
X. Mapping Techniques

27th April, 2009

Content

- mapping classification
 - schemes for static mapping
 - schemes for dynamic mapping
 - maximizing data locality
 - overlapping computations with interactions,
 - replication,
 - optimized collective interactions
-

Tasks vs. Processes vs. Processors

- Processors
 - are the hardware units that physically perform computations.
 - in most cases, there is a one-to-one correspondence between processes and processors
 - The tasks, into which a problem is decomposed, run on physical processors.
 - A process
 - refers to a processing or computing agent that performs tasks.
 - is an abstract entity that uses the code and data corresponding to a task to produce the output of that task within a finite amount of time after the task is activated by the parallel program.
 - In addition to performing computations, a process may synchronize or communicate with other processes, if needed.
 - In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks
-

A good Mapping (1)

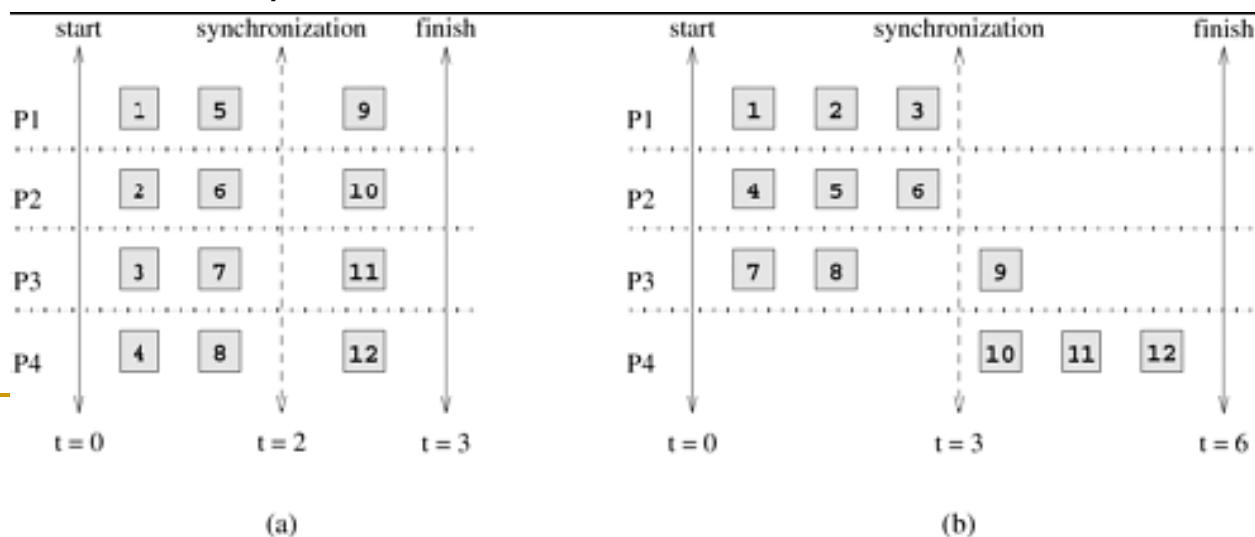
- Mapping = mechanism by which tasks are assigned to processes for execution
 - The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good mapping
 - A good mapping should seek to
 1. Max. the use of concurrency by mapping independent tasks onto different processes,
 2. Min. the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become executable
 3. Min. interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process.
 - In most nontrivial parallel algorithms, these three tend to be conflicting goals.
 - Example: the most efficient decomposition-mapping combination is a single task mapped onto a single process. It wastes no time in idling or interacting, but achieves no speedup either.
 - Mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel algorithm is.
 - Even though the degree of concurrency is determined by the decomposition, it is the mapping that det. how much of that concurrency is actually utilized & how efficiently.
-

A good mapping (2)

- Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the obj that all tasks complete in the shortest amount of elapsed time.
 - To achieve a small exec.time, overheads of executing the tasks in parallel must be minim.
 - For a given decomposition, there are two key sources of overhead.
 1. The time spent in inter-process interaction.
 2. The time that some processes may spend being idle.
 - Processes can be idle even before the overall comput. is finished for a variety of reasons:
 - Uneven load distribution may cause some processes to finish earlier than others.
 - All the unfinished tasks mapped onto a process may be waiting for tasks mapped onto other processes to finish in order to satisfy the constraints imposed by the task-dependency graph.
 - Both interaction and idling are often a function of mapping.
 - A good mapping of tasks onto processes must strive to achieve the objs of reducing
 1. the amount of time processes spend in interacting with each other,
 2. the total amount of time some processes are idle while the others are performing some tasks.
 - These two objectives often conflict with each other.
 - Example:
 - Min. the interactions can be easily achieved by assigning sets of tasks that need to interact with each other onto the same process.
 - Such a mapping will result in a highly unbalanced workload among the processes.
 - Following this strategy to the limit will often map all tasks onto a single process.
 - Processes with a lighter load will be idle when those with a heavier load are finishing their tasks.
 - To balance the load among processes => assign tasks that interact heavily to different processes.
-

