ĐẠI HỌC

BÁCH KHOA

25 YEARS ANNIVERSARY

SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Parallel Programming Issues

References

- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **. Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

7.1 Parallel Model and Domain Decomposition

Some Serial Algorithms

Working Examples

- Dense Matrix-Matrix & Matrix-Vector Multiplication
- Sparse Matrix-Vector Multiplication
- Floyd’s All-pairs Shortest Path
- Minimum/Maximum Finding
- Heuristic Search—15-puzzle problem

Dense Matrix-Vector Multiplication

```
procedure MAT_VECT (A, x, y)
begin
    for i := 0 to n − 1 do
    begin
        y[i] := 0;
        for j := 0 to n − 1 do
            y[i] := y[i] + A[i, j] × x[j];
    endfor;
end MAT_VECT
```

A serial algorithm for multiplying an $n \times n$ matrix A with an $n \times 1$ vector x to yield an $n \times 1$ product vector y.

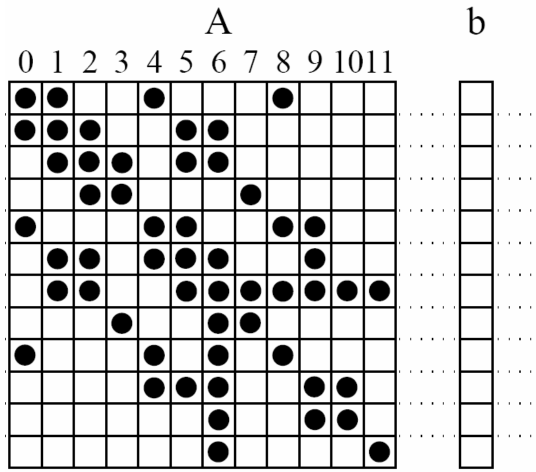
Dense Matrix-Matrix Multiplication

```
procedure MAT_MULT (A, B, C)
begin
    for i := 0 to n − 1 do
        for j := 0 to n − 1 do
            begin
                C[i, j] := 0;
                for k := 0 to n − 1 do
                    C[i, j] := C[i, j] + A[i, k] × B[k, j];
            endfor;
end MAT_MULT
```

The conventional serial algorithm for multiplication of two $n \times n$ matrices.

Sparse Matrix-Vector Multiplication

$$y = Ab$$



$$y[i] = \sum_{j=1}^{n} (A[i, j] \times b[j])$$

Floyd's All-Pairs Shortest Path

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if} \quad k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & \text{if} \quad k \geq 1 \end{cases}$$

```
procedure FLOYD_ALL_PAIRS_SP(A)
begin
    D^(0) = A;
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                d_{i,j}^(k) := min( d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1) );
end FLOYD_ALL_PAIRS_SP
```

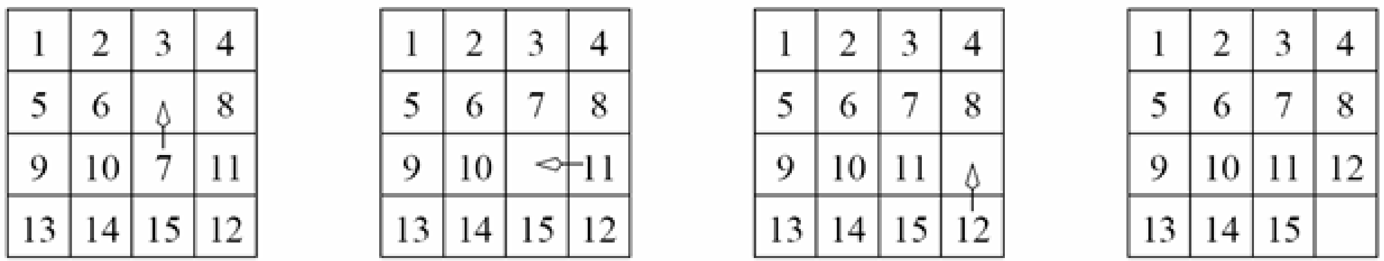
Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph $G = (V, E)$ with adjacency matrix A.

Minimum Finding

```
procedure SERIAL_MIN (A, n)
begin
min = A[0];
for i := 1 to n − 1 do
    if (A[i] < min) min := A[i];
endfor;
return min;
end SERIAL_MIN
```

A serial program for finding the minimum in an array of numbers A of length n.

15—Puzzle Problem



(a) (b) (c) (d)

A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

Parallel Algorithm *vs* Parallel Formulation

- *Parallel Formulation*
  - Refers to a *parallelization* of a serial algorithm.
- *Parallel Algorithm*
  - May represent an entirely different algorithm than the one used serially.

- We primarily focus on *"Parallel Formulations"*
  - Our goal today is to primarily discuss how to develop such parallel formulations.
  - Of course, there will always be examples of "parallel algorithms" that were not derived from serial algorithms.

Elements of a Parallel Algorithm/Formulation

- Pieces of work that can be done concurrently
  - tasks
- Mapping of the tasks onto multiple processors
  - processes *vs* processors
- Distribution of input/output & intermediate data across the different processors
- Management the access of shared data
  - either input or intermediate
- Synchronization of the processors at various points of the parallel execution

Note:

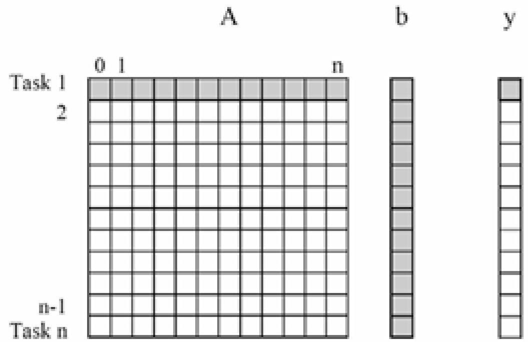
Maximize concurrency and reduce overheads due to parallelization!

Maximize potential speedup!

Finding Concurrent Pieces of Work

- 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

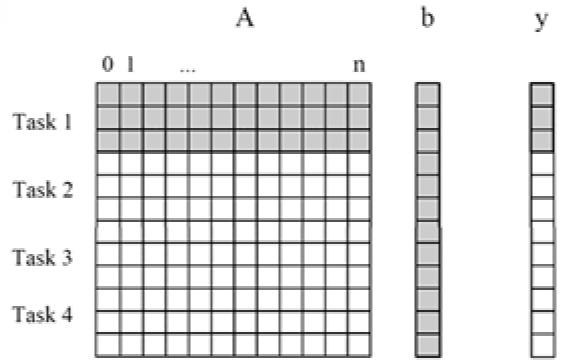
Example: Dense Matrix-Vector Multiplication



Decomposition of dense matrix-vector multiplication into n tasks, where n is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

Tasks can be of different size.

