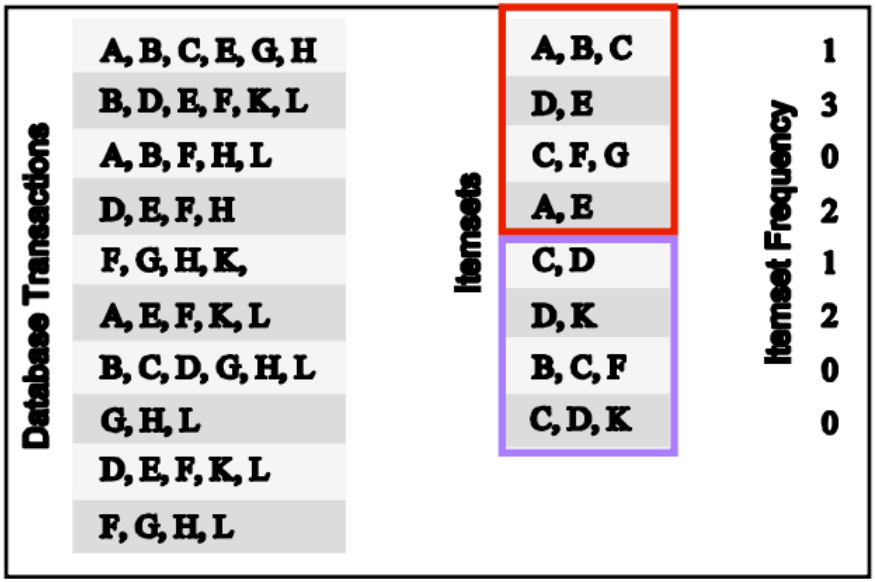
Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previus foil, with identical output data distribution, we can derive the following two (other) decompositions:

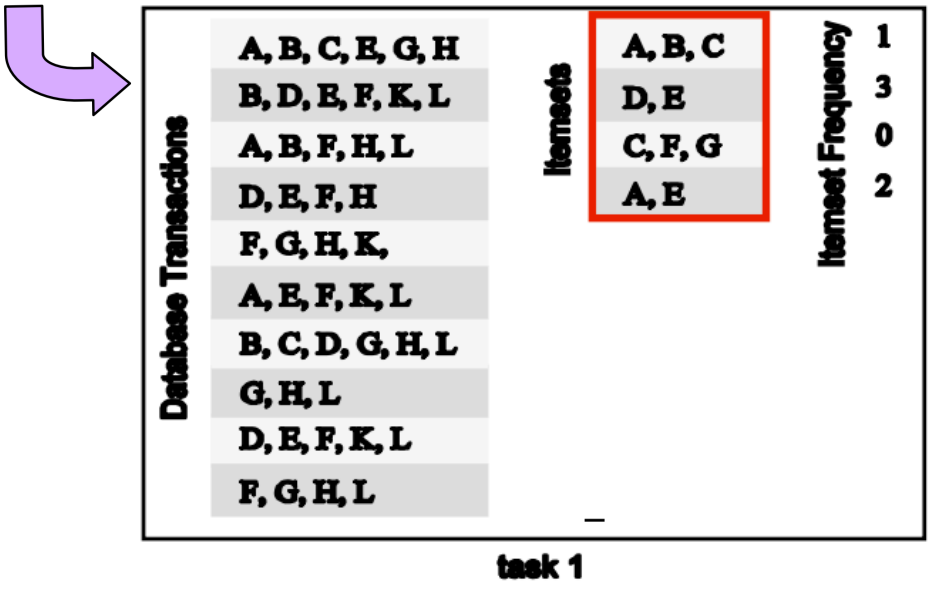
Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1} B_{1,1}$	Task 1: $C_{1,1} = A_{1,1} B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Task 3: $C_{1,2} = A_{1,1} B_{1,2}$	Task 3: $C_{1,2} = A_{1,2} B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Task 5: $C_{2,1} = A_{2,1} B_{1,1}$	Task 5: $C_{2,1} = A_{2,2} B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Task 7: $C_{2,2} = A_{2,1} B_{1,2}$	Task 7: $C_{2,2} = A_{2,1} B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

