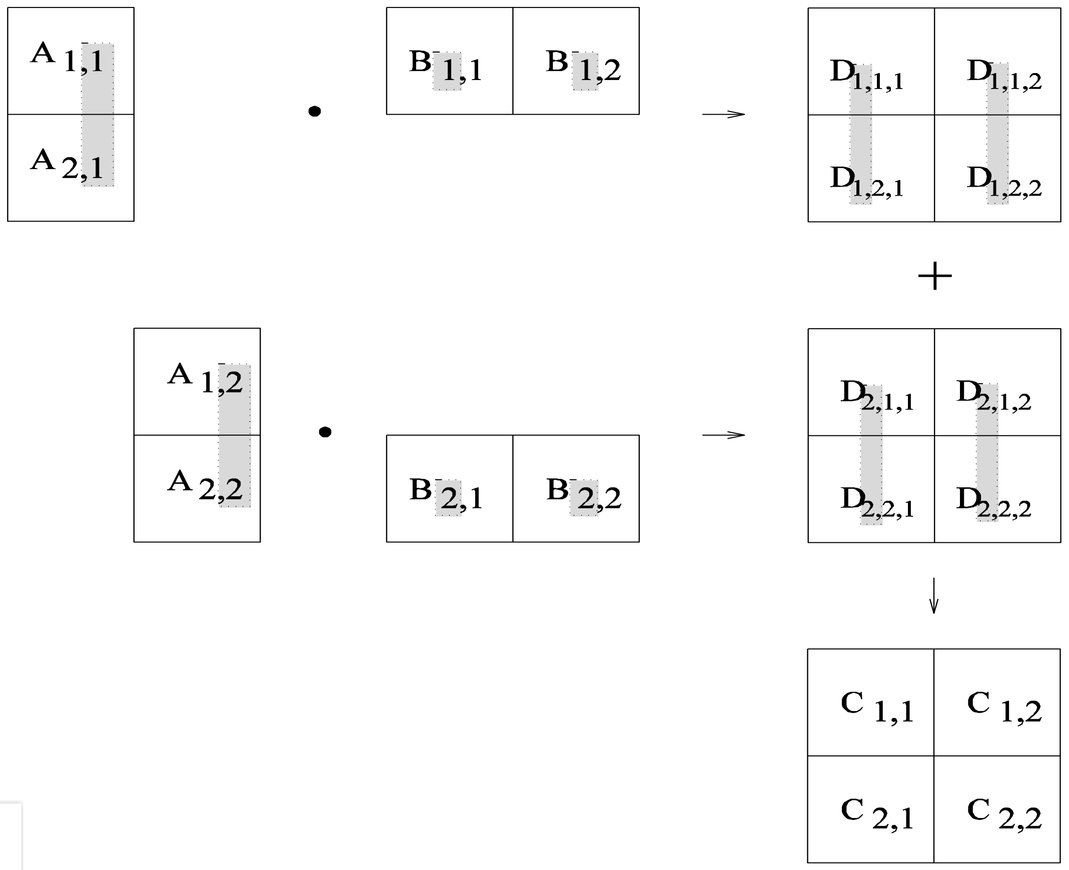
INTERMEDIATE DATA PARTITIONING

Computation can often be viewed as a sequence of transformation from the input to the output data.

In these cases, it is often beneficial to use one of the intermediate stages as a basis for decomposition.

INTERMEDIATE DATA PARTITIONING: EXAMPLE

Let us revisit the example of dense matrix multiplication. We first show how we can visualize this computation in terms of intermediate matrices ***D***.