- *granularity of a task*



Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

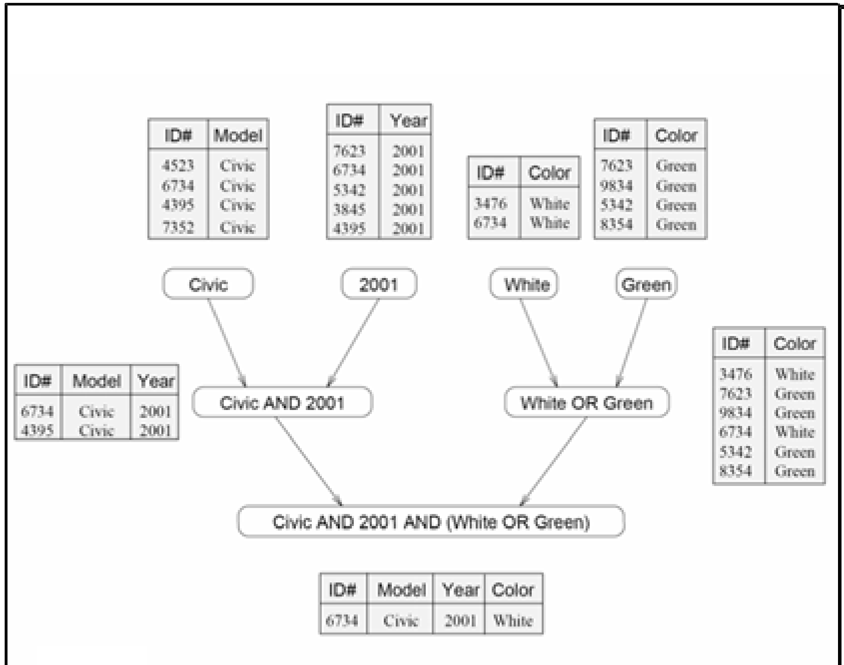
Example: Query Processing

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

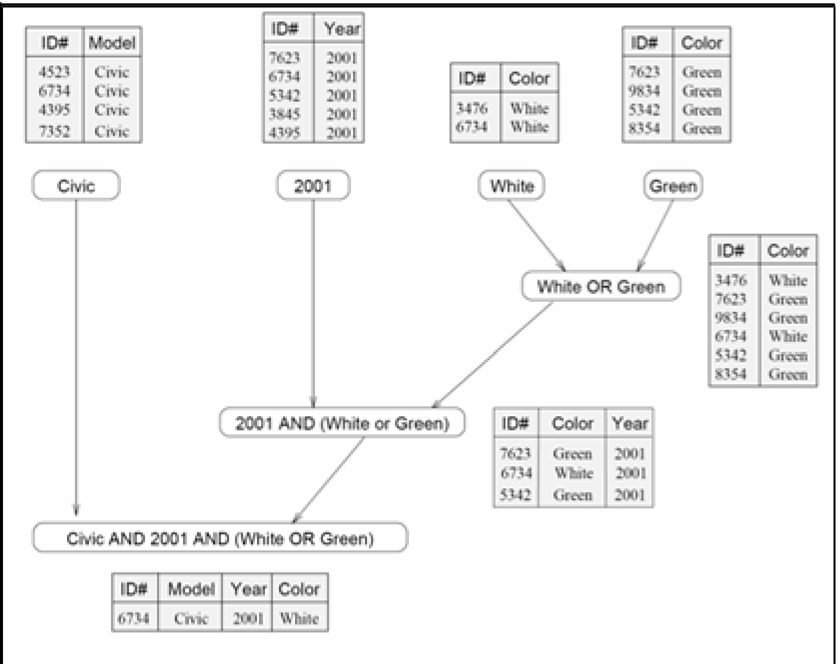
Query: MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")

Example: Query Processing

- Finding concurrent tasks…



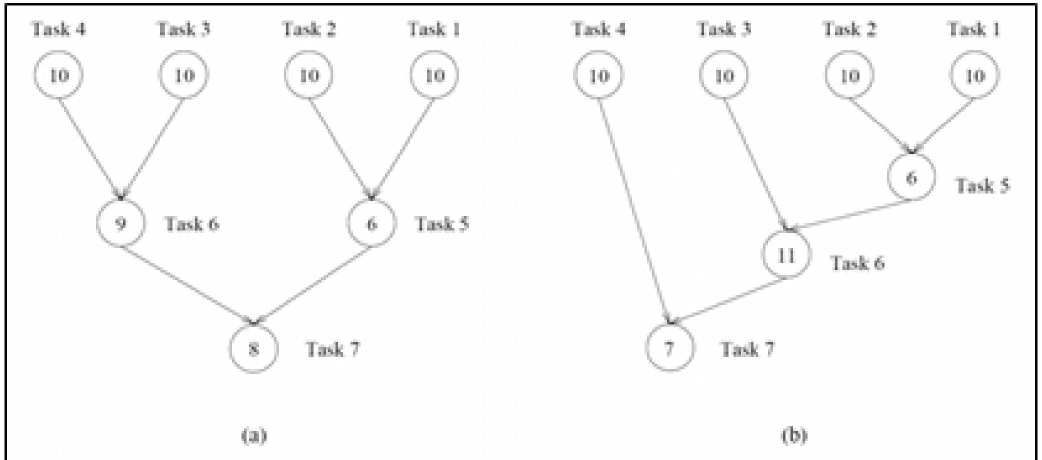
The different tables and their dependencies in a query processing operation.



An alternate data-dependency graph for the query processing operation.

Task-Dependency Graph

- In most cases, there are dependencies between the different tasks
  - certain task(s) can only start once some other task(s) have finished
    - e.g., producer-consumer relationships
- These dependencies are represented using a DAG called *task-dependency graph*



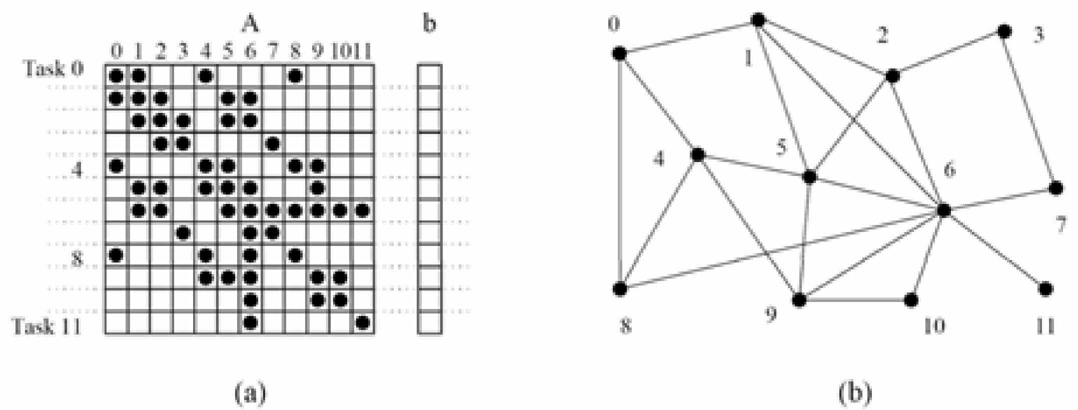
Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

Task-Dependency Graph (cont)

- Key Concepts Derived from the Task-Dependency Graph
  - Degree of Concurrency
    - The number of tasks that can be concurrently executed
      - we usually care about the *average* degree of concurrency
  - Critical Path
    - The longest vertex-weighted path in the graph
      - The weights represent task size
  - Task granularity affects both of the above characteristics

Task-Interaction Graph

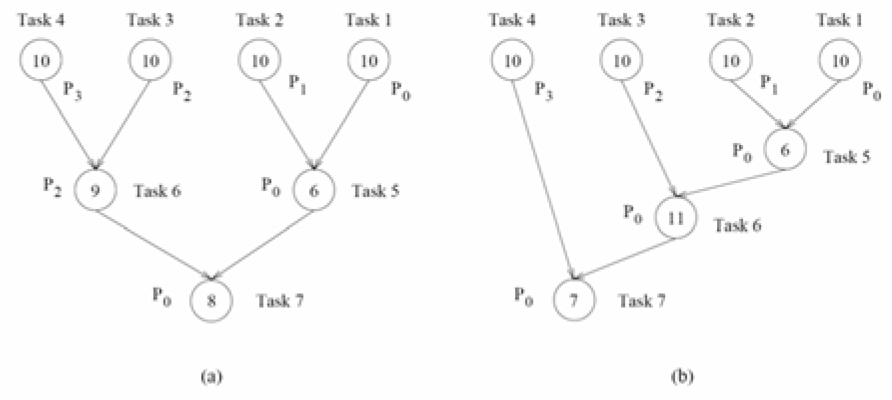
- Captures the pattern of interaction between tasks
  - This graph usually contains the task-dependency graph as a *subgraph*
    - i.e., there may be interactions between tasks even if there are no dependencies
      - these interactions usually occur due to accesses on shared data



A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \ne 0} A[i, j].b[j]$.

