Preliminaries - Decomposition, Tasks, and Dependency Graphs

- The first step in developing a parallel algorithm is to *decompose* the problem into tasks that can be executed concurrently
- A given problem may be decomposed into tasks in many different ways.
- Tasks may be of same, different, or even indeterminate sizes.
- A decomposition can be illustrated in the form of a directed graph.
  - Such a graph is called a *task-dependency graph*.
  - Nodes correspond to tasks and edges indicate dependencies

Example: Multiplying a Dense Matrix with a Vector

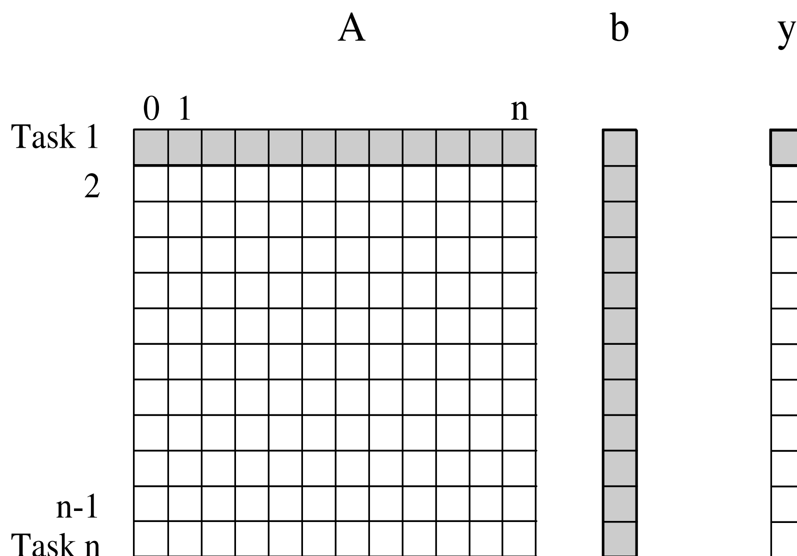

Fig 1 Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

- **Observations:**
  - Tasks share the vector ***b*** but they have no control dependencies.
    - There are zero edges in the task-dependency graph
  - All tasks are of the same size in terms of number of operations.
- Is this the maximum number of tasks we could decompose this problem into?

- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution.
- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program.
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

Critical Path, Critical Path Length

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.
- The longest such path between any pair of zero in-degree to zero out-degree node is known as the *critical path*.
- The length of the longest path in a task dependency graph is called the *critical path length*.

