CSE 891 - Section 1: Parallel Computing - Fundamentals and Applications

Fall 2014 - Lecture 9: Principles of Parallel Algorithm Design (Chapter 3 by Grama et al. [2])

H. Metin Aktulga
hma@cse.msu.edu
517-355-1646

Lecture 8- Summary

- Groups and Communicators
- An Example: Parallel Matrix-Vector Multiplication
  - A simple 1D decomposition approach
  - Alternative primitives for inter-node comm's
  - Communication volume analysis
  - 2D decomposition for better scalability
  - Importance of algorithm design + choice of MPI primitives
- Creating and using Cartesian topologies
- How to time your MPI applications?

Overview

Terminology

- Tasks and decomposition
- Processes and mapping
- Processes vs. processors

Decomposition techniques

- Spatial, Recursive, Exploratory, Hybrid decompositions

Characteristics of tasks and interactions

- Task generation, granularity and context
- Characteristics of task interactions

Overview

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

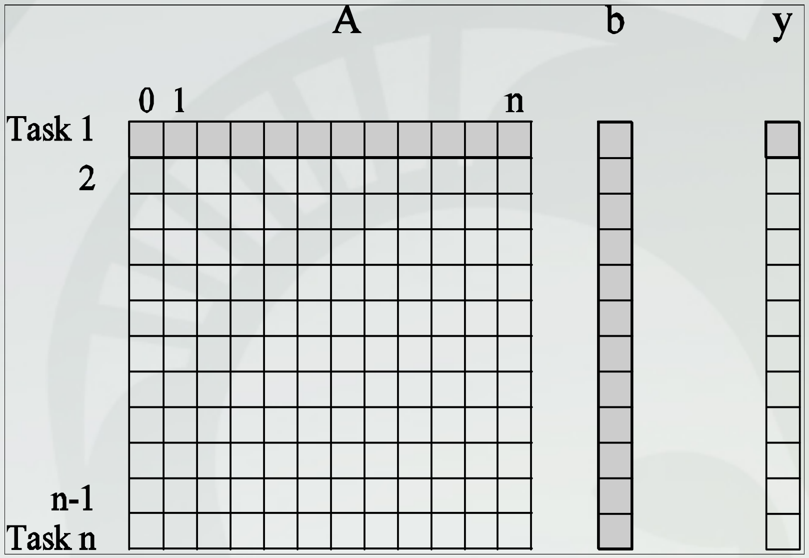
Parallel Algorithm Design Models

- Data-Parallel, Work-Pool, Task Graph, Master-Slave, Pipeline, and Hybrid Models