Example: Two mappings of a hypothetical decomposition with a synchronization

- A task-dependency graph determines which tasks can execute in parallel and which must wait for some others to finish at a given stage in the execution of a parallel algorithm
- Poor synchronization among interacting tasks can lead to idling if one of the tasks has to wait to send or receive data from another task.
- A good mapping ensure that the computations and interactions among processes at each stage of the execution of the par. alg. are well balanced.
- Fig. shows two mappings of 12-task decomposition in which the last 4 tasks can be started only after the first 8 are finished due to dependencies among tasks.
 - As the figure shows, two mappings, each with an overall balanced workload, can result in different completion times.



Mapping classification

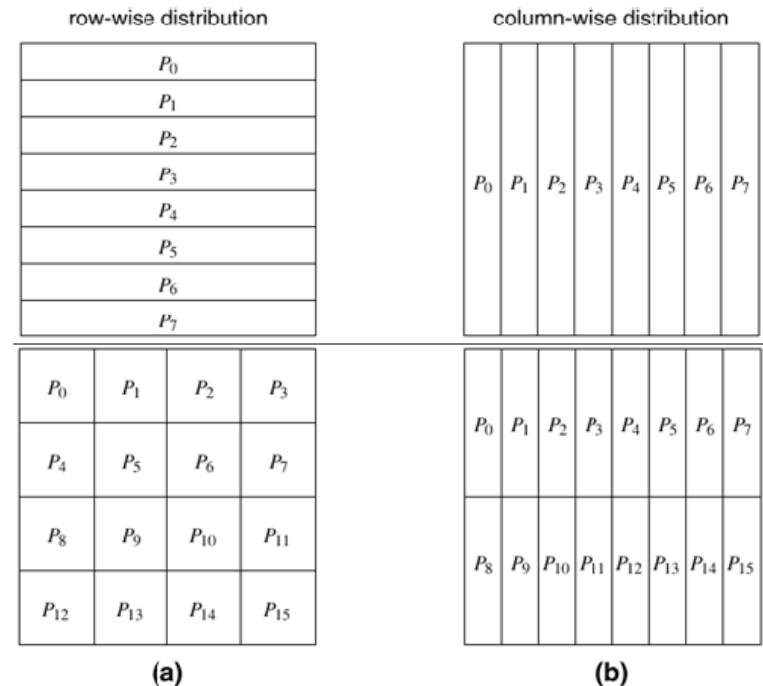
- The choice of a good mapping in this case depends on several factors, including
 - the knowledge of task sizes,
 - the size of data associated with tasks,
 - the characteristics of inter-task interactions, and
 - the parallel programming paradigm.
- Obtaining an optimal mapping is an NP-complete problem for non-uniform tasks.
- 1. **Static Mapping:**
 - Distribute the tasks among processes prior to the execution of the algorithm.
 - For statically generated tasks, either static or dynamic mapping can be used.
 - For many practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static mapping problem.
 - Algorithms that make use of static mapping are in general easier to design and program.
 - If task sizes are unknown, a static mapping can potentially lead to serious load-imbalances
- 2. **Dynamic Mapping:**
 - Distribute the work among processes during the execution of the algorithm.
 - If tasks are generated dynamically, then they must be mapped dynamically too.
 - If the amount of data associated with tasks is large relative to the computation, then a dynamic mapping may entail moving this data among processes.
 - In a shared-address-space paradigm, dynamic mapping may work well even with large data associated with tasks if the interaction is read-only.
 - Algs. that require dynamic mapping are complicated, particularly in the mes-pas progr. paradigm.