Task Dependency/Interaction Graphs

- These graphs are important in developing effectively mapping the tasks onto the different processors
  - Maximize concurrency and minimize overheads



Mappings of the task graphs of Figure 3.5 onto four processes.

- More on this later…

Common Decomposition Methods

- Data Decomposition
- Recursive Decomposition
- Exploratory Decomposition
- Speculative Decomposition
- Hybrid Decomposition

Task decomposition methods

Recursive Decomposition

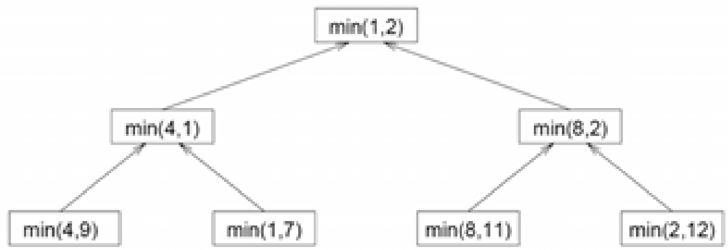
- Suitable for problems that can be solved using the divide-and-conquer paradigm
- Each of the *subproblems* generated by the *divide* step becomes a task

Example: Finding the Minimum

- Note that we can obtain divide-and-conquer algorithms for problems that are traditionally solved using non-divide-and-conquer approaches

```
procedure RECURSIVE_MIN (A, n)
begin
if (n = 1) then
    min := A[0];
else
    lmin := RECURSIVE_MIN (A, n/2);
    rmin := RECURSIVE_MIN (&(A[n/2]), n − n/
    if (lmin < rmin) then
        min := lmin;
    else
        min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

Algorithm A recursive program for finding the minimum in an array of numbers A of length n.



The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.

Recursive Decomposition

- How good are the decompositions that it produces?
  - average concurrency?
  - critical path?
- How do the quicksort and min-finding decompositions measure-up?

Data Decomposition

- Used to derive concurrency for problems that operate on large amounts of data
- The idea is to derive the tasks by focusing on the multiplicity of data
- Data decomposition is often performed in two steps
  - Step 1: Partition the data
  - Step 2: Induce a computational partitioning from the data partitioning
- Which data should we partition?
  - Input/Output/Intermediate?
    - Well… all of the above—leading to different data decomposition methods
- How do induce a computational partitioning?
  - Owner-computes rule

Example: Matrix-Matrix Multiplication

- Partitioning the output data

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Example: Matrix-Matrix Multiplication

- Partitioning the intermediate data

Stage I

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

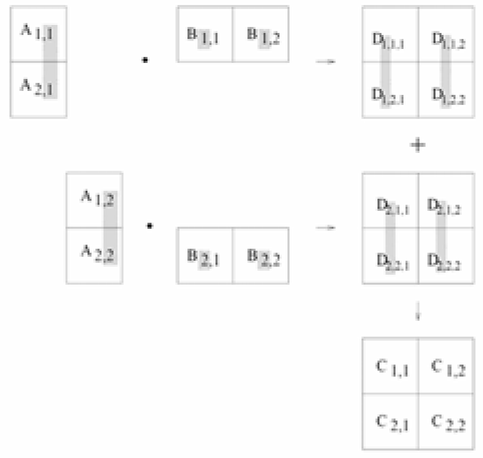
Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

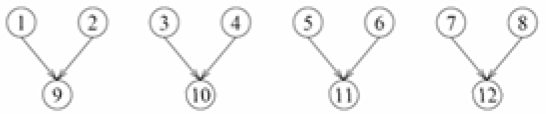
A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.



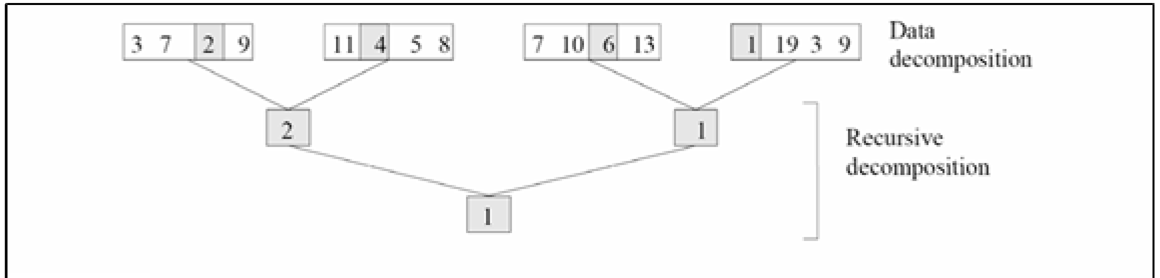
Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D.



The task-dependency graph of the decomposition

Data Decomposition