Tasks and decomposition

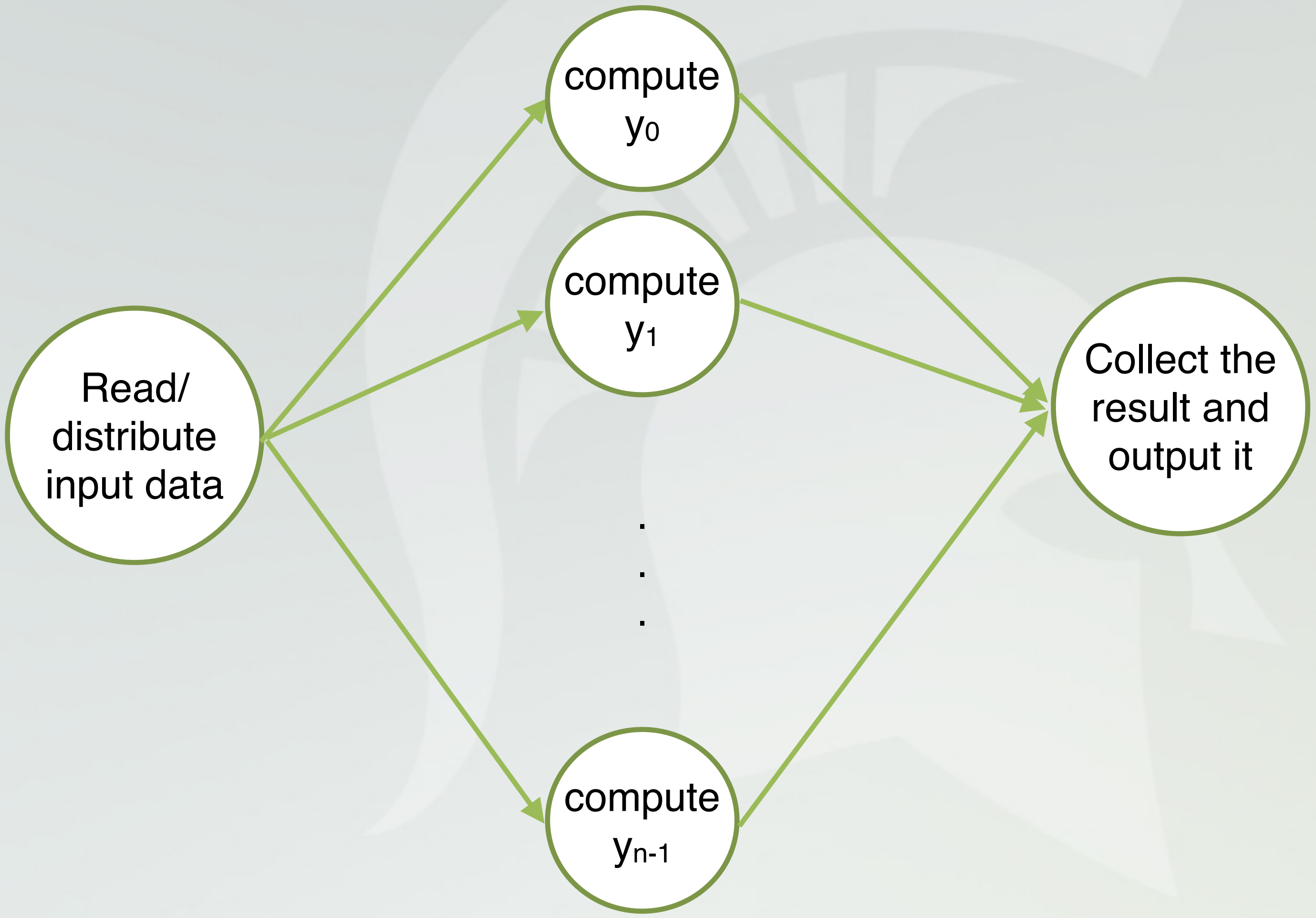
- A **task** is a set of instructions that can be executed by a single process.
- Tasks may be of same, different or even *unknown* sizes.
- The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently.
- Several different decompositions may exist for a given problem.
- A **task dependency graph** is a directed graph with nodes corresponding to tasks and edges corresponding to dependencies between tasks.

Example: Matrix-vector multiplication



- Computation of each element of output vector ***y*** is independent of others. So a dense matrix-vector product can be decomposed into ***n*** tasks. Highlighted portion of the matrix and vector are accessed by *Task 1*.
- What would the task dependency graph look like for this example?

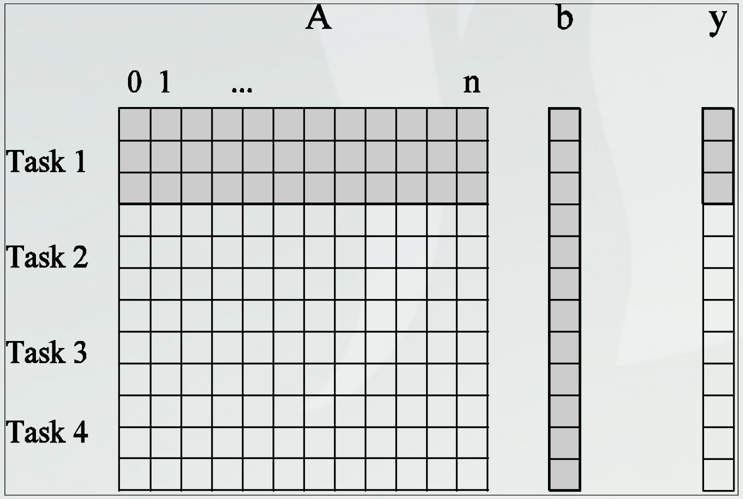
Example: Matrix-vector multiplication



n tasks can be executed in parallel. Is this the *only* number of tasks that we can decompose this problem into?