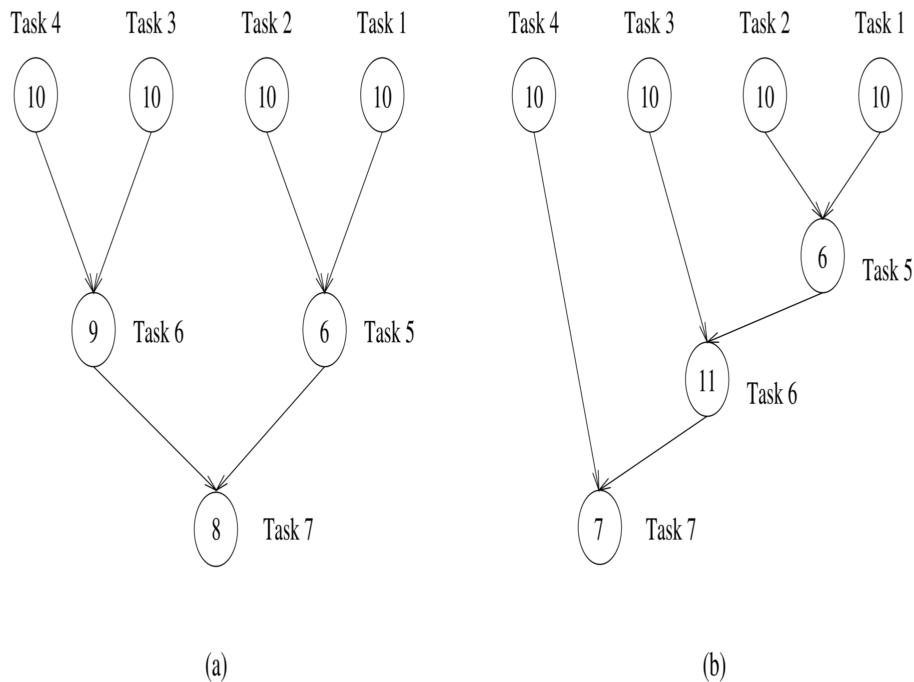

Fig 2

Limitations in Parallel Performance

- Parallel time cannot be made arbitrarily small by making the decomposition finer in granularity.
  - A parallel algorithm will inherently have a limited number of decomposable tasks
  - For example, in the case of multiplying a dense matrix with a vector, there can be no more than ($\mathbf{n^2}$) concurrent tasks.
  - Tasks interaction is another limiting factor on parallel performance
- *Task interaction graph*: an undirected graph that captures the pattern of interaction among tasks
- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.
- The edge-set of a task-interaction graph is a superset of the edge-set of a task-dependency graph.
- **Task Interaction Graphs: An Example**
  Consider the problem of multiplying a sparse matrix ***A*** with a vector ***b***. The following observations can be made:
- **Notes**
  - Decomposition is as before; each y[i] computation is a task.
  - Only non-zero elements of matrix ***A*** participate in the computation, in this case.
  - We also partition ***b*** across tasks; b[i] is held by Task i.

Processes and Mapping

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.
  - Thus, a parallel algorithm must also provide a mapping of tasks to processes.
  - **Note:** mapping is from tasks to processes, as opposed to processors.

- Because typical programming APIs do not allow easy binding of tasks to physical processors.
- We aggregate tasks into processes and rely on the system to map these processes to physical processors.
- Processes (no in UNIX sense): logical computing agents that perform tasks
- Task + task data + task code required to produce the task's output
- Processors: physical hardware units that perform tasks

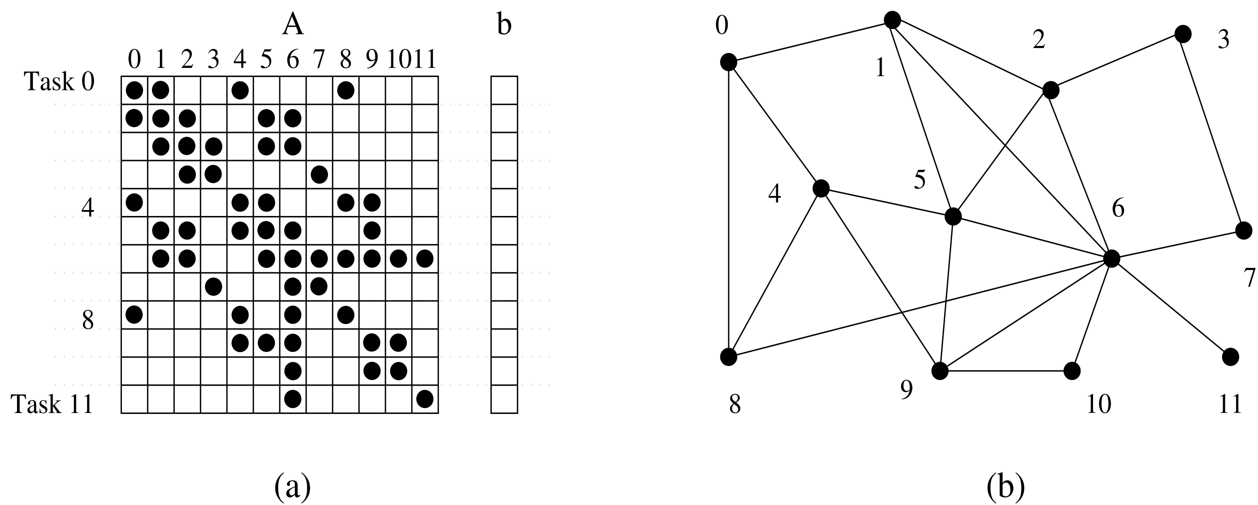

Fig 3

- An appropriate mapping must minimize parallel execution time by:
  1. Mapping independent tasks to different processes.
  2. Assigning tasks on critical path to processes as soon as they become available.
  3. Minimizing interaction between processes by mapping tasks with dense interactions to the same process.
- These criteria often conflict with each other.