Static Mapping: Mappings Based on Data Partitioning

- Static mapping
 - is often-used in conjunction with a decomposition based on data partitioning.
 - used for mapping certain problems that are expressed naturally by a static task-dependency graph
 - In a decomposition based on partitioning data, the tasks are closely associated with portions of data by the owner-computes rule.
 - ⇒ mapping the relevant data onto the processes is equivalent to mapping tasks onto processes.
 - Two of the most common ways of representing data in algorithms:
 1. arrays and
 2. graphs.
-

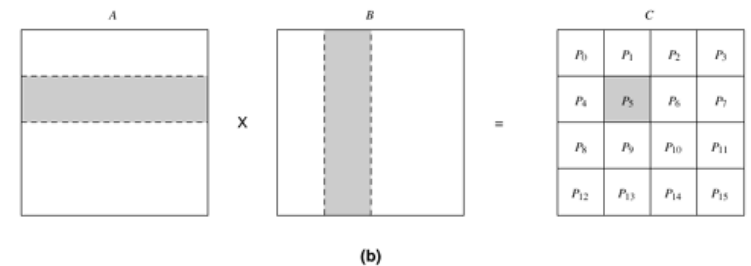
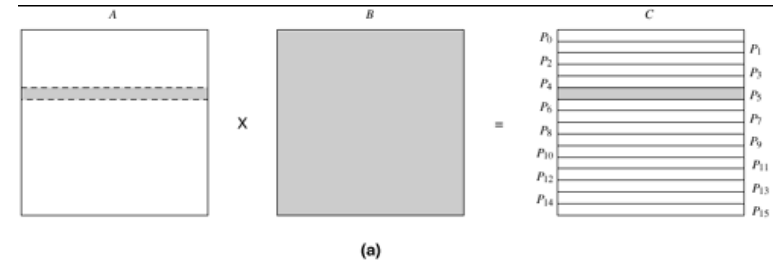
Block Distributions

- A d-dimensional array is distributed among the processes: each process receives a contiguous block of array entries along a specified subset of array dimensions.
- Block distributions of arrays are suitable when there is a locality of interaction, i.e., computation of an element of an array requires other nearby elements in the array.
- Example: 2-d array A with n rows and n columns.
 - Row-wise array distributions:
 - Partition the array into p parts such that the k th part contains rows $kn/p \dots (k+1)n/p - 1$, where $0 \leq k < p$.
 - Each partition contains a block of n/p consecutive rows
 - Column-wise array distributions:
 - Partition A along the second dimension, then each partition contains a block of n/p consecutive columns.
 - Two-dimensional distributions
 - Instead of selecting a single dim, select multiple dims
 - Select both dimensions and partition the matrix into blocks such that each block corresponds to a $n/p_1 \times n/p_2$ section of the matrix, with $p = p_1 \times p_2$ being the number of processes.
 - Fig. illustrates two different 2-d distributions, on a 4×4 and 2×8 process grid



Example 1: matrix multiplication

- Consider the $n \times n$ matrix multiplication $C = A \times B$.
- Partition the output matrix C .
- Balance the computations by partitioning C uniformly among the p available processes.
 1. 1-d block distribution: each process will get a block of n/p rows (or columns) of C ,
 2. 2-d block distribution: each process will get a block of size $n/\sqrt{p} \times n/\sqrt{p}$.
- Higher dim. distributions allow to use more processes.
 1. 1-d block distribution/ matrix-matrix x: up to n processes by assigning a single row of C to each process.
 2. 2-d distribution will allow us to use up to n^2 processes by assigning a single element of C to each process.
- Higher dim. distributions reduce the amount of interactions among the different processes for many problems.
 1. 1-d partitioning along the rows, each process needs to access the corresponding n/p rows of matrix A and the entire matrix B : the total amount of data that needs to be accessed is $n^2/p + n^2$.
 2. 2-d distribution: each process needs to access n/\sqrt{p} rows of matrix A and n/\sqrt{p} columns of matrix B : the total amount of shared data that each process needs to access is $O(n^2/\sqrt{p})$, $\ll O(n^2)$ in 1-d case.



Example 2: LU factorization

- If the amount of work differs for different elements of a matrix, a block distribution can potentially lead to load imbalances.
- Classic example of this phenomenon: **LU factorization** of a matrix: the amount of computation increases from the top left to the bottom right of the matrix.
- LU factorization alg. factors a nonsingular square matrix A into the product of
 - a lower triangular matrix L with a unit diagonal and
 - an upper triangular matrix U .
- The following alg. shows the serial column-based algorithm.
 - Matrices L and U share space with A : A is modified to store L and U in its lower and upper triangular parts,

- Serial:

