Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

2:Parallel Programming

Twara Parekh

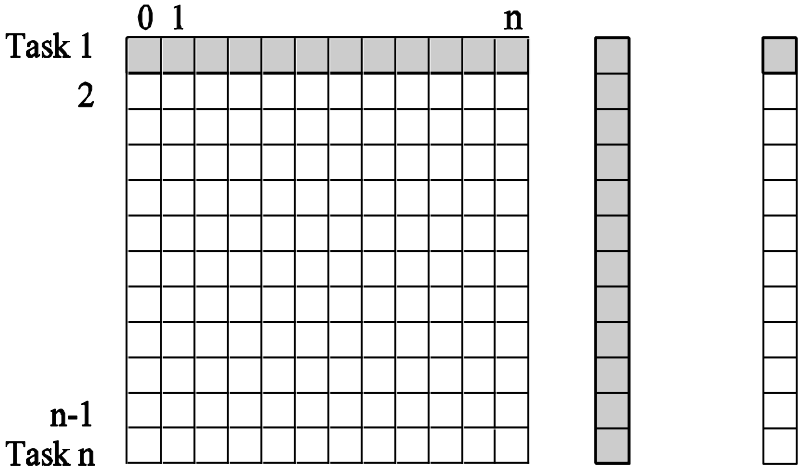
Lecturer

Computer Science & Engineering

Syllabus

- Principles of Parallel Algorithm Design: Preliminaries, Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing, Methods for Containing Interaction Overheads, Parallel Algorithm Models,
- Processor Architecture, Interconnect, Communication, Memory Organization, and Programming Models in high performance computing architecture examples: IBM CELL BE, Nvidia Tesla GPU, Intel Larrabee Micro architecture and Intel Nehalem micro architecture
- Memory hierarchy and transaction specific memory design, Thread Organization

- The first step in developing a parallel algorithm is to *decompose* the problem into tasks that can be executed concurrently
- A given problem may be decomposed into tasks in many different ways.
- Tasks may be of same, different, or even indeterminate sizes.
- A decomposition can be illustrated in the form of a directed graph.
  - Such a graph is called a *task-dependency graph*.
  - Nodes correspond to tasks and edges indicate dependencies



Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

- **Observations:**
  - Tasks share the vector ***b*** but they have no control dependencies.
    - There are zero edges in the task-dependency graph
  - All tasks are of the same size in terms of number of operations.
- Is this the maximum number of tasks we could decompose this problem into?