Granularity of task decompositions

- The number of tasks into which a problem is decomposed determines its **granularity**.
    - a large number of tasks —> fine-grained decomposition
    - a small number of tasks —> coarse grained decomposition



A coarse grained decomposition of the dense matrix-vector product example.

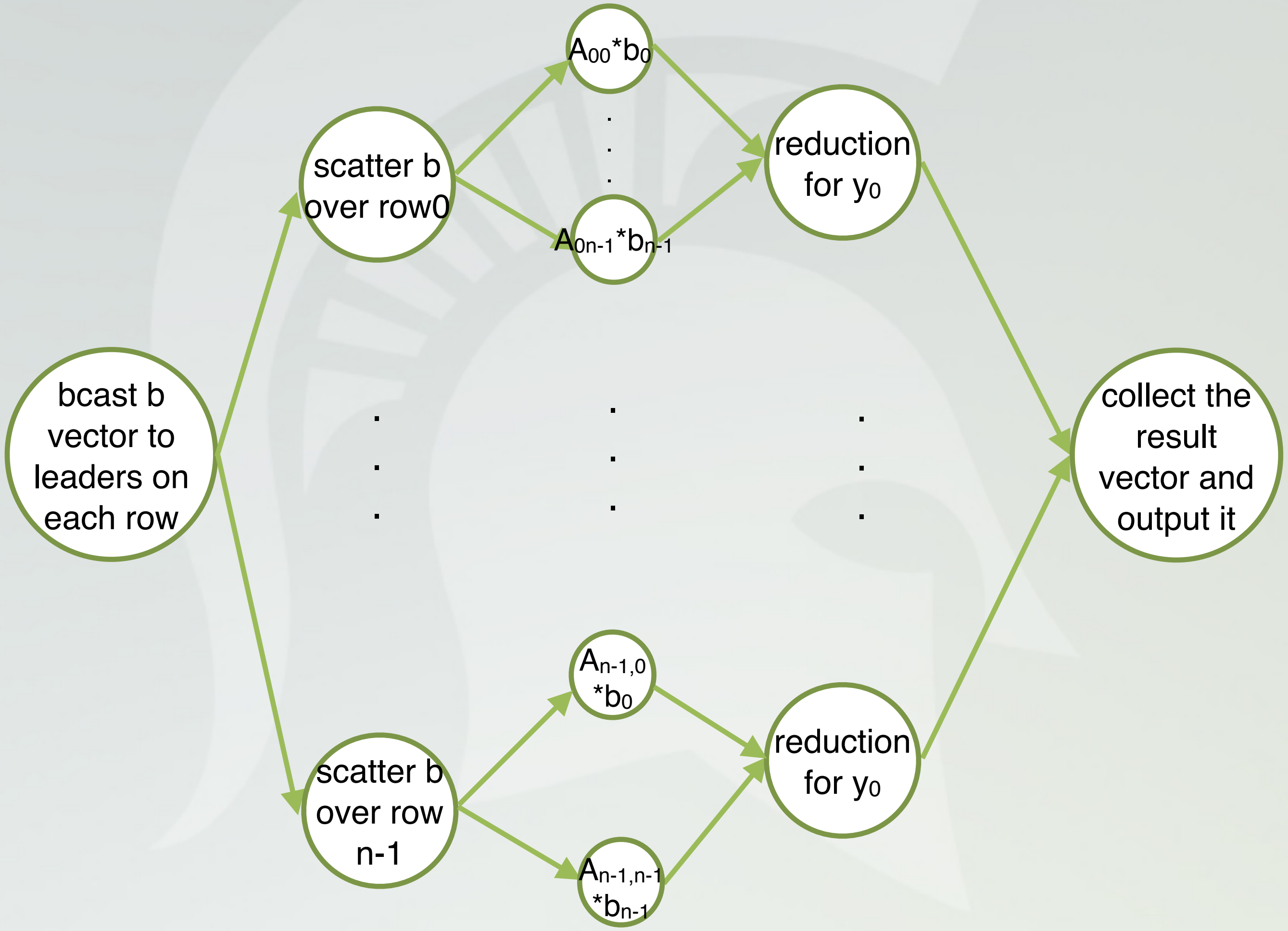
- Is *n* the maximum number of tasks that this problem can be decomposed into?