INTERMEDIATE DATA PARTITIONING: EXAMPLE

A decomposition of intermediate data structure   leads to the following decomposition into 8 + 4 tasks:

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \left(\left\{ \begin{matrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \\ D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{matrix} \right\} \right)$$

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

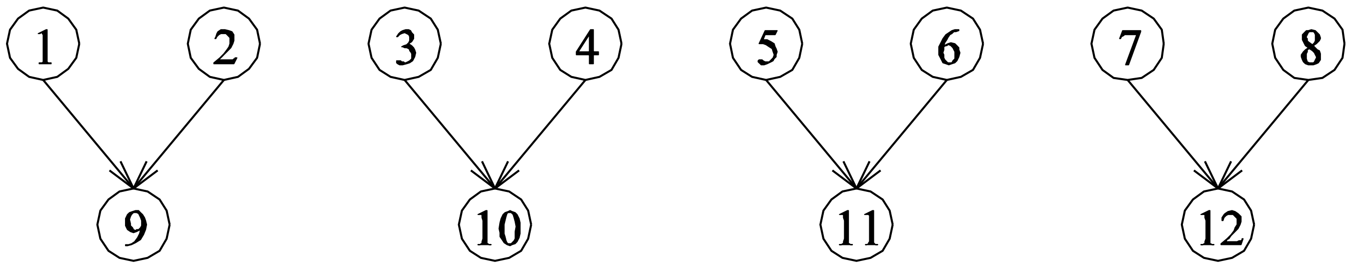
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

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

The *Owner Computes Rule* generally states that the process assined a particular data item is responsible for all computation associated with it.

In the case of input data decomposition, the owner computes rule imples that all computations that use the input data are performed by the process.

In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.

EXPLORATORY DECOMPOSITION

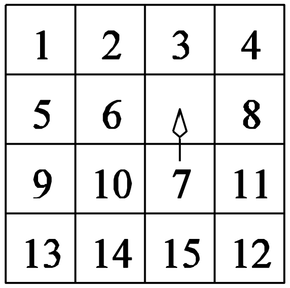
In many cases, the decomposition of the problem goes hand-in-hand with its execution.

These problems typically involve the exploration (search) of a state space of solutions.

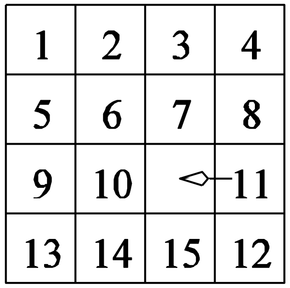
Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

EXPLORATORY DECOMPOSITION: EXAMPLE

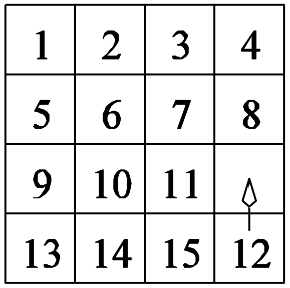
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).



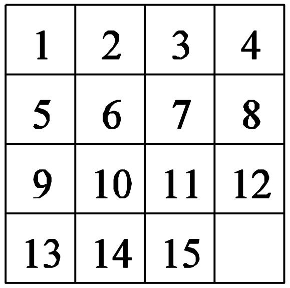
(a)



(b)



(c)