1. E.g., a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

Processes and Mapping: Example

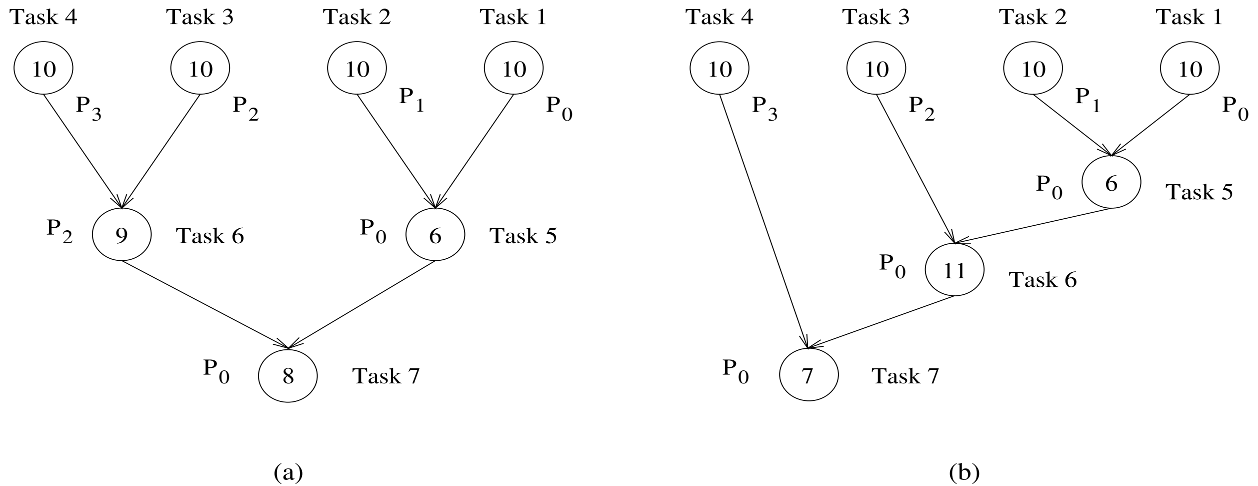

Fig 4 Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

Decomposition Techniques

- Decomposition:
  - The process of dividing the computation into smaller pieces of work i.e., *tasks*
  - Tasks are programmer defined and are considered to be indivisible
- So how does one decompose a task into various subtasks?
- There is no single recipe that works for all problems!
- Commonly used techniques that apply to broad classes of problems:
  - Recursive decomposition
  - Data decomposition
  - Exploratory decomposition
  - Speculative decomposition
  - Hybrid decomposition

Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.