- Is the most widely-used decomposition technique
  - after all parallel processing is often applied to problems that have a lot of data
  - splitting the work based on this data is the natural way to extract high-degree of concurrency
- It is used by itself or in conjunction with other decomposition methods
  - Hybrid decomposition

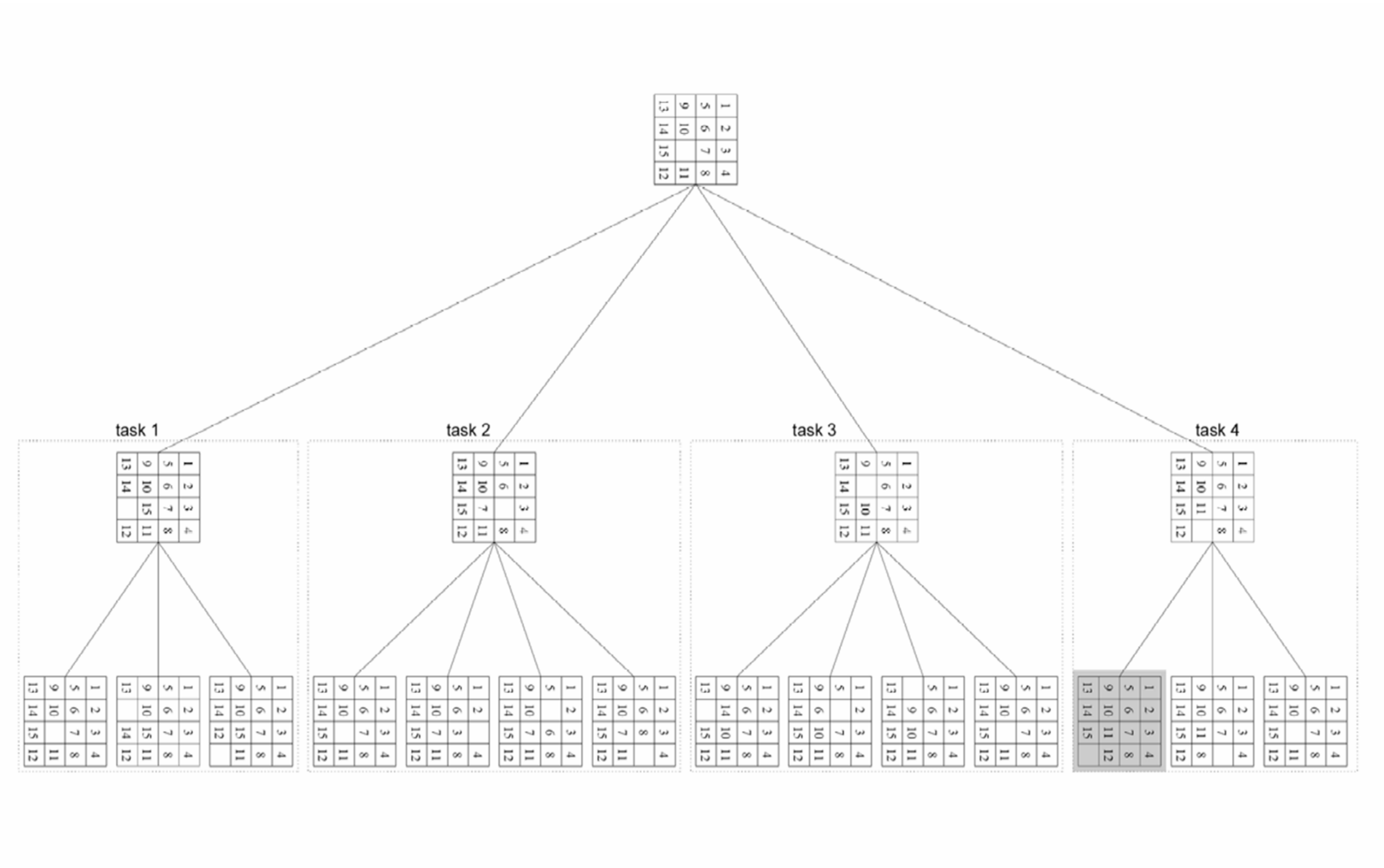


Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.

Exploratory Decomposition

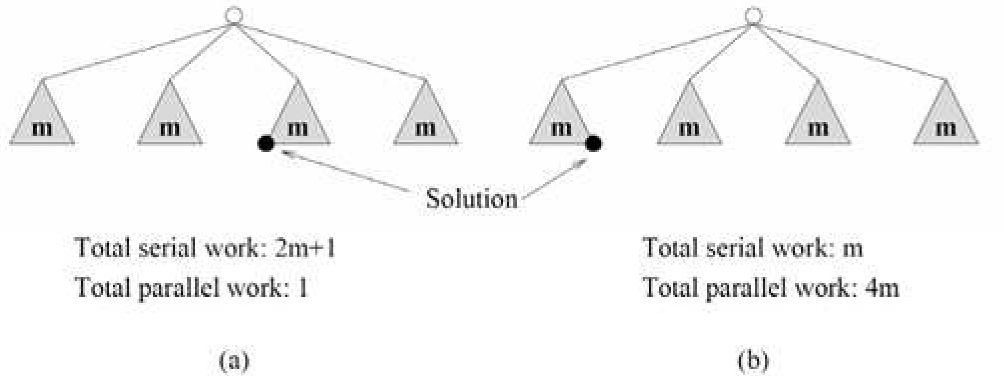
- Used to decompose computations that correspond to a search of a space of solutions

Example: 15-puzzle Problem



Exploratory Decomposition

- It is not as general purpose
- It can result in speedup anomalies
  - *engineered* slow-down or superlinear speedup

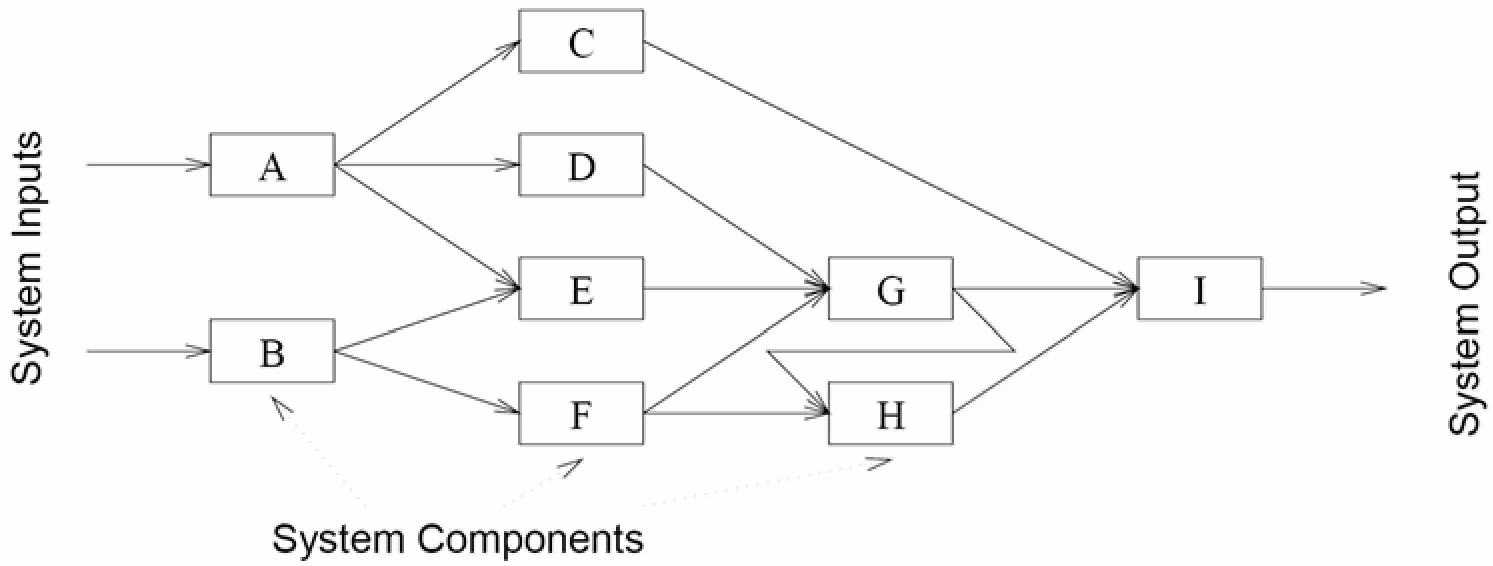


An illustration of anomalous speedups resulting from exploratory decomposition.

Speculative Decomposition

- Used to extract concurrency in problems in which the *next step* is one of many possible actions that can only be determined when the current tasks finishes
- This decomposition assumes a certain *outcome* of the currently executed task and *executes* some of the next steps
  - Just like speculative execution at the microprocessor level

Example: Discrete Event Simulation



A simple network for discrete event simulation.

Speculative Execution

- If predictions are wrong…
  - work is wasted
  - work may need to be *undone*
    - state-restoring overhead
      - memory/computations
- However, it may be the only way to extract concurrency!

Mapping the Tasks

- Why do we care about task mapping?
  - Can I just randomly assign them to the available processors?
- Proper mapping is critical as it needs to minimize the parallel processing overheads
  - If T_p is the parallel runtime on p processors and T_s is the serial runtime, then the *total overhead* T_o is $p*T_p - T_s$
    - The work done by the parallel system beyond that required by the serial system
  - Overhead sources:
    - Load imbalance
    - Inter-process communication
      - coordination/synchronization/data-sharing

they can be at odds with each other

remember the holy grail…

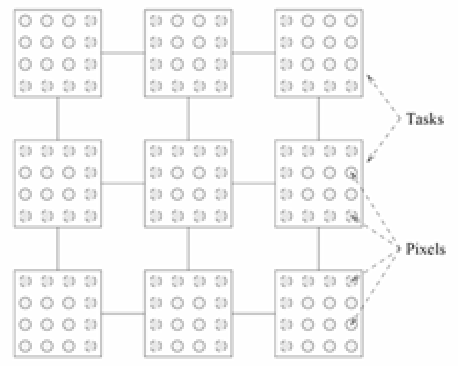
Why Mapping can be Complicated?

- Proper mapping needs to take into account the task-dependency and interaction graphs
  - Are the tasks available a priori?
    - Static vs dynamic task generation
  - How about their computational requirements?
    - Are they uniform or non-uniform?
    - Do we know them a priori?
  - How much data is associated with each task?
  - How about the interaction patterns between the tasks?