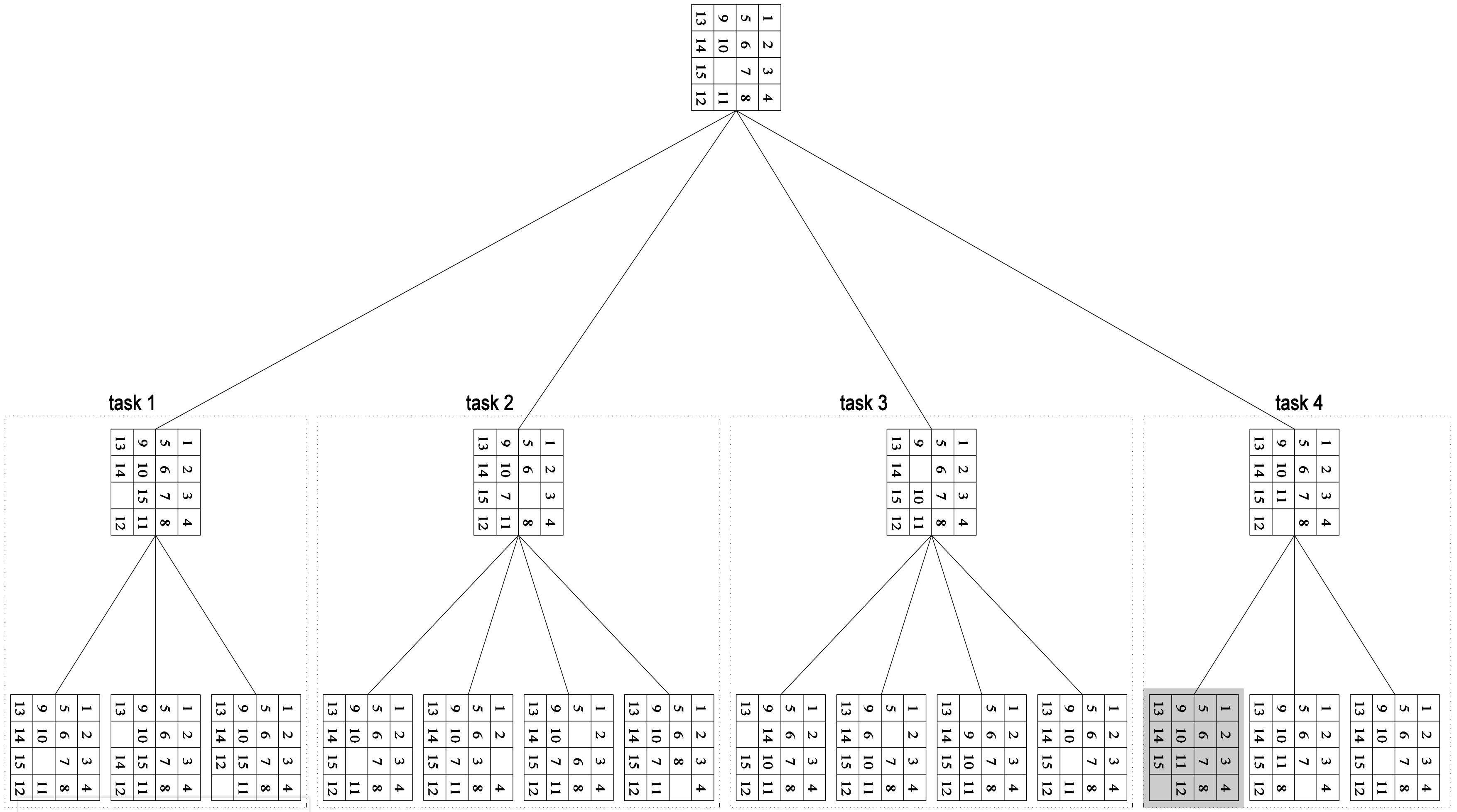


(d)

Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.