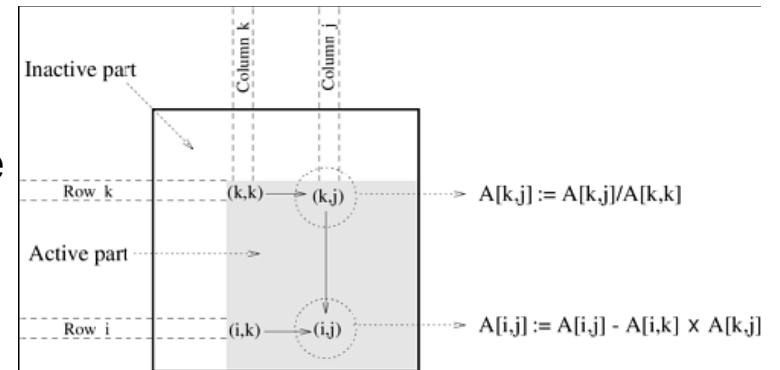
```
procedure COL_LU (A)
begin
  for k := 1 to n do
    for j := k to n do
      A[j, k] := A[j, k]/A[k, k];
    endfor;
    for j := k + 1 to n do
      for i := k + 1 to n do
        A[i, j] := A[i, j] - A[i, k] x A[k, j];
      endfor;
    endfor;
    /* After this iteration, column A[k + 1 : n,
       k] is logically the kth
       column of L and row A[k, k : n] is logically
       the kth row of U. */
  endfor;
end COL_LU
```

Block version of LU f.

- Fig. 1 shows a possible decomposition of LU factorization into 14 tasks using a 3 x 3 block partitioning of the matrix and using a block version
 - For each iteration of the outer loop $k := 1$ to n , the next nested loop in the above Alg goes from $k + 1$ to n .
- The active part of the matrix, as shown in Fig. 2, shrinks towards the bottom right corner of the matrix as the computation proceeds.
- In a block distribution, the processes assigned to the beginning rows & columns would perform far less work than those assigned to the later rows & cols.
- Computing different blocks of the matrix requires different amounts of work
 - Illustrated in Fig. 3.
 - Comp. value of $A_{1,1}$ requires only one task – Task 1.
 - Comp. value of $A_{3,3}$ requires 3 tasks – 9, 13, and 14.
- The process working on a block may idle even when there are unfinished tasks associated with that block.
 - This idling can occur if the constraints imposed by the task-dependency graph do not allow the remaining tasks on this process to proceed until one or more tasks mapped onto other processes are completed.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

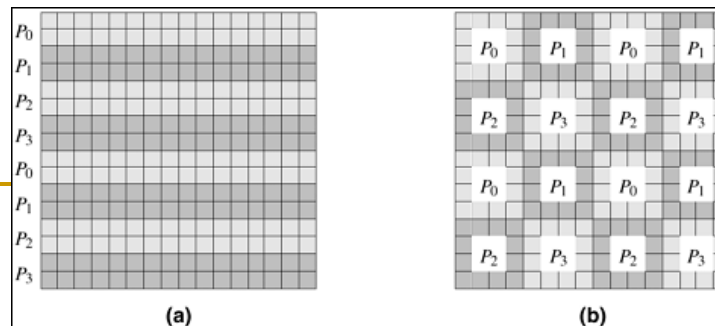
$$\begin{array}{l|l|l} 1: A_{1,1} \rightarrow L_{1,1}U_{1,1} & 6: A_{2,2} = A_{2,2} - L_{2,1}U_{1,2} & 11: L_{3,2} = A_{3,2}U_{2,2}^{-1} \\ 2: L_{2,1} = A_{2,1}U_{1,1}^{-1} & 7: A_{3,2} = A_{3,2} - L_{3,1}U_{1,2} & 12: U_{2,3} = L_{2,2}^{-1}A_{2,3} \\ 3: L_{3,1} = A_{3,1}U_{1,1}^{-1} & 8: A_{2,3} = A_{2,3} - L_{2,1}U_{1,3} & 13: A_{3,3} = A_{3,3} - L_{3,2}U_{2,3} \\ 4: U_{1,2} = L_{1,1}^{-1}A_{1,2} & 9: A_{3,3} = A_{3,3} - L_{3,1}U_{1,3} & 14: A_{3,3} \rightarrow L_{3,3}U_{3,3} \\ 5: U_{1,3} = L_{1,1}^{-1}A_{1,3} & 10: A_{2,2} \rightarrow L_{2,2}U_{2,2} & \end{array}$$



