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.

TASK GENERATION

Static task generation: Concurrent tasks can be identified a-priori. Typical matrix operations, graph algorithms, image processing applications, and other regularly structured problems fall in this class. These can typically be decomposed using data or recursive decomposition techniques.

Dynamic task generation: Tasks are generated as we perform computation. A classic example of this is in game playing - each 15 puzzle board is generated from the previous one. These applications are typically decomposed using exploratory or speculative decompositions.

TASK SIZES

Task sizes may be uniform (i.e., all tasks are the same size) or non-uniform.

Non-uniform task sizes may be such that they can be determined (or estimated) a-priori or not.

Examples in this class include discrete optimization problems, in which it is difficult to estimate the effective size of a state space.

SIZE OF DATA ASSOCIATED WITH TASKS

The size of data associated with a task may be small or large when viewed in the context of the size of the task.

A small context of a task implies that an algorithm can easily communicate this task to other processes dynamically (e.g., the 15 puzzle).

A large context ties the task to a process, or alternately, an algorithm may attempt to reconstruct the context at another processes as opposed to communicating the context of the task (e.g., 0/1 integer programming).

CHARACTERISTICS OF TASK INTERACTIONS

Tasks may communicate with each other in various ways. The associated dichotomy is:

Static interactions: The tasks and their interactions are known a-priori. These are relatively simpler to code into programs.

Dynamic interactions: The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especitally, as we shall see, using message passing APIs.

CHARACTERISTICS OF TASK INTERACTIONS

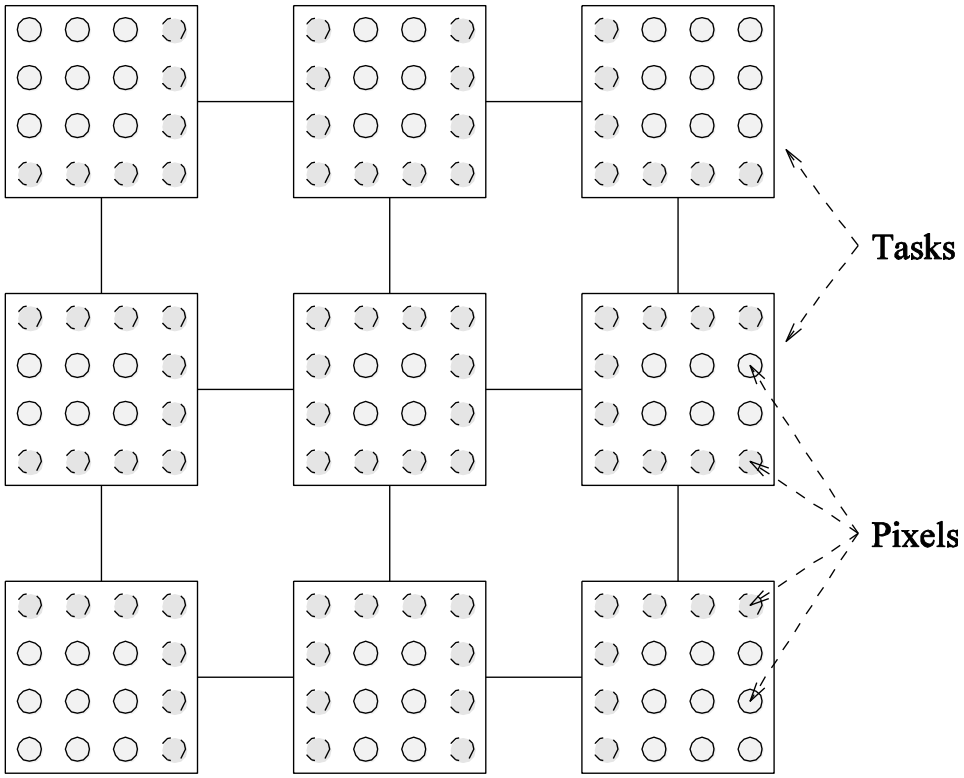
Regular interactions: There is a definite pattern (in the graph sense) to the interactions. These patterns can be exploited for efficient implementation.

Irregular interactions: Interactions lack well-defined topologies.