EXPLORATORY DECOMPOSITION: EXAMPLE

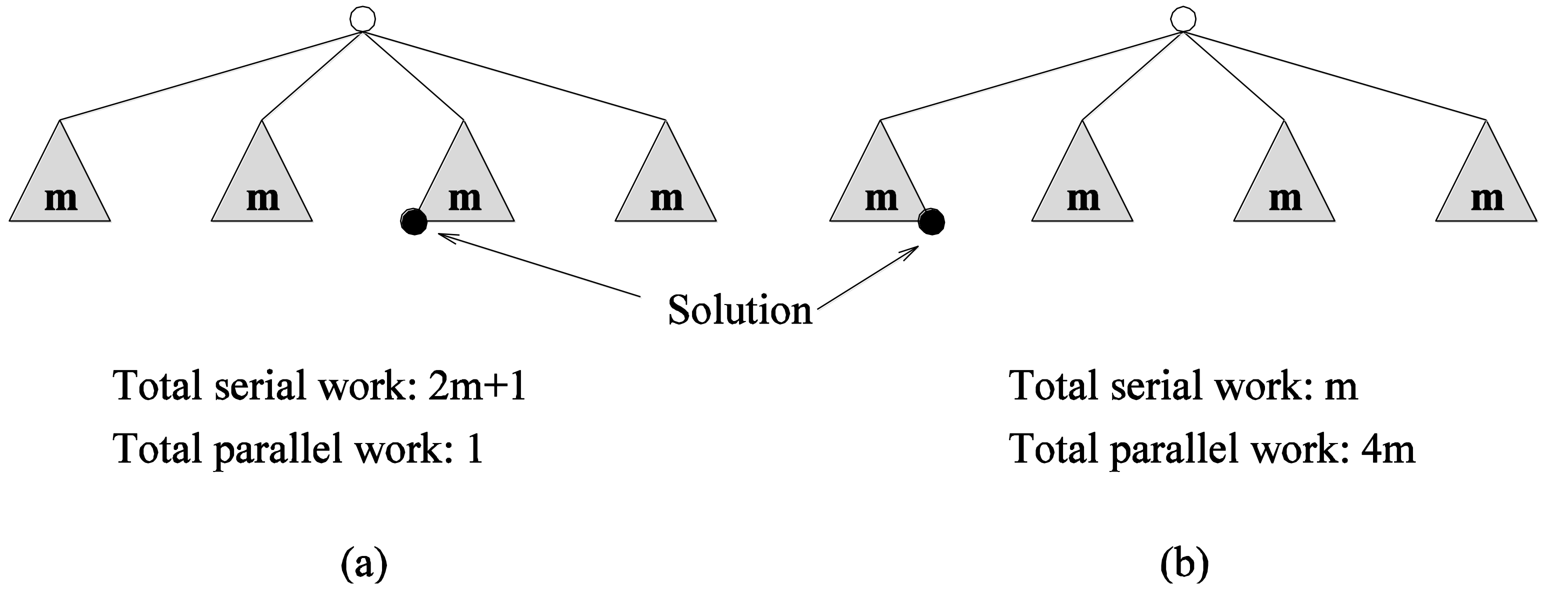
The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



EXPLORATORY DECOMPOSITION: ANOMALOUS COMPUTATIONS

In many instances of exploratory decomposition, the decomposition technique may change the amount of work done by the parallel formulation.

This change results in super- or sub-linear speedups.



SPECULATIVE DECOMPOSITION

In some applications, dependencies between tasks are not known a-priori.

For such applications, it is impossible to identify independent tasks.

There are generally two approaches to dealing with such applications: conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies, and, optimistic approaches, which schedule tasks even when they may potentially be erroneous.

Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

SPECULATIVE DECOMPOSITION: EXAMPLE

A classic example of speculative decomposition is in discrete event simulation.

The central data structure in a discrete event simulation is a time-ordered event list.

Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.

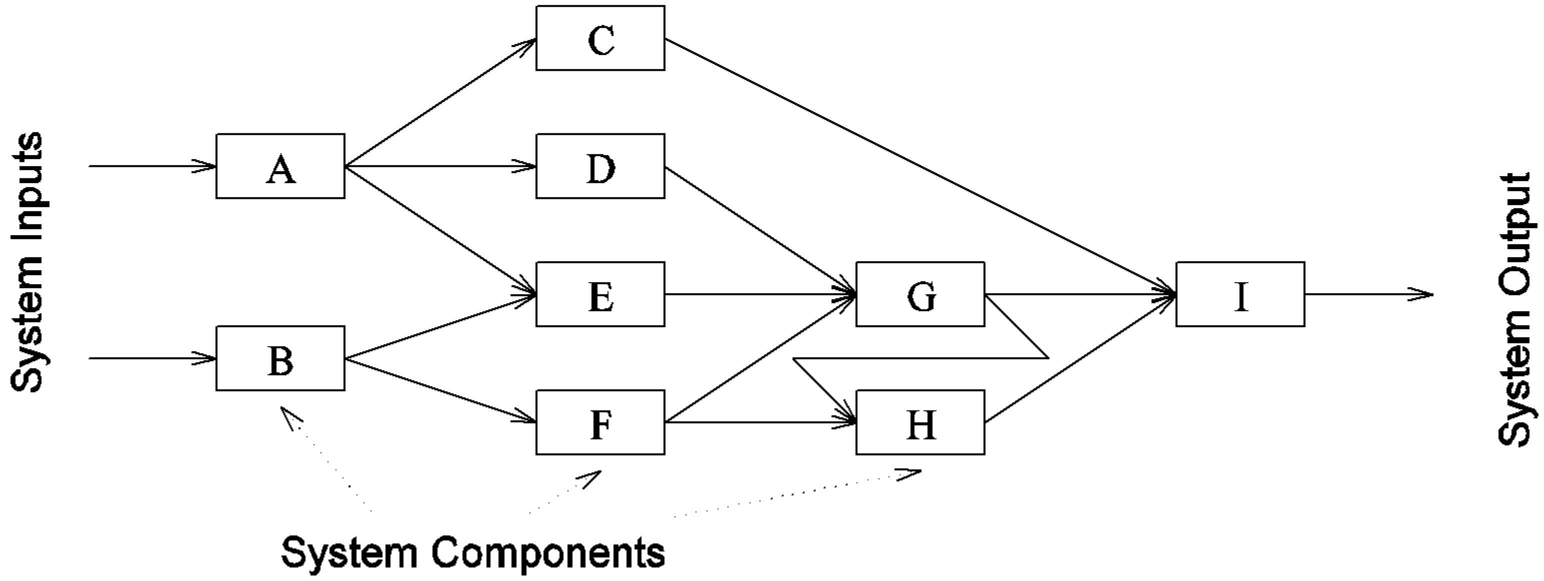
Consider your day today as a discrete event system - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.

Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.

Therefore, an optimistic scheduling of other events will have to be rolled back.

SPECULATIVE DECOMPOSITION: EXAMPLE

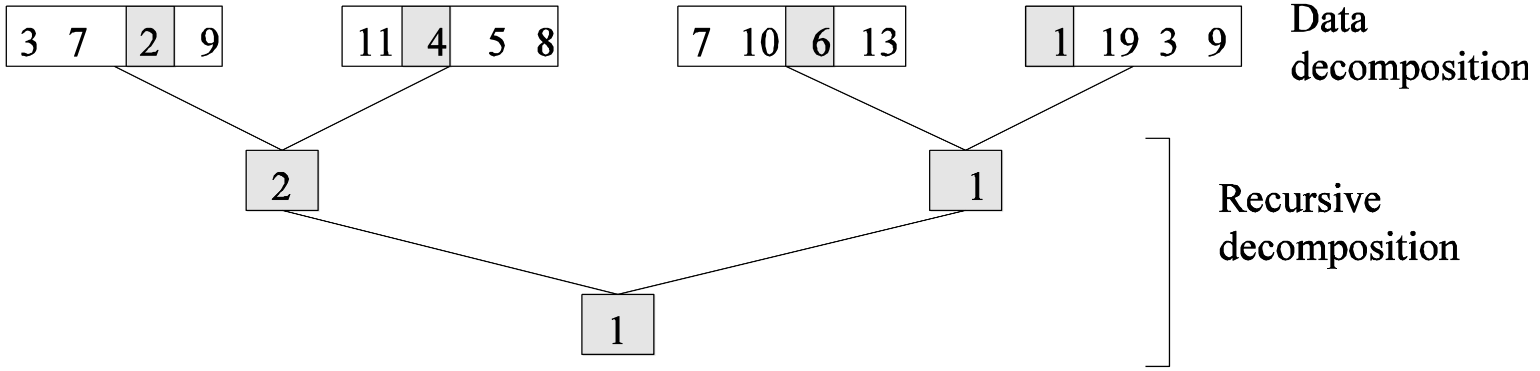
Another example is the simulation of a network of nodes (for instance, an assembly line or a computer network through which packets pass). The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue sizes, etc.).



HYBRID DECOMPOSITIONS

Often, a mix of decomposition techniques is necessary for decomposing a problem. Consider the following examples:

- In quicksort, recursive decomposition alone limits concurrency (Why?). A mix of data and recursive decompositions is more desirable.
- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.



CHARACTERISTICS OF TASKS

Once a problem has been decomposed into independent tasks, the characteristics of these tasks critically impact choice and performance of parallel algorithms. Relevant task characteristics include:

Task generation.

Task sizes.

Size of data associated with tasks.