P₀ T ₁	P₃ T ₄	P₆ T ₅
P₁ T ₂	P₄ T ₆ T ₁₀	P₇ T ₈ T ₁₂
P₂ T ₃	P₅ T ₇ T ₁₁	P₈ T ₉ T ₁₃ T ₁₄

Cyclic and Block-Cyclic Distributions

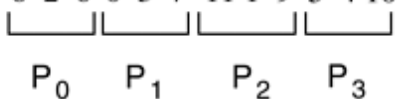
- The block-cyclic distribution is a variation of the block distribution scheme that can be used to alleviate the load-imbalance and idling problems.
- Central idea behind a block-cyclic distribution:
 - partition an array into many more blocks than the number of available processes.
 - assign the partitions (and the associated tasks) to processes in a round-robin manner so that each process gets several non-adjacent blocks.
- Example 1: a 1-d block-cyclic distribution of a matrix among p processes,
 - the rows (columns) of an $n \times n$ matrix are divided into αp groups of $n/(\alpha p)$ consecutive rows (columns), where $1 \leq \alpha \leq n/p$.
 - These blocks are distributed among the p processes in a wraparound fashion such that block bi is assigned to process $P_i \% p$ ('%' is the modulo operator).
 - Assigns α blocks of the matrix to each process, but each subsequent block that gets assigned to the same process is p blocks away.
- Example 2: a 2-d block-cyclic distribution of an $n \times n$ array by partitioning it into square blocks of size $\alpha \sqrt{p} \times \alpha \sqrt{p}$ and distributing them on a hypothetical $\sqrt{p} \times \sqrt{p}$ array of processes in a wraparound fashion.
- The block-cyclic distribution can be extended to arrays of higher dimensions.
- Fig illustrates 1-d & 2-d block cyclic distributions of a two-dimensional array.



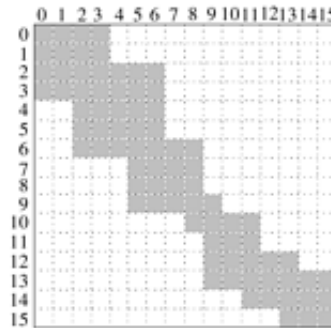
Randomized Block Distributions

- load balance is sought by partitioning the array into many more blocks than the number of available processes.
- the blocks are uniformly and randomly distributed among the processes

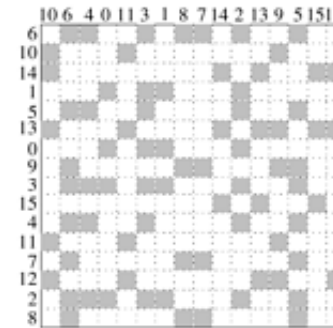
■ Ex. 1:

$V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$
 $\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$
 $\text{mapping} = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$


Ex. 2:



(a)



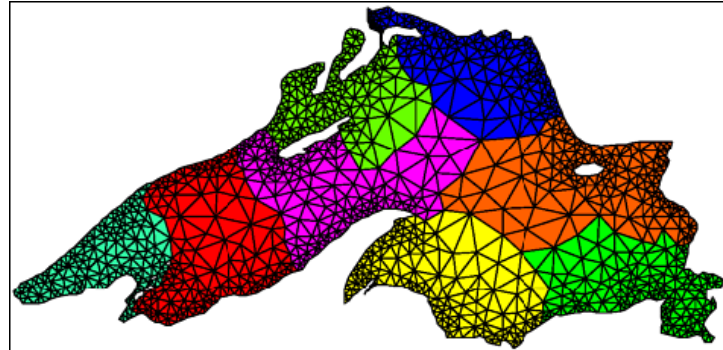
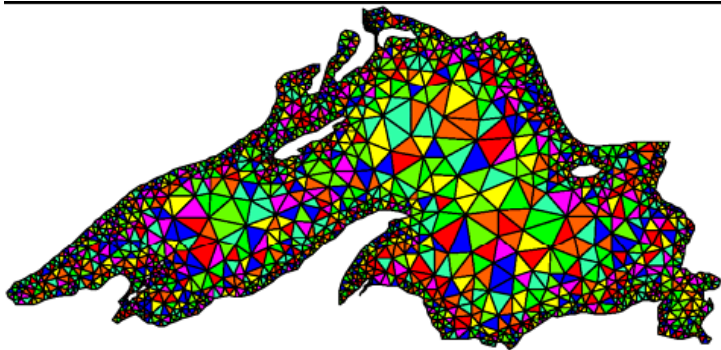
(b)

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

(c)