No! A range of finer-grained decomposition possibilities exist in the range [n, n^2].

Degree of Concurrency

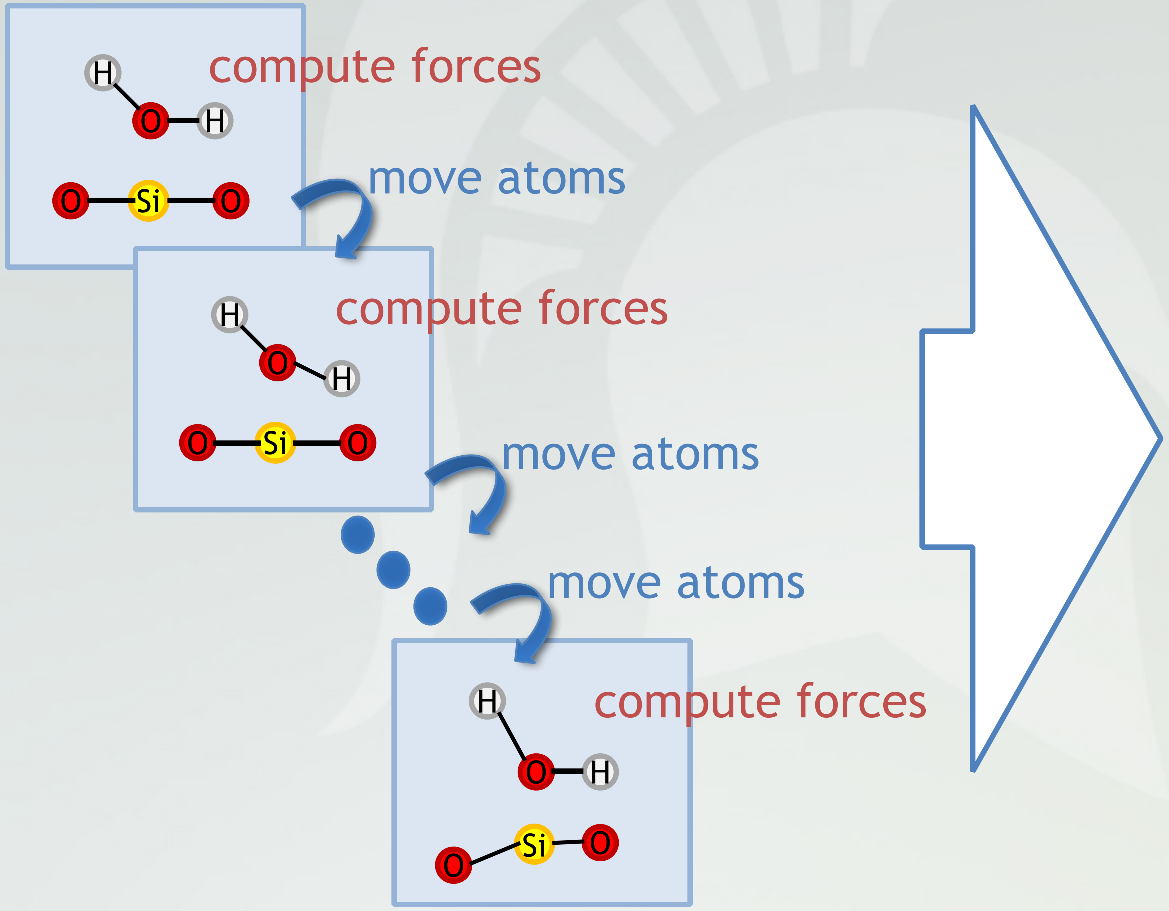
- The number of tasks that can be executed in parallel is the **degree of concurrency** of a decomposition.
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.
- The degree of concurrency may change over the course of a computation.
  - **maximum degree of concurrency** is the maximum number of concurrent tasks at any point during execution.
  - **average degree of concurrency** is the average number of concurrent tasks. Need to weigh different phases by their computational intensities.