    - Are they static or dynamic?
    - Do we know them a priori?
    - Are they data instance dependent?
    - Are they regular or irregular?
    - Are they read-only or read-write?
- Depending on the above characteristics different mapping techniques are required of different complexity and cost

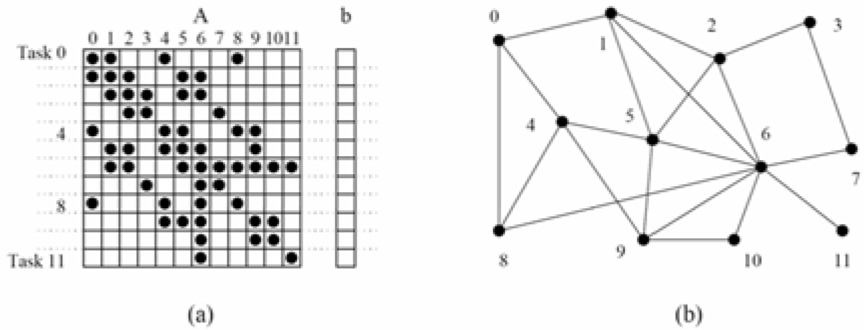
Task dependency graph

Task interaction graph

Example: Simple & Complex Task Interaction



The regular two-dimensional task-interaction graph for image dithering. The pixels with dotted outline require color values from the boundary pixels of the neighboring tasks.

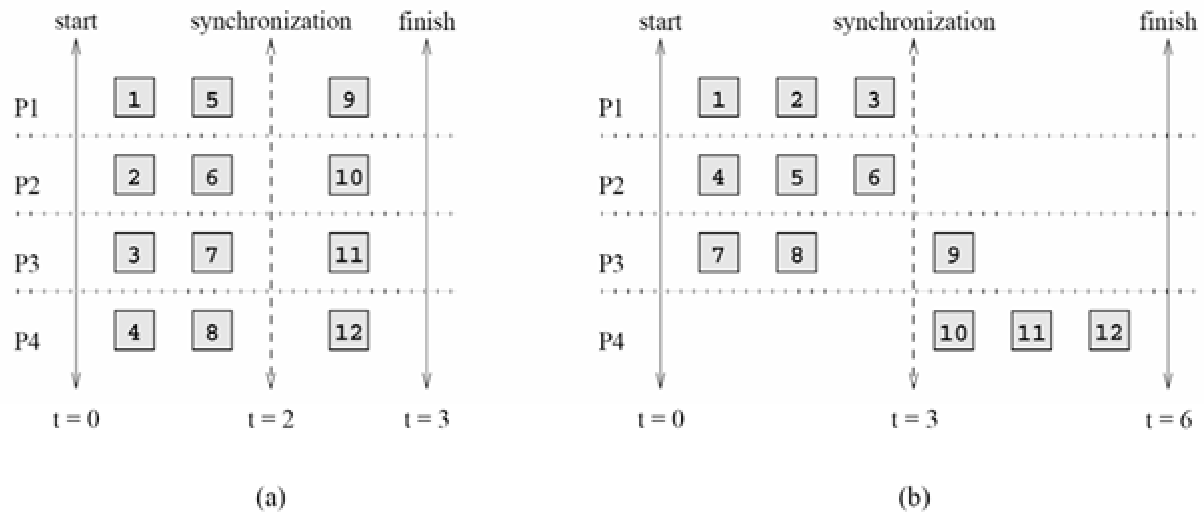


A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \neq 0} A[i,j].b[j]$.

Mapping Techniques for Load Balancing

- Be aware…
  - The assignment of tasks whose aggregate computational requirements are the same does not automatically ensure load balance.

Each processor is assigned three tasks but (a) is better than (b)!



Two mappings of a hypothetical decomposition with a synchronization.

Load Balancing Techniques

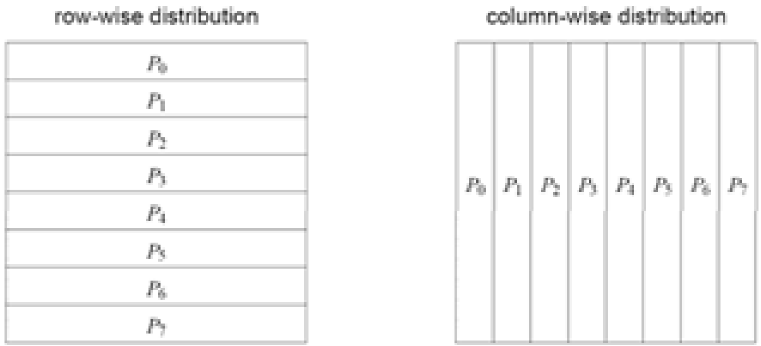
- Static
  - The tasks are distributed among the processors prior to the execution
  - Applicable for tasks that are
    - generated statically
    - known and/or uniform computational requirements
- Dynamic
  - The tasks are distributed among the processors during the execution of the algorithm
    - i.e., tasks & data are migrated
  - Applicable for tasks that are
    - generated dynamically
    - unknown computational requirements

Static Mapping—Array Distribution

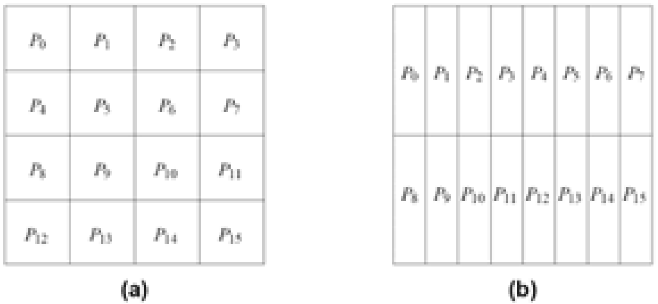
- Suitable for algorithms that
  - use data decomposition
  - their underlying input/output/intermediate data are in the form of arrays
- Block Distribution
- Cyclic Distribution
- Block-Cyclic Distribution
- Randomized Block Distributions

1D/2D/3D

Examples: Block Distributions

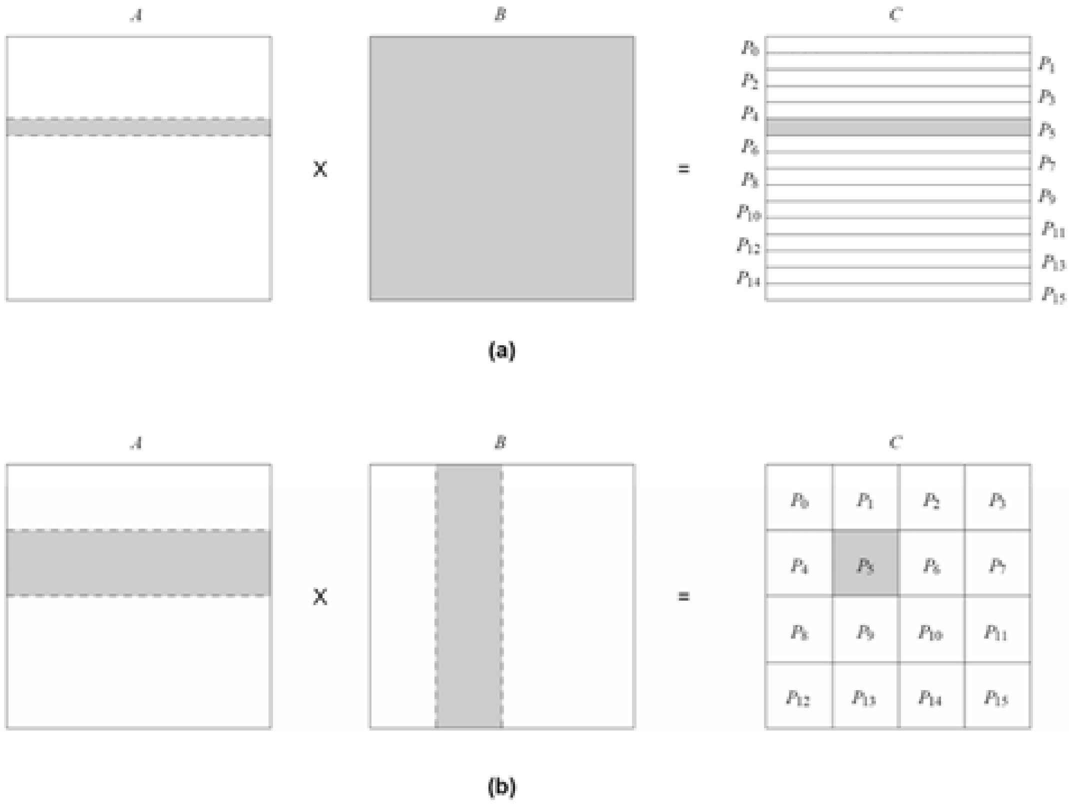


Examples of one-dimensional partitioning of an array among eight processes.



Examples of two-dimensional distributions of an array, (a) on a 4×4 process grid, and (b) on a 2×8 process grid.

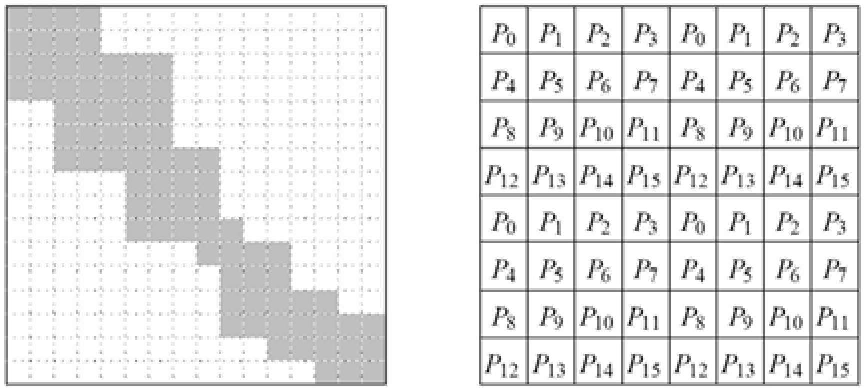
Examples: Block Distributions



Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

Random Block Distributions

- Sometimes the computations are performed only at certain portions of an array
  - sparse matrix-matrix multiplication

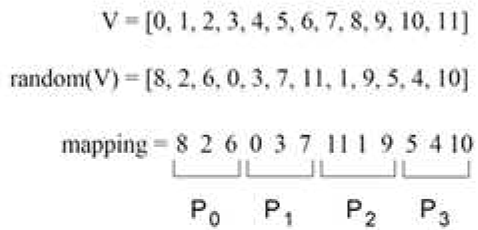


(a) (b)

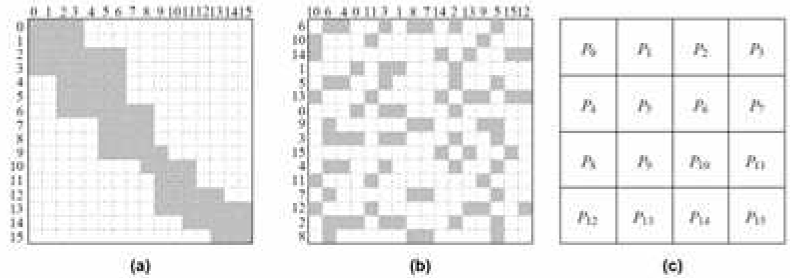
Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

Random Block Distributions

- Better load balance can be achieved via a random block distribution