Example: Quicksort

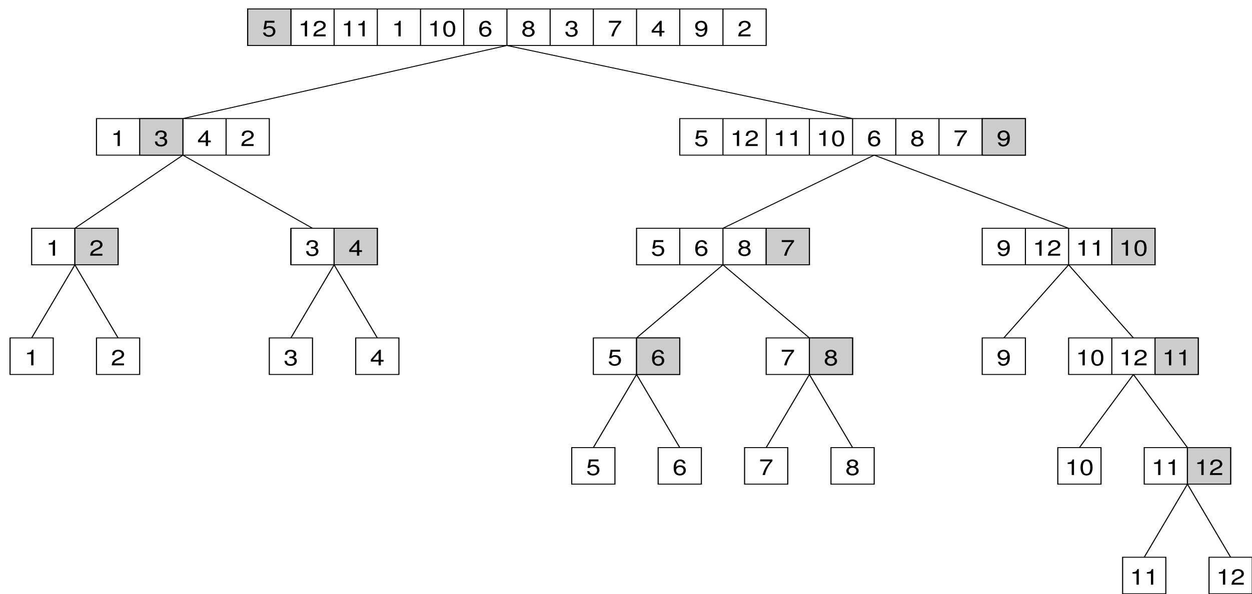

Fig 5 The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.

Example: Finding the Minimum

1. **procedure** SERIAL_MIN(A,n)

    2. **begin**

    3. *min* =A[0];

    4. **for** i:= 1 **to** n − 1 **do**

    5. **if** $(A[i] < min)$ $min := A[i]$;

    6. **endfor**;

    7. **return** *min*;

    8. **end** SERIAL_MIN

Example: Finding the Minimum

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.         min := A [0]  ;
5. else
6.         lmin := RECURSIVE_MIN ( A, n/2 );
7.         rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.         if (lmin  < rmin) then
9.                 min := lmin;
10.        else
11.                min := rmin;
12.        endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

The code in the previous foil can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of finding the minimum number in the set {4, 9, 1, 7, 8, 11, 2, 12}. The task dependency graph associated with this computation is as follows:

Load Balancing

- Dynamic mapping is sometimes also referred to as dynamic load balancing, since load balancing is the primary motivation for dynamic mapping.
- Dynamic mapping schemes can be *centralized* or *distributed*.

Centralized Dynamic Mapping

- Processes are designated as masters or slaves.
- When a process runs out of work, it requests the master for more work.
- When the number of processes increases, the master may become the bottleneck.
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling.
- Selecting large chunk sizes may lead to significant load imbalances as well.
- A number of schemes have been used to gradually decrease chunk size as the computation progresses.

Distributed Dynamic Mapping

- Each process can send or receive work from other processes.
- This alleviates the bottleneck in centralized schemes.
- There are four critical questions:
  - How are sending and receiving processes paired together,
  - Who initiates work transfer,
  - How much work is transferred, and
  - When is a transfer triggered?

Minimizing Interaction Overheads

- **Maximize data locality**: Where possible, reuse intermediate data. Restructure computation so that data can be reused in smaller time windows.
- **Minimize volume of data exchange**: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated.
- **Minimize frequency of interactions**: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible.
- **Minimize contention and hot-spots**: Use decentralized techniques, replicate data where necessary.
- **Overlapping computations with interactions**: Use non-blocking communications, multithreading, and prefetching to hide latencies.
- Replicating data or computations.
- Using group communications instead of point-to-point primitives.
- Overlap interactions with other interactions.

Parallel Algorithm Models

- An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.
- **Data Parallel Model**: Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.

  - Usually based on data decomposition followed by static mapping
  - Uniform partitioning of data followed by static mapping guarantees load balance
  - Example algorithm: dense matrix multiplication

- **Task Graph Model**: Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.
  - Typically used to solve problems where amount of data associated with a task is large relative to computation
  - Static mapping usually used to optimize data movement costs
  - Example algorithm: parallel quicksort, sparse matrix factorization
- **Master-Slave Model**: One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- **Pipeline / Producer-Consumer Model**: A stream of data is passed through a succession of processes, each of which perform some task on it.
- **Hybrid Models**: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.