Example: Matrix-vector multiplication



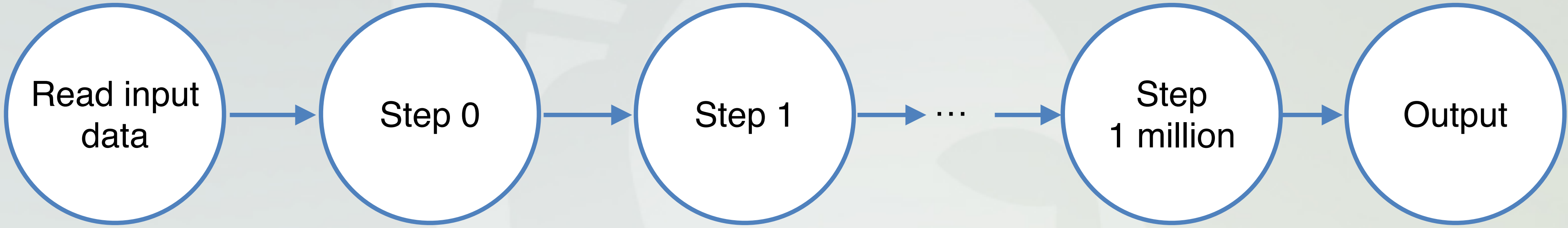
Degrees of concurrency	1	n	n^2	n	1

Example: Molecular Dynamics



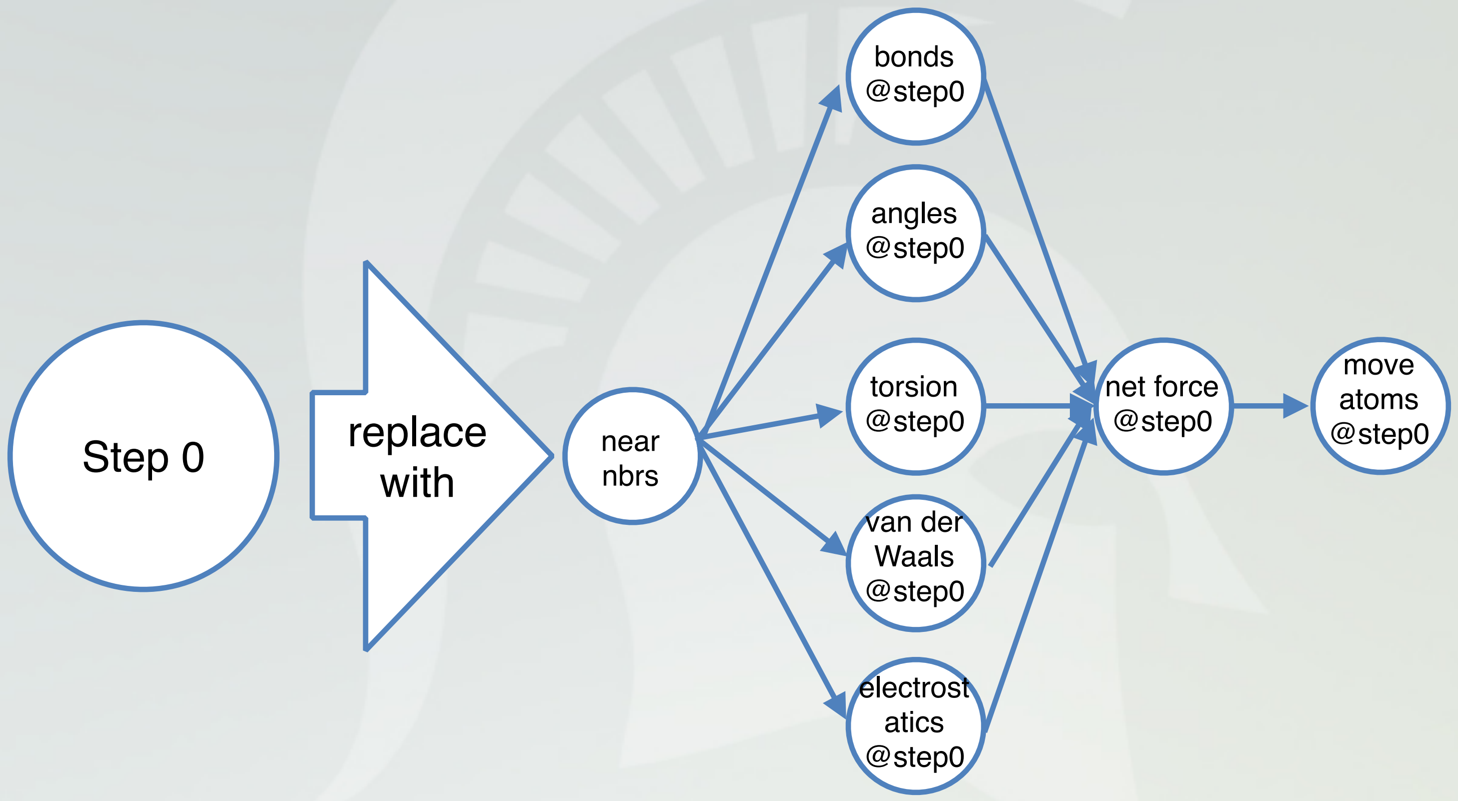
- Analyze trajectory
- Calculate properties
- Applications in:
  - Protein folding
  - Drug design
  - Nano-materials