A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., $\alpha = 3$).



Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

Dynamic Load Balancing Schemes

- There is a huge body of research
  - Centralized Schemes
    - A certain processors is responsible for giving out work
      - master-slave paradigm
    - Issue:
      - task granularity
  - Distributed Schemes
    - Work can be transferred between any pairs of processors.
    - Issues:
      - How do the processors get paired?
      - Who initiates the work transfer? push vs pull
      - How much work is transferred?

Mapping to Minimize Interaction Overheads

- Maximize data locality
- Minimize volume of data-exchange
- Minimize frequency of interactions
- Minimize contention and hot spots
- Overlap computation with interactions
- Selective data and computation replication

Achieving the above is usually an interplay of decomposition and mapping and is usually done iteratively

7.2 Dependency in parallel computing

- Definition
- Types of dependency
- Solution
- Example

What is data dependency?

- A data dependency is a situation in which a program statement (instruction) refers to the data of a preceding statement
- A dependency exists between statements when order of statement execution affects the results of program
- A data dependency results from multiple use of the same location(s) in storage by different tasks
- In parallel computing: a **data dependency** consist of a situation in which calculation on this thread/core/cpu/node use data calculated by other thread/core/cpu/node and/or use data stored in memory managed by other thread/core/cpu/node

Types of data dependency

TYPE	NOTATION	DESCRIPTION
True (Flow) Dependence	S1 ->T S2	A true dependence between S1 and S2 means that S1 writes to a location later read from by S2
Anti Dependence	S1 ->A S2	An anti-dependence between S1 and S2 means that S1 reads from a location later written to by S2.(before)
Output Dependence	S1 ->I S2	An input dependence between S1 and S2 means that S1 and S2 read from the same location.

Types of data dependency

True dependence

S0: int a, b;

S1: a = 2;

S2: b = a + 40;

S1 ->T S2, meaning that S1 has a true dependence on S2 because S1writes to the variable a, which S2 reads from.

Anti-dependence

S0: int a, b = 40;

S1: a = b - 38;

S2: b = -1;

S1 ->A S2, meaning that S1 has an anti-dependence on S2 because S1reads from the variable b before S2 writes to it.

Output-dependence

S0: int a, b = 40;

S1: a = b - 38;

S2: a = 2;

S1 ->O S2, meaning that S1 has an output dependence on S2 because both write to the variable a.

Control dependency