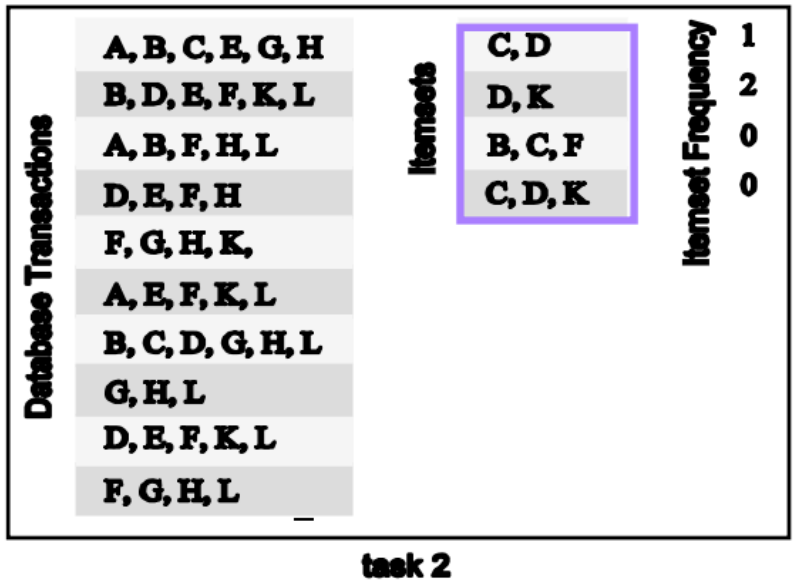
Output Data Decomposition: Example

Count the frequency of item sets in database transactions



- Decompose the item sets to count
  - each task computes total count for each of its item sets
  - append total counts for item sets to produce total count result





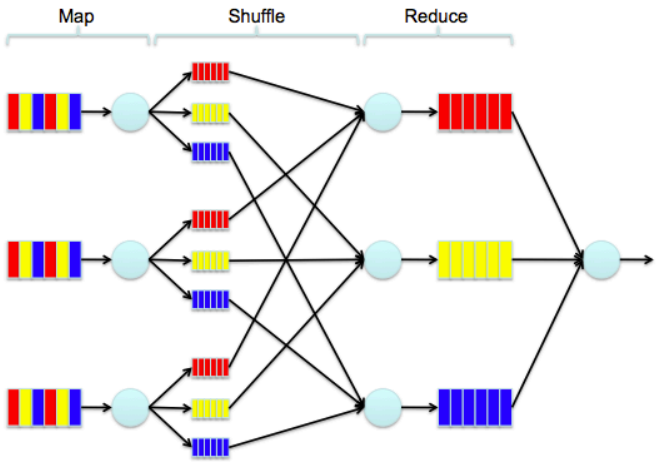
Output Data Decomposition: Example

From the previous example, the following observations can be made:

- If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

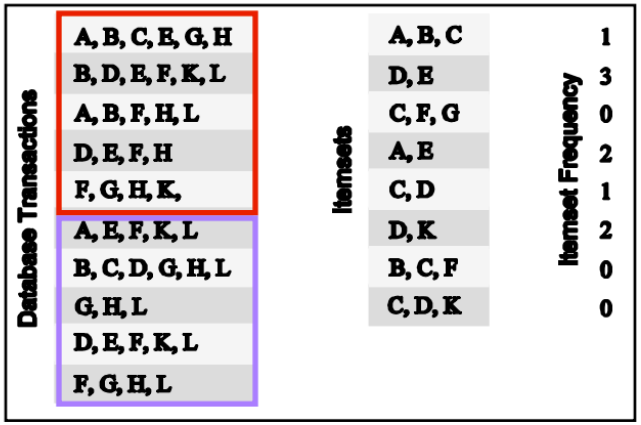
Input Data Partitioning

- Generally applicable if each output can be naturally computed as a function of the input.
- In many cases, this is the only natural decomposition if the output is not clearly known a-priori
  - e.g., the problem of finding the minimum, sorting.
- A task is associated with each input data partition
  - The task performs computation with its part of the data.
  - Subsequent processing combines these partial results.
- MapReduce