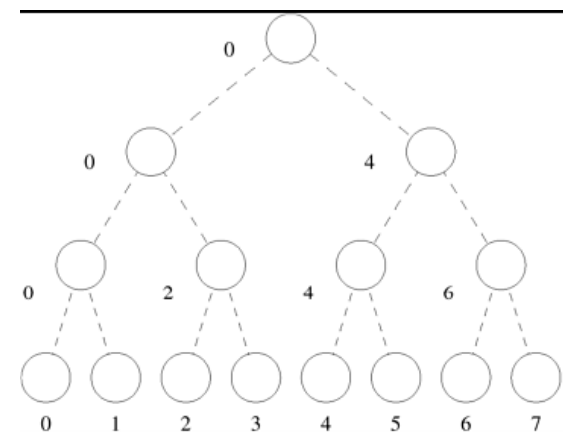
Graph partitioning

- there are many algorithms that operate on sparse datastructures and for which the pattern of interaction among data elements is data dependent and highly irregular.
 - Numerical simulations of physical phenomena provide a large source of such type of comps.
 - In these computations, the physical domain is discretized and represented by a mesh of elements.
 - Ex: Simulation of a physical phenomenon such the dispersion of a water contaminant in the lake involve computing the level of contamination at each vertex of this mesh at various intervals of time.
- Random: each process will need to access a large set of points belonging to other processes to complete computations for its assigned portion of the mesh.
- ? Partition the mesh into p parts such that each part contains roughly the same no. of mesh-points or vertices, & no. of edges that cross partition boundaries is minimized.
 - NP-complete problem.
 - Algorithms that employ powerful heuristics are available to compute reasonable partitions.
 - Each process is assigned a contiguous region of the mesh such that the total number of mesh points that needs to be accessed across partition boundaries is minimized.



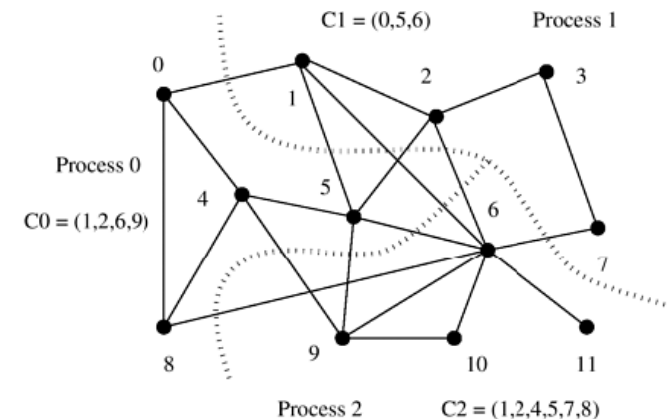
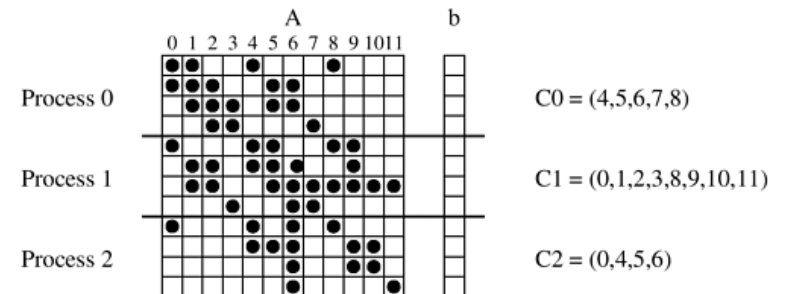
Static mapping: Mappings Based on Task Partitioning

- When the computation is naturally expressible in the form of a static task-dependency graph with tasks of known sizes
- Conflicting objectives of minimizing idle time and minimizing the interaction time of the parallel algorithm
- Determining an optimal mapping for a general task-dependency graph is an NP-complete problem
- Ex:
 - task-dependency graph that is a perfect binary tree (e.g. finding minimum)
 - mapping on a hypercube
 - minimizes the interaction overhead by mapping many interdependent tasks onto the same process (i.e., the tasks along a straight branch of the tree) and others on processes only one communication link away from each other.