```
if(a == b)
then
{
  c = “controlled”;
}
d=“not
controlled”;
```

```
if(a == b)
then
{
 }
c = “controlled”;
d=“not
controlled”;
```

```
if(a == b)
then
{
  c = “controlled”;
  d=“not
controlled”;
}
```

Loop dependency

- Loop dependency has two types:
  - Loop – carried dependency
  - Loop – independent dependency

Loop – Carried dependency

- In Loop – carried dependency, statements in an iteration of a loop depend on statements in other iteration of the loop

```
for(i=0;i<4;i++)
{
S1: b[i]=8;
S2: a[i]=b[i-1] + 10;
}
```

Loop – Independent dependency

- In Loop – independent dependency, loops have inter-iteration dependence, but do not have dependence between iterations.
- Each iteration may be treated as a block and performed in parallel without other synchronization efforts.

```
for (i=0;i<4;i++)
{
S1: b[i] = 8;
S2: a[i] =b[i] + 10;
}
```

Solution

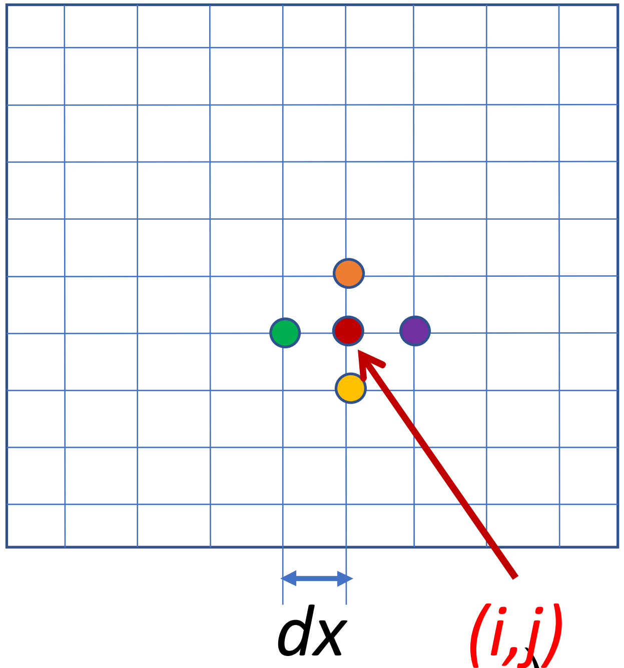
- Parallel computing in a shared memory system: OpenMP, CUDA
  - Synchronization
- Parallel computing in a distributed memory system: MPI, Cloud, Grid
  - Synchronization
  - Communication

Data dependency: Example

- Heat Diffusion Equations:

$$\frac{\partial C}{\partial t} = D\nabla^2 C$$

$$\nabla^2 C = \frac{\partial^2 C}{\partial x^2} + \frac{\partial^2 C}{\partial y^2}$$



- Solving approach
  - Initialize inputs: $C^0_{i,j}$
  - At step n+1:

$$\nabla^2 C^{tn}_{\boxed{i,j}} = FD^{tn}_{\boxed{i,j}} = \frac{\left(C^{tn}_{\boxed{i+1,j}} + C^{tn}_{\boxed{i-1,j}} + C^{tn}_{\boxed{i,j+1}} + C^{tn}_{\boxed{i,j-1}} - 4C^{tn}_{\boxed{i,j}}\right)}{dx^2}$$

$$C^{tn+1}_{\boxed{i,j}} = C^{tn}_{\boxed{i,j}} + dt * D * FD^{tn}_{\boxed{i,j}}$$

- Data dependency?

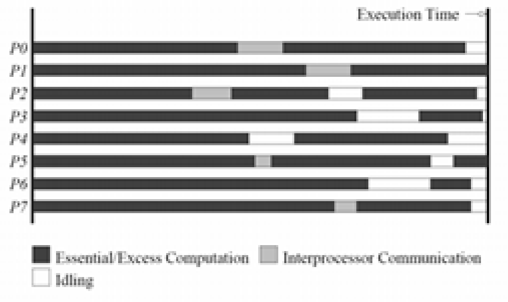
Data dependency

- Calculation at point (i,j) needs data from neighboring points: (i-1,j), (i+1,j) , (i,j-1) , (i,j+1)
- This is data dependency
- Solution
  - Shared memory system: Synchronization
  - Distributed memory system: Communication and Synchronization (Difficult, Optimization)
- Exercise:
  - Write a OpenMP program to implement Heat Equations problem
  - Write a MPI program to implement Heat Equations problem

7.3 Performance Analysis

Sources of Overhead in Parallel Programs

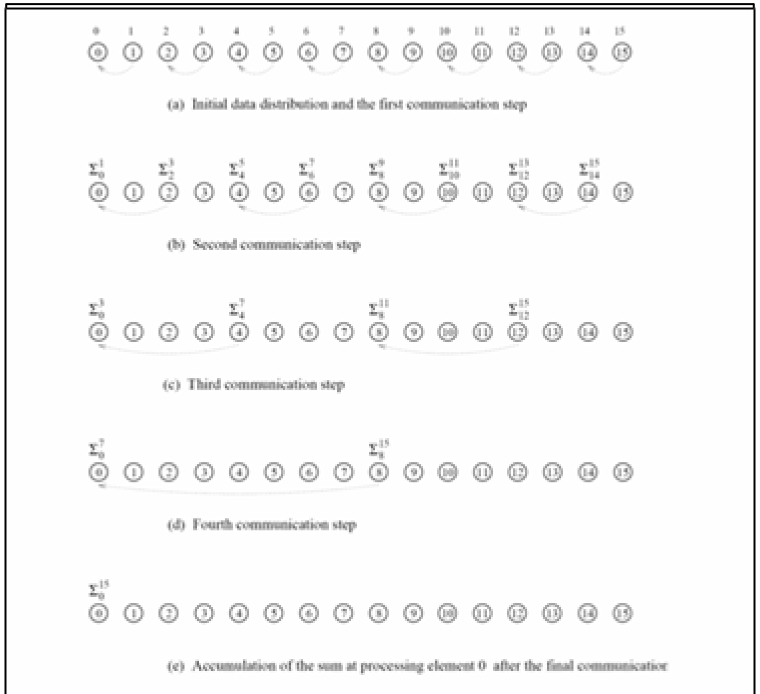
- The total time spent by a parallel system is usually higher than that spent by a serial system to solve the same problem.
  - Overheads!
    - Interprocessor Communication & Interactions
    - Idling
      - Load imbalance, Synchronization, Serial components
    - Excess Computation
      - Sub-optimal serial algorithm
      - More aggregate computations
- Goal is to minimize these overheads!