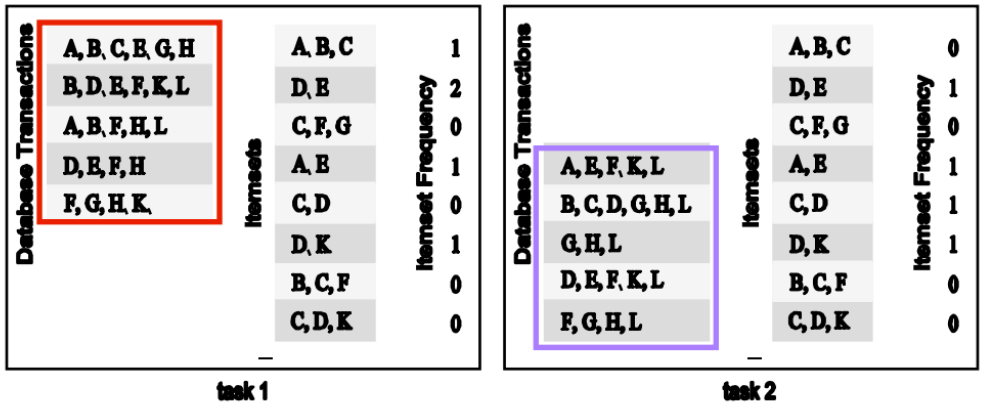


Input Data Partitioning: Example

Count the frequency of item sets in database transactions



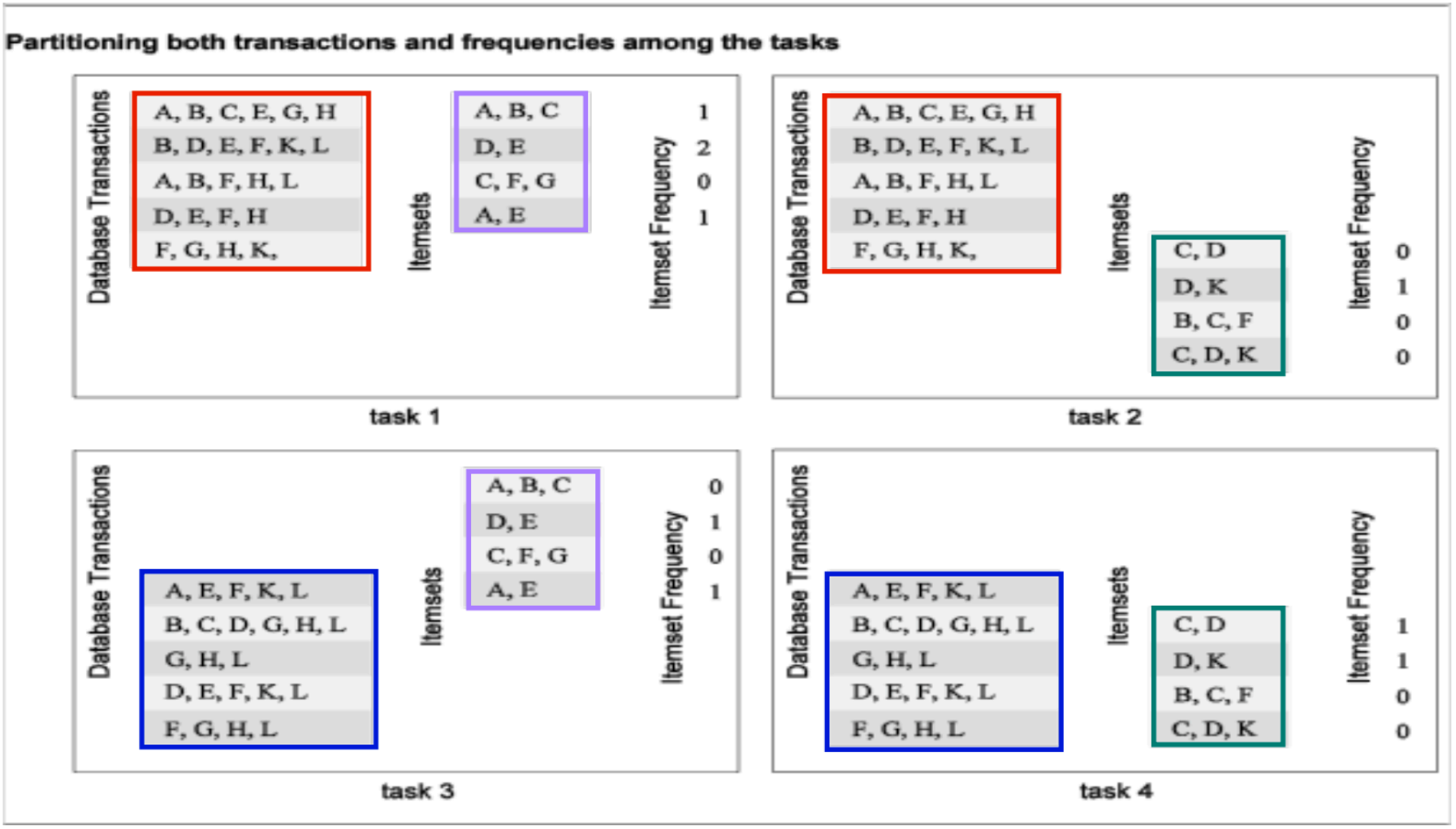
- **Partition computation by partitioning the set of transactions**
  - **—a task computes a local count for each item set for its transactions**