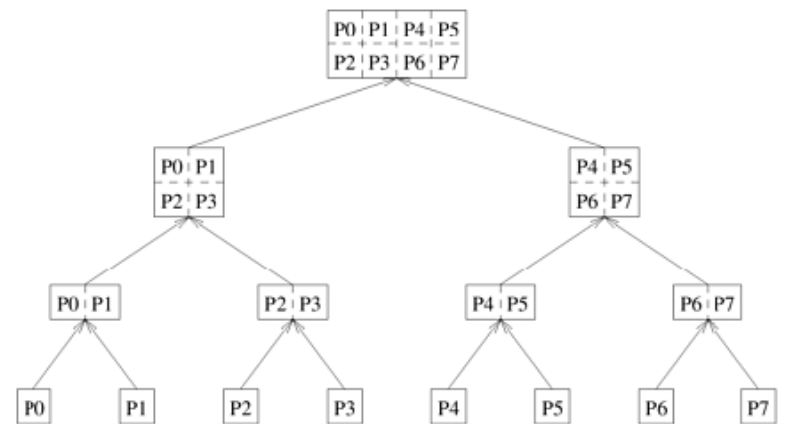
A solution: partition the task-interaction graph

- An approximate solution
- Ex: A mapping of three for sparse matrix-vector multiplication onto three processes
 - First: this mapping assigns tasks corresponding to four consecutive entries of b to each process.
 - Second: partitioning for the task interaction graph
 - C_i contains the indices of b that the tasks on Process i need to access from tasks mapped onto other processes
 - A quick comparison of the lists C_0 , C_1 , and C_2 in the two cases readily reveals that the second mapping entails fewer exchanges of elements of b between processes than the first mapping.



Static Mapping: Hierarchical Mappings

- a mapping based solely on the task-dependency graph may suffer from load-imbalance or inadequate concurrency
- If the tasks are large enough, then a better mapping can be obtained by a further decomposition of the tasks into smaller subtasks.
- Ex 1: Quicksort has a task-dependency graph that is an ideal candidate for a hierarchical mapping
- Ex 2 - figure



Dynamic Mapping

- necessary in situations
 - where a static mapping may result in a highly imbalanced distribution of work among processes or
 - where the task-dependency graph itself is dynamic, thus precluding a static mapping.
 - the primary reason for using a dynamic mapping is balancing the workload among processes,
 - dynamic mapping is often referred to as dynamic load-balancing.
 - Dynamic mapping techniques are classified:
 - centralized or
 - distributed.
-

Dynamic Mapping: Centralized Schemes

- all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes.
 - a special process is designated to manage the pool of available tasks, then it is often referred to as the *master*
 - the other processes that depend on the master to obtain work are referred to as *slaves*
 - whenever a process has no work, it takes a portion of available work from the central data structure or the master process.
 - whenever a new task is generated, it is added to this centralized data structure or reported to the master process.
 - centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability.
 - the large no. accesses to the common data structure or the master process tends to become a bottleneck.
 - Example:
 - the problem of sorting the entries in each row of an $n \times n$ matrix A .
 - A naive mapping - an equal no. rows to each process - may lead to load-imbalance.
 - Another solution: maintain a central pool of indices of the rows that have yet to be sorted.
 - Whenever a process is idle, it picks up an available index, deletes it, and sorts the row with that index
 - Scheduling the independent iterations of a loop among parallel processes is known as self scheduling.
-

Dynamic Mapping: Distributed Schemes

- the set of executable tasks are distributed among processes which exchange tasks at run time to balance work
 - each process can send work to or receive work from any other process
 - the critical parameters of a distributed load balancing scheme are as follows:
 - How are the sending and receiving processes paired together?
 - Is the work transfer initiated by the sender or the receiver?
 - How much work is transferred in each exchange?
 - When is the work transfer performed?
-

Methods for Containing Interaction Overheads

- Reducing the interaction overhead among concurrent tasks is important for an efficient parallel program.
 - The overhead that a parallel program incurs due to interaction among its processes depends on many factors:
 - the volume of data exchanged during interactions,
 - the frequency of interaction,
 - the spatial and temporal pattern of interactions, etc.
 - Some general techniques that can be used to reduce the interaction overheads incurred by parallel programs:
 1. Maximizing Data Locality and Minimizing Contention and Hot Spots
 2. Overlapping Computations with Interactions
 3. Replicating Data or Computations
 4. Using Optimized Collective Interaction Operations
 5. Overlapping Interactions with Other Interactions
-