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Performance Metrics

- Parallel Execution Time
  - Time spent to solve a problem on *p* processors.
    - T_p
- Total Overhead Function
  - $T_o = pT_p - T_s$
- Speedup
  - $S = T_s/T_p$
  - Can we have superlinear speedup?
    - exploratory computations, hardware features
- Efficiency
  - $E = S/p$
- Cost
  - $p\ T_p$ (processor-time product)
  - Cost-optimal formulation
- Working example: Adding *n* elements on *n* processors.

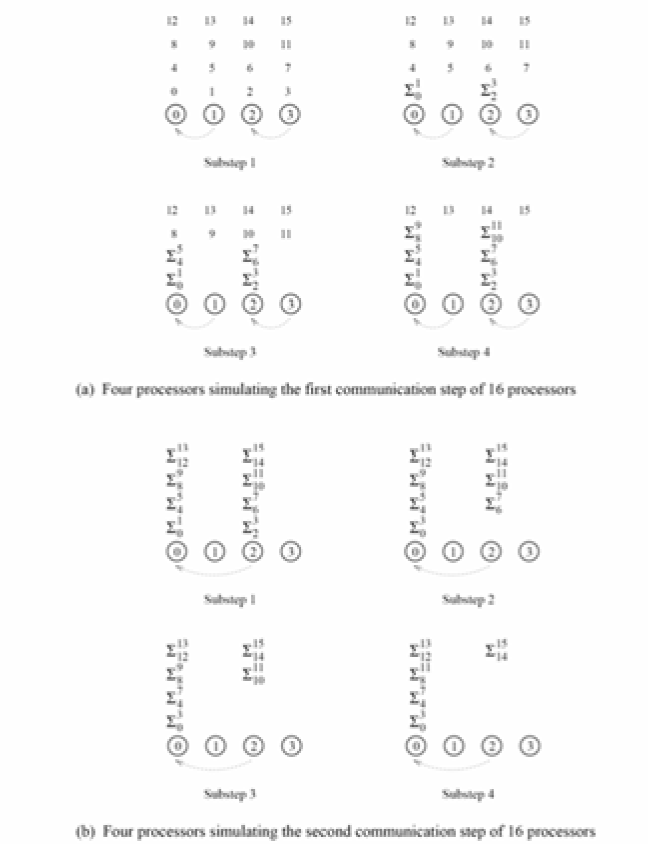


Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

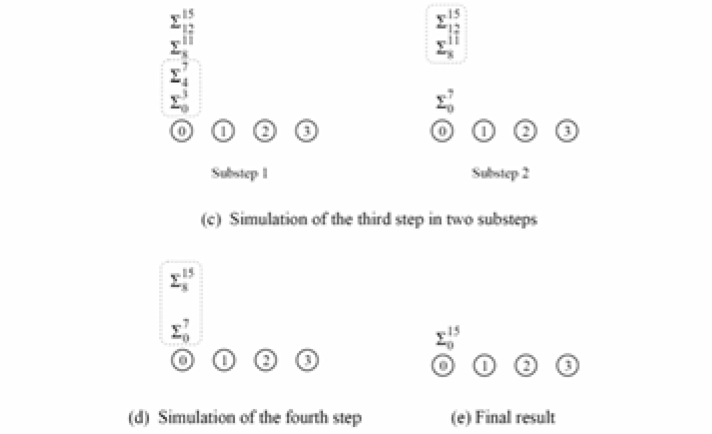
Effect of Granularity on Performance

- Scaling down the number of processors
- Achieving cost optimality
- Naïve emulations vs Intelligent scaling down
  - adding *n* elements on *p* processors

Scaling Down by Emulation

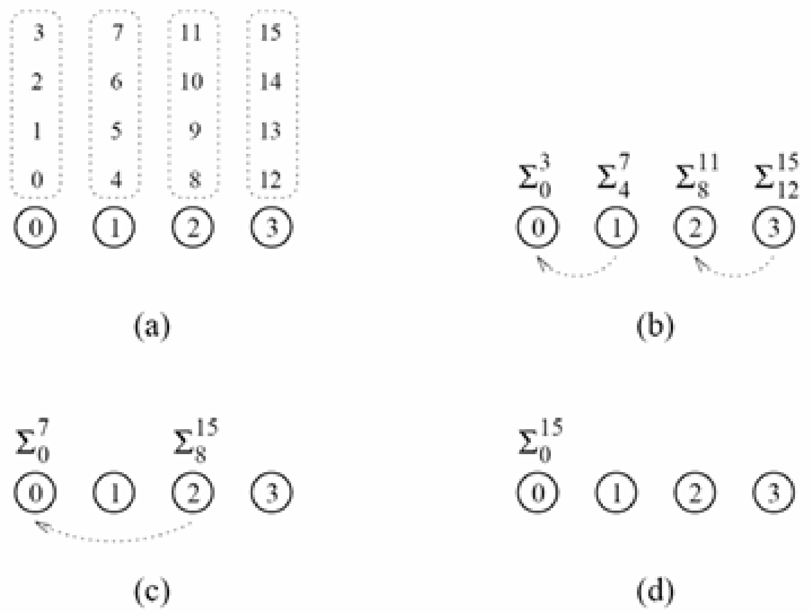


Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j.



Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).

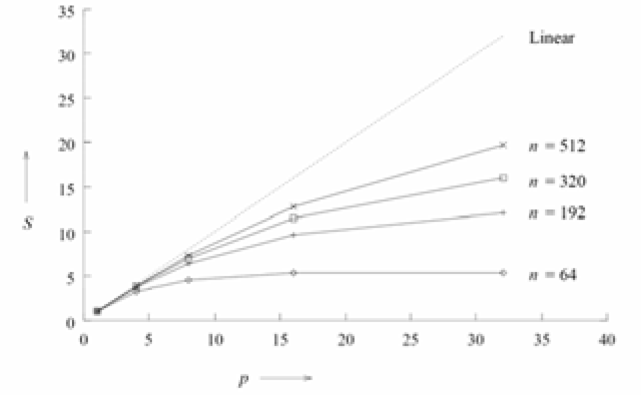
Intelligent Scaling Down



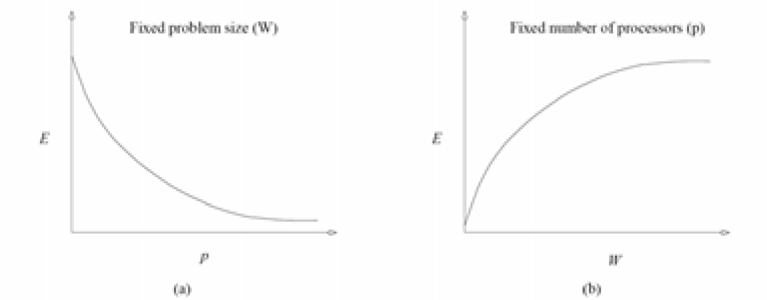
A cost-optimal way of computing the sum of 16 numbers using four processing elements.

Scalability of a Parallel System

- The need to predict the performance of a parallel algorithm as *p* increases
- Characteristics of the T_o function
  - Linear on the number of processors
    - serial components
  - Dependence on T_s
    - usually sub-linear
- Efficiency drops as we increase the number of processors and keep the size of the problem fixed
- Efficiency increases as we increase the size of the problem and keep the number of processors fixed



Speedup versus the number of processing elements for adding a list of numbers.



Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

Scalable Formulations

- A parallel formulation is called *scalable* if we can maintain the efficiency constant when increasing *p* by increasing the size of the problem
- Scalability and cost-optimality are related
- Which system is more scalable?

Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p=1$	$p=4$	$p=8$	$p=16$	$p=32$
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

Measuring Scalability

- What is the *problem size?*
- Isoefficiency function
  - measures the rate by which the problem size has to increase in relation to *p*
- Algorithms that require the problem size to grow at a lower rate are more scalable
- Isoefficiency and cost-optimality
- What is the best we can do in terms of isoefficiency?

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions !

soict.hust.edu.vn/ fb.com/groups/soict