PURDUE
NExT

CHARACTERISTICS OF TASK INTERACTIONS: EXAMPLE

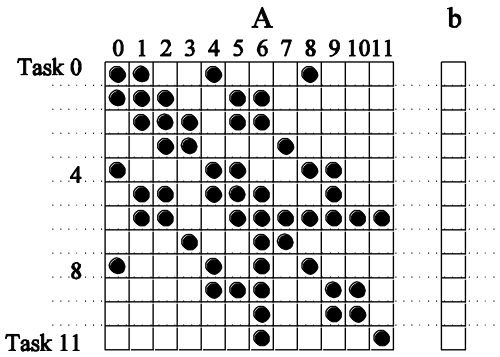
A simple example of a regular static interaction pattern is in image dithering. The underlying communication pattern is a structured (2-D mesh) one as shown here:



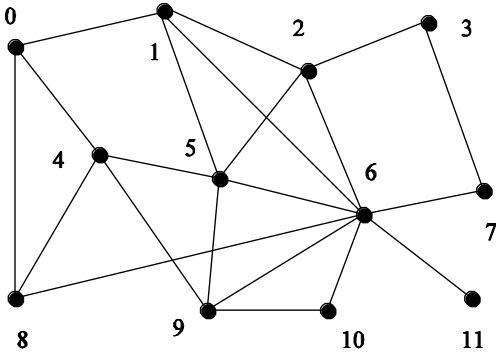
PURDUE NExT

CHARACTERISTICS OF TASK INTERACTIONS: EXAMPLE

The multiplication of a sparse matrix with a vector is a good example of a static irregular interaction pattern. Here is an example of a sparse matrix and its associated interaction pattern.



(a)



(b)

PURDUE NExT

CHARACTERISTICS OF TASK INTERACTIONS

Interactions may be read-only or read-write.

In read-only interactions, tasks just read data items associated with other tasks.

In read-write interactions tasks read, as well as modily data items associated with other tasks.

In general, read-write interactions are harder to code, since they require additional synchronization primitives.

PURDUE
NExT

CHARACTERISTICS OF TASK INTERACTIONS

Interactions may be one-way or two-way.

A one-way interaction can be initiated and accomplished by one of the two interacting tasks.

A two-way interaction requires participation from both tasks involved in an interaction.

One way interactions are somewhat harder to code in message passing APIs.

MAPPING TECHNIQUES

Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).

Mappings must minimize overheads.

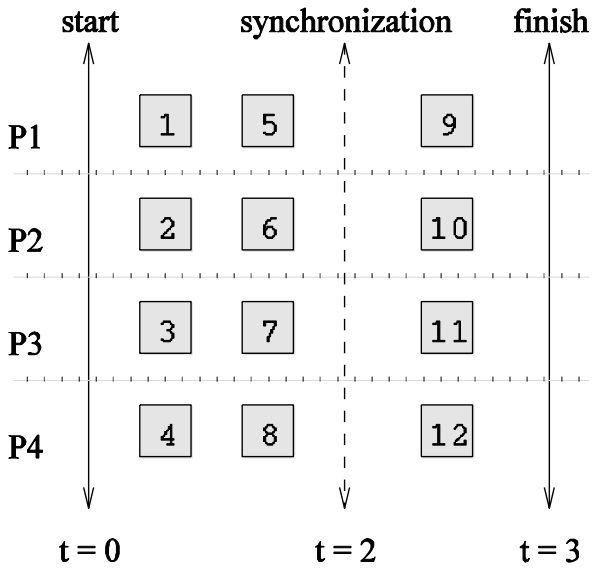
Primary overheads are communication and idling.

Minimizing these overheads often represents contradicting objectives.