Max. Data Locality & Min. Contention and Hot Spots

- Data locality enhancing techniques encompass a wide range of schemes that try to
 - minimize the volume of nonlocal data that are accessed,
 - maximize the reuse of recently accessed data, and
 - minimize the frequency of accesses.
 - Minimize Volume of Data-Exchange
 - by using appropriate decomposition and mapping schemes.
 - Example: matrix multiplication
 - using a two-dimensional mapping of the computations to the processes we reduce the amount of shared data that needs to be accessed by each task as opposed to a one-dimensional mapping
 - Minimize Frequency of Interactions
 - by restructuring the algorithm such that shared data are accessed and used in large pieces
 - Minimizing Contention and Hot Spots
 - contention occurs when
 - multiple tasks try to access the same resources concurrently.
 - multiple simultaneous transmissions of data over the same interconnection link,
 - multiple simultaneous accesses to the same memory block,
 - multiple processes sending messages to the same process at the same time
-

Overlapping Computations with Interactions

- There are a no. techniques that can be used
 - The simplest: initiate an interaction early enough so that it is completed before it is needed for computation.
 - need to identify computations that can be performed before the interaction and do not depend on it
 - Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original alg.
 - In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process.
 - If the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance
 - On a shared-address-space arch, assisted by prefetching hardware.
 - The prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed.
-

Replicating Data or Computations

■ Data replication

□ Example:

- multiple processes may require frequent read-only access to shared data structure, such as a hash-table, in an irregular pattern.
 - replicate a copy of the shared data structure on each process
 - after the initial interaction during replication, all subsequent accesses to this data structure are free of any interaction overhead

□ Increases the memory requirements of a parallel program

- The aggregate amount of memory required to store the replicated data increases linearly with the no. concurrent processes.
- This may limit the size of the problem that can be solved on a given parallel computer.

■ Computation replication

□ the processes in a parallel program often share intermediate results.

- In some situations, it may be more cost-effective for a process to compute these intermediate results than to get them from another process that generates them.

□ Example:

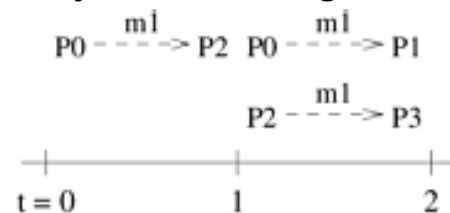
- Fast Fourier Transform, on an N-point series, N distinct powers of ω or "twiddle factors" are computed and used at various points in the computation.
 - In a parallel implementation of FFT, different processes require overlapping subsets of these N twiddle factors.
 - Message-passing paradigm: each process locally compute all the twiddle factors it needs.
 - Although the parallel algorithm may perform many more twiddle factor computations than the serial algorithm, it may still be faster than sharing the twiddle factors.
-

Using Optimized Collective Interaction Operations

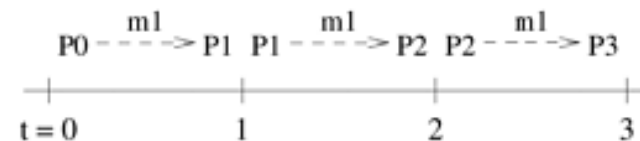
- Often the interaction patterns among concurrent activities are static and regular.
 - A class of such static and regular interaction patterns are those that are performed by groups of tasks, and they are used to achieve regular data accesses or to perform certain type of computations on distributed data.
 - A number of key such collective interaction operations have been identified that appear frequently in many parallel algorithms.
 - Examples:
 - Broadcasting some data to all the processes or
 - adding up numbers, each belonging to a different process
 - The collective data-sharing operations can be classified into three categories.
 1. operations that are used by the tasks to access data,
 2. operations are used to perform some communication-intensive computations,
 3. used for synchronization.
 - Highly optimized implementations of these collective operations have been developed that minimize the overheads due to data transfer as well as contention.
 - Optimized implementations of these collective interaction operations are available in library form from the vendors of most parallel computers, e.g., MPI (message passing interface).
 - the algorithm designer does not need to think about how these operations are implemented and needs to focus only on the functionality achieved by these operations.
-

Overlapping Interactions with Other Interactions

- Overlapping interactions between multiple pairs of processes can reduce the effective volume of communication.
- Example:
 - the commonly used collective communication operation of one-to-all broadcast in a message-passing paradigm with four processes P0, P1, P2, and P3.
 - A commonly used algorithm to broadcast some data from P0 to all other processes works as follows.
 - In the first step, P0 sends the data to P2.
 - In the second step, P0 sends the data to P1, and concurrently, P2 sends the same data that it had received from P0 to P3.
 - The entire operation is thus complete in two steps because the two interactions of the second step require only one time step.
 - This operation is illustrated in Figure (a).
 - A naive broadcast algorithm would send the data from P0 to P1 to P2 to P3, thereby consuming three steps as illustrated in Figure (b).



(a)



(b)