Example: MD task dependency graph



- What is the degree of concurrency here?
- Is this a good parallel task decomposition? Why?
- How can we improve this?

Example: MD task dependency graph



- Note that there are various kinds of interactions to be computed at each step!
- Limited by the # of interaction types — does not scale well…

Example: MD task decomposition

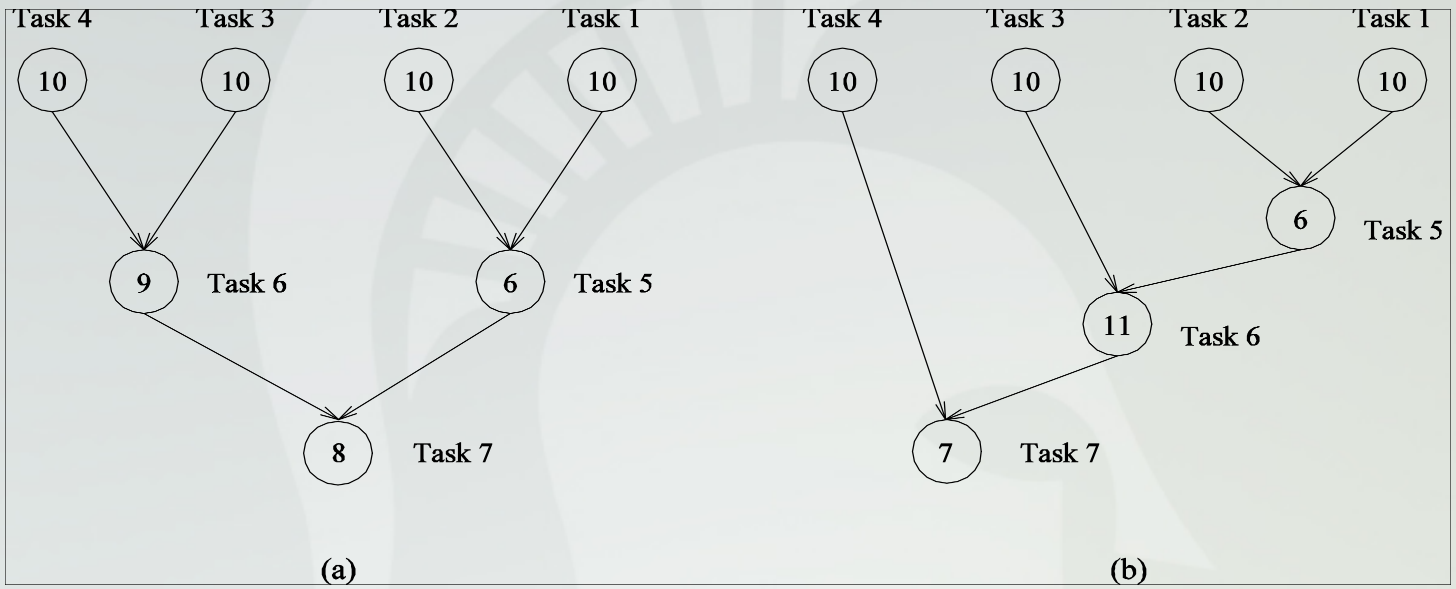
- Note that the interactions of each atom (particle) can be computed independently
- So the previous decomposition may be made even finer by identifying the interactions of each atom as a separate task
  - max concurrency = #interaction_types * #atoms

- An even finer task decomposition can be achieved by defining each individual interaction as a task!
  - max concurrency ~ c * $(\#atoms)^2$
  - or with range-limited intrs ~ C * (#atoms)

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 determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the **critical path length**.