  - **—sum local count vectors for item sets to produce total count vector**

Partitioning Input *and* Output Data

- Partition on both input and output for more concurrency
- Example: item set counting



Histogram

- Parallelizing Histogram using OpenMP in Assignment 2
  - Similar as count frequency

Intermediate Data Partitioning

- If computation is a sequence of transforms
  - from input data to output data, e.g. image processing workflow
- Can decompose based on data for intermediate stages

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

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

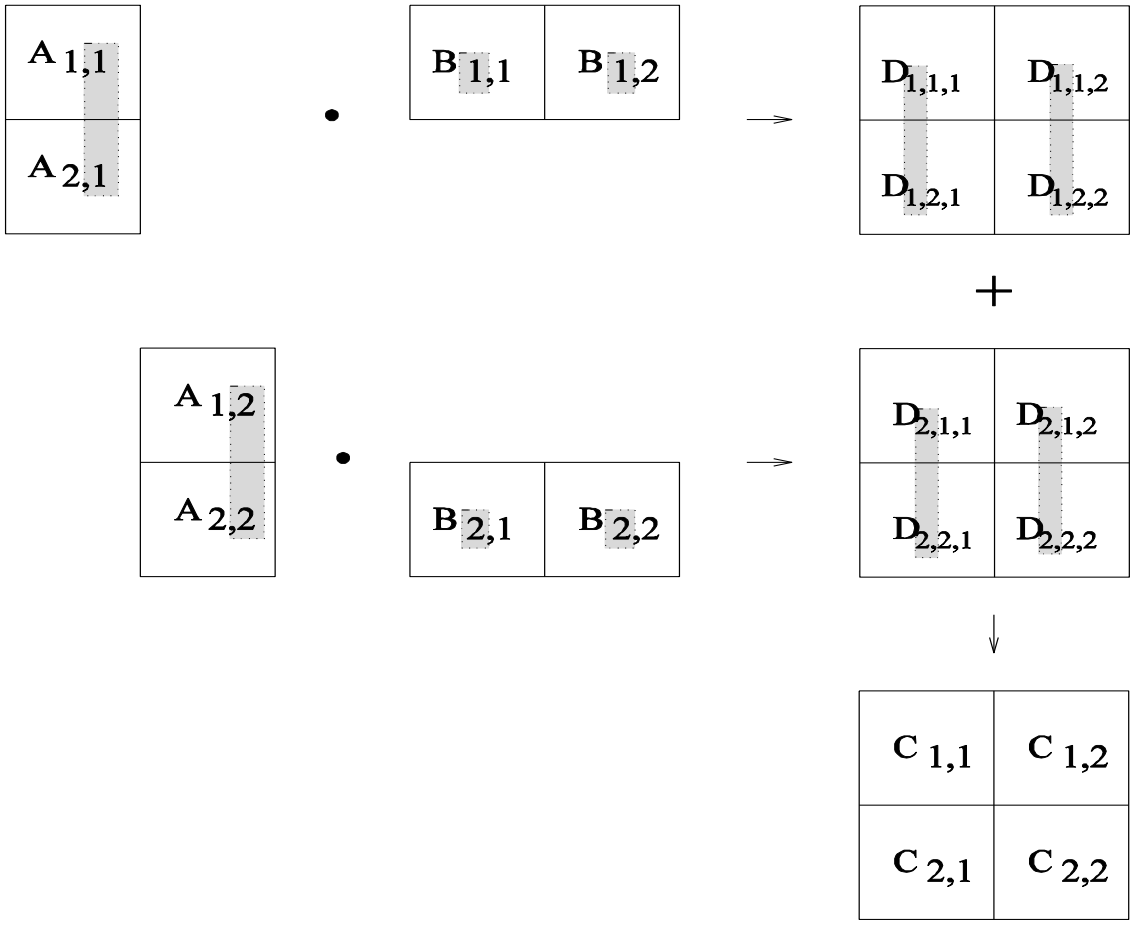
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Intermediate Data Partitioning: Example