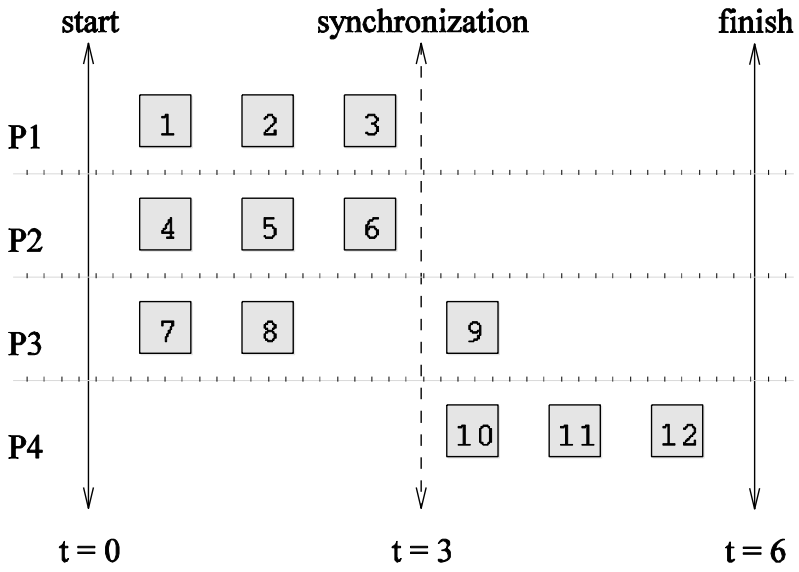
Assigning all work to one processor trivially minimizes communication at the expense of significant idling.

MAPPING TECHNIQUES FOR MINIMUM IDLING

Mapping must simultaneously minimize idling and load balance. Merely balancing load does not minimize idling.



(a)



(b)

PURDUE NExT

MAPPING TECHNIQUES FOR MINIMUM IDLING

Mapping techniques can be static or dynamic.

Static Mapping: Tasks are mapped to processes a-priori. For this to work, we must have a good estimate of the size of each task. Even in these cases, the problem may be NP complete.

Dynamic Mapping: Tasks are mapped to processes at runtime. This may be because the tasks are generated at runtime, or that their sizes are not known.

Other factors that determine the choice of techniques include the size of data associated with a task and the nature of underlying domain.

SCHEMES FOR STATIC MAPPING

Mappings based on data partitioning.

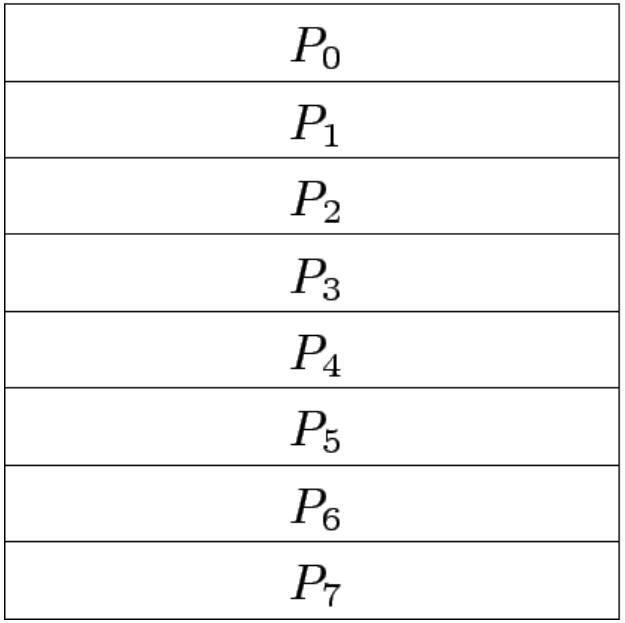
Mappings based on task graph partitioning.

Hybrid mappings.

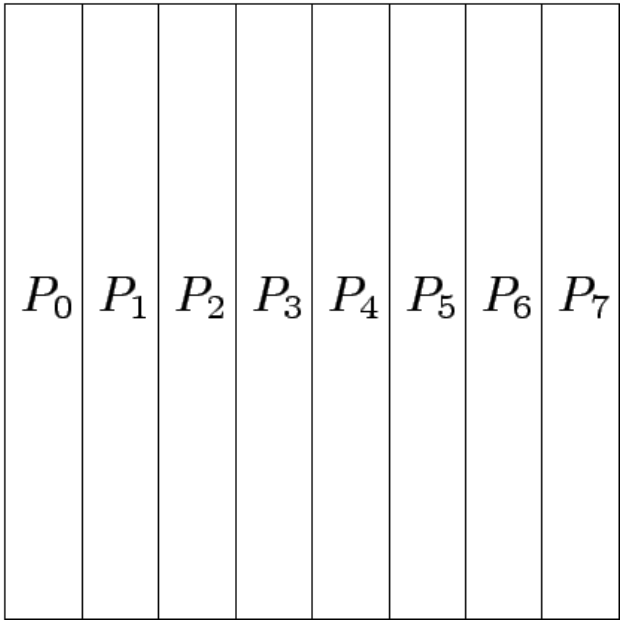
MAPPINGS BASED ON DATA PARTITIONING

We can combine data partitioning with the ``owner-computes'' rule to partition the computation into subtasks. The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution



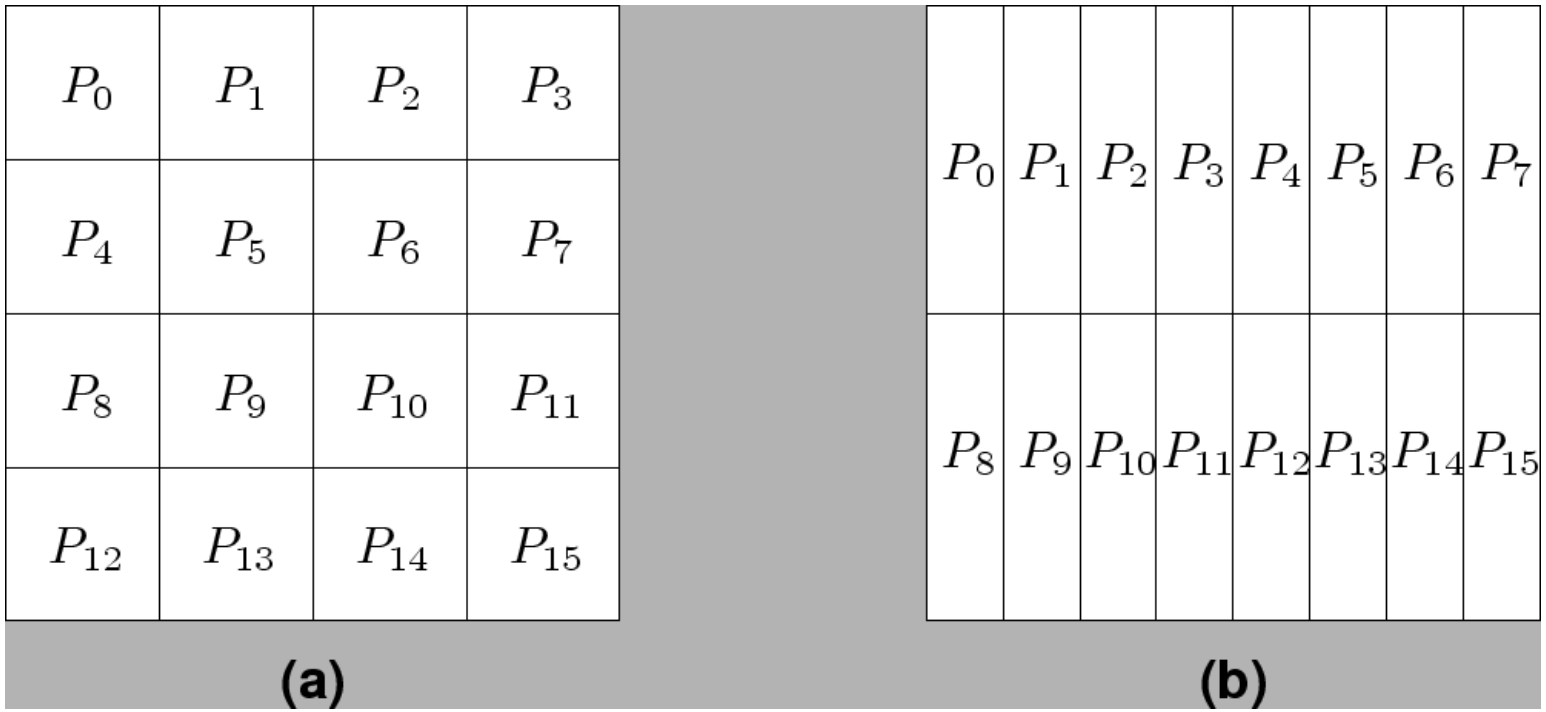
column-wise distribution



PURDUE NExT

BLOCK ARRAY DISTRIBUTION SCHEMES

Block distribution schemes can be generalized to higher dimensions as well.