Critical Path Length



- Numbers indicate the (relative) computational costs of each task
- What is the maximum degree of concurrency?
- What is the critical path length?
- What is the minimum possible parallel execution time? and how many processors are needed to achieve this?

Limits on parallel performance

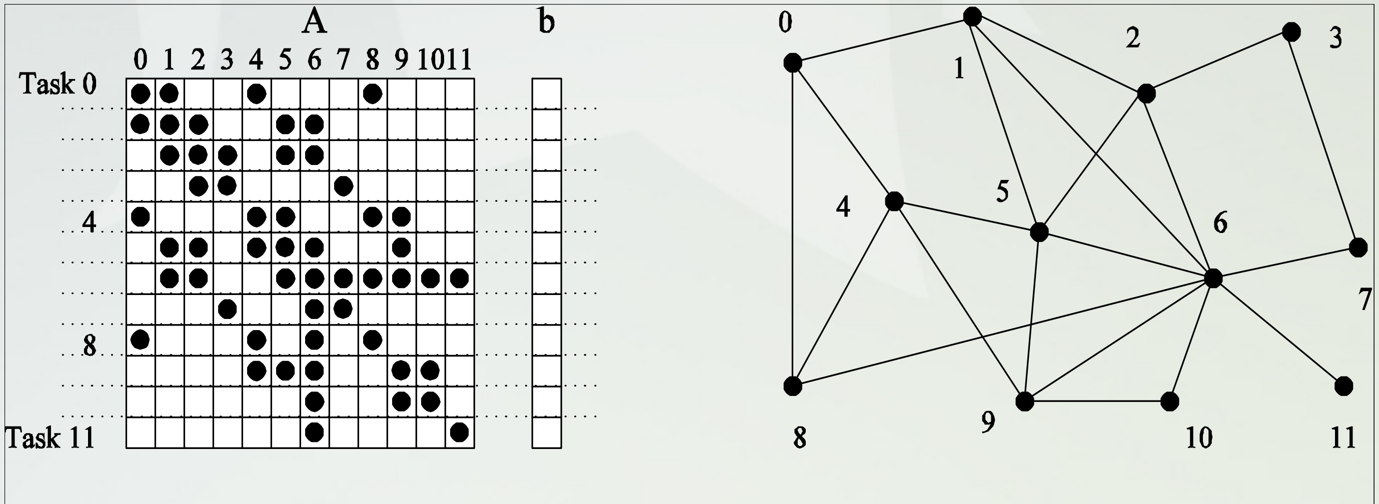
- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.
- There is an inherent bound on how fine the granularity of a computation can be.
  - In dense matrix-vector multiplication, there can be no more than $\mathbf{n}^2$ concurrent tasks.
  - In MD, total number of interactions is the inherent bound.
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The trade-off between the granularity of a decomposition and associated overheads often determines performance bounds.

Task interaction graphs

- Tasks in a computation generally need to exchange data with others in a decomposition.
  - In 1D decomposition of the dense matrix-vector multiplication, if the vector is not already replicated across all tasks, they will have to communicate elements of the vector.
- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a **task interaction graph**.
- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.
- The finer the decomposition, the more complex the task interaction graph.

Task Interaction Graphs: An Example

- Now consider the multiplication of a sparse matrix ***A*** with a vector ***b***
- As before, the computation of each element of the result vector can be viewed as an independent task (1D decomposition).
- Unlike a dense matrix-vector product though, only non-zero elements of matrix ***A*** participate in the computation.
- If, for memory optimality, we also partition ***b*** across tasks, then one can see that the task interaction graph of the computation is identical to the graph of the matrix ***A*** (the graph for which ***A*** represents the adjacency structure).