- dense matrix multiplication
  - visualize this computation in terms of intermediate matrices ***D***.



Intermediate Data Partitioning: Example

- A decomposition of intermediate data: 8 + 4 tasks:

Stage I

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

Stage II

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

Task 01: $\boldsymbol{D}_{1,1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$

Task 02: $\boldsymbol{D}_{2,1,1} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$

Task 03: $\boldsymbol{D}_{1,1,2} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$

Task 04: $\boldsymbol{D}_{2,1,2} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$

Task 05: $\boldsymbol{D}_{1,2,1} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$

Task 06: $\boldsymbol{D}_{2,2,1} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$

Task 07: $\boldsymbol{D}_{1,2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$

Task 08: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

Task 09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

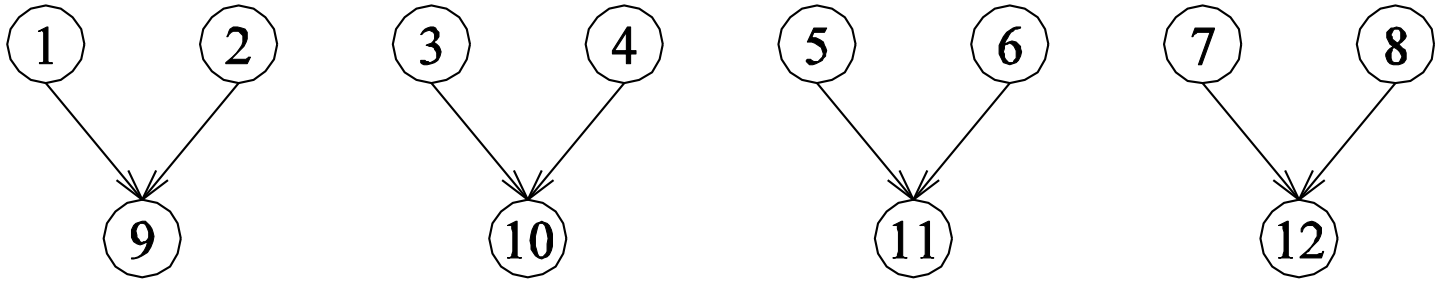
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

Task 11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

Task 12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

Intermediate Data Partitioning: Example

The task dependency graph for the decomposition (shown in previous foil) into 12 tasks is as follows:



The Owner Computes Rule

- Each datum is assigned to a process
- Each process computes values associated with its data
- Implications
  - input data decomposition
    - all computations using an input datum are performed by its process
  - output data decomposition
    - an output is computed by the process assigned to the output data

References

- Adapted from slides “Principles of Parallel Algorithm Design” by Ananth Grama
- Based on Chapter 3 of “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003