PURDUE NExT

BLOCK ARRAY DISTRIBUTION SCHEMES: EXAMPLES

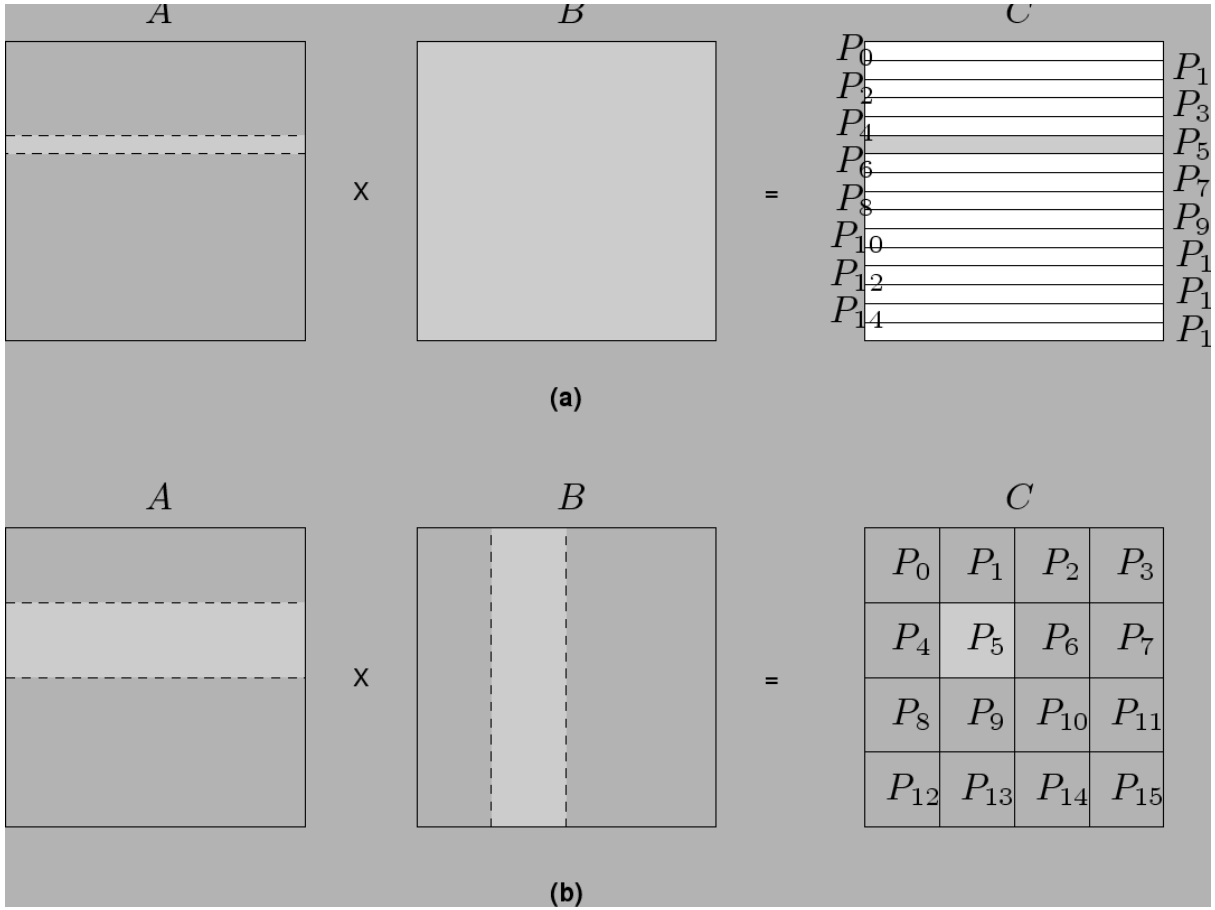
For multiplying two dense matrices ***A*** and ***B***, we can partition the output matrix ***C*** using a block decomposition.

For load balance, we give each task the same number of elements of ***C***. (Note that each element of ***C*** corresponds to a single dot product.)

The choice of precise decomposition (1-D or 2-D) is determined by the associated communication overhead.

In general, higher dimension decomposition allows the use of larger number of processes.

DATA SHARING IN DENSE MATRIX MULTIPLICATION



PURDUE NExT