Task Interaction Graphs, Granularity, and Communication

- Exposing maximum parallelism requires fine task decompositions
- However, as a decomposition gets finer, associated overheads (data movement overhead/useful work) generally increase.
- So related trade-offs must be carefully considered in the design of a parallel algorithm for best performance and scalability!
- For example, consider the overheads incurred by defining 12 tasks vs. 4 tasks in sparse matrix-vector multiply (SpMV)

Processes and tasks

- In general, #tasks in a decomposition exceeds #processing units available.
- For this reason, a parallel algorithm must also provide a grouping of tasks (and associated data) to processes.
- For example, how should we group the tasks in SpMV?
- There exists a whole body of literature on how to partition graphs such that good load balance and minimum data movement is achieved
- NP-hard problem, heuristics and various packages exist for this purpose (METIS, Scotch, PaToH, etc.)

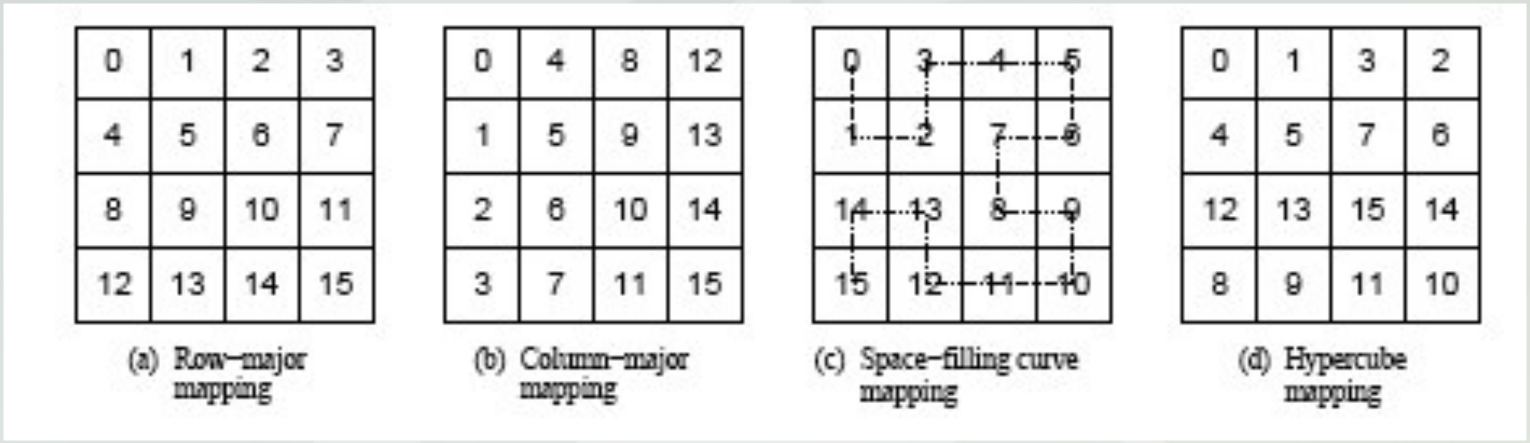
Topologies and Mappings

- Every application has its own communication pattern: communication graph *G* (binomial tree for parallel sum)
- Each machine has its own network topology (recall ring, hypercube, k-d torus, fat trees): interconnection graph *H*
- *Γ*: *G* -> *H* defines a mapping of processes to physical processors (cores).
- #messages & volume is important, but equally important is the distance and routes traveled by each message
- The goodness of mapping *Γ* can be measured by metrics such as dilation (average # of hops), average traffic and congestion.
- Graph isomorphism is NP-complete - heuristics are used.

Mappings in MPI

- MPI allows programmers to define logical topologies
- A commonly used topology is *k*-d mesh/torus - domain decomposition as in molecular dynamics, stencils
- When mapping, processor ids in `MPI_COMM_WORLD` can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways.
- The goodness of any such mapping is determined by the interaction pattern of the underlying program, the topology of the machine and the MPI implementation!
- Programmer does not have any control over MPI mappings. Options: custom mappings or 3rd-party tools

Topologies and Mappings



Different ways to map a set of processes to a 2D grid. There is no single best map, depends on the application and architecture.
(a) row-major mapping
(b) column-major mapping
(c) a space-filling curve (dotted line), and
(d) hypercube mapping - neighboring processes are directly connected in a hypercube.