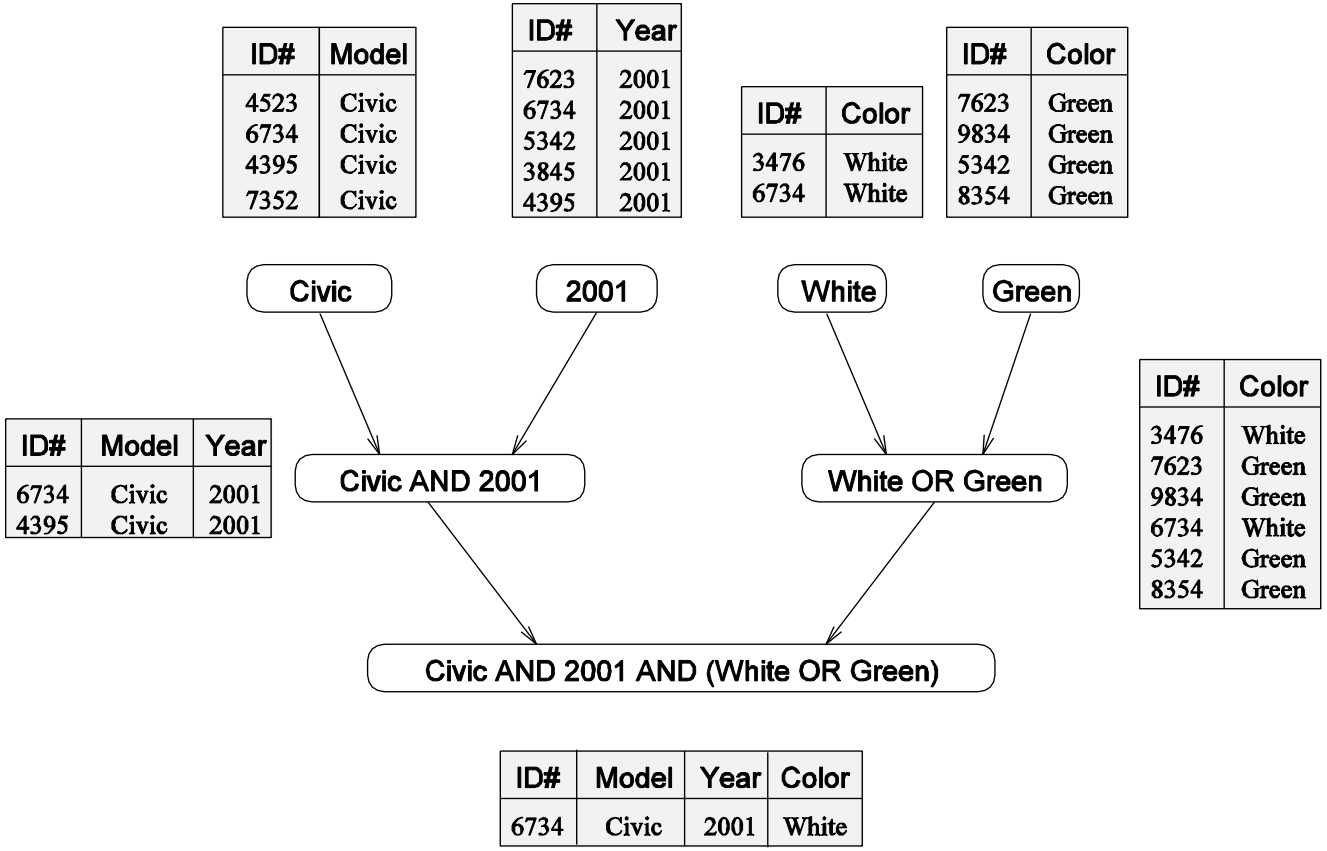
Consider the execution of the query:

```
MODEL = "CIVIC" AND YEAR = "2001" AND
   (COLOR = "GREEN" OR COLOR = "WHITE")
```

on the following database:

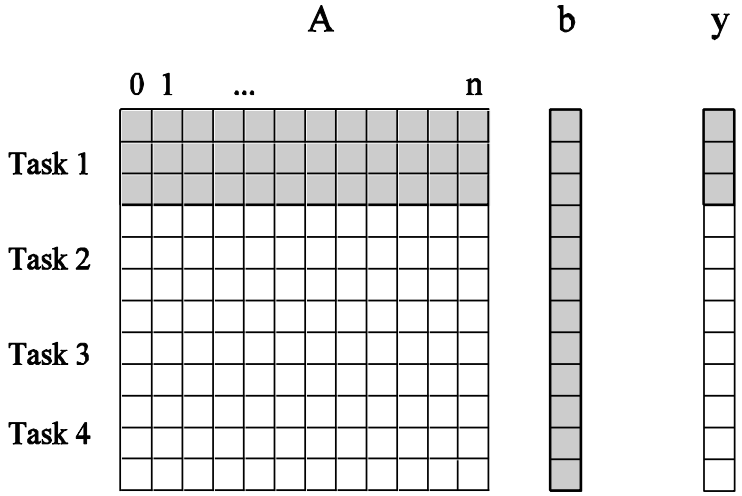
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000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

- Assume the query is divided into four subtasks
  - Each task generates an intermediate table of entries



Edges in this graph denote dependencies.

- The number of tasks into which a problem is decomposed determines its granularity.
  - *Fine granularity*: Decomposition into a large number of tasks
  - *Coarse granularity*: Decomposition into a small number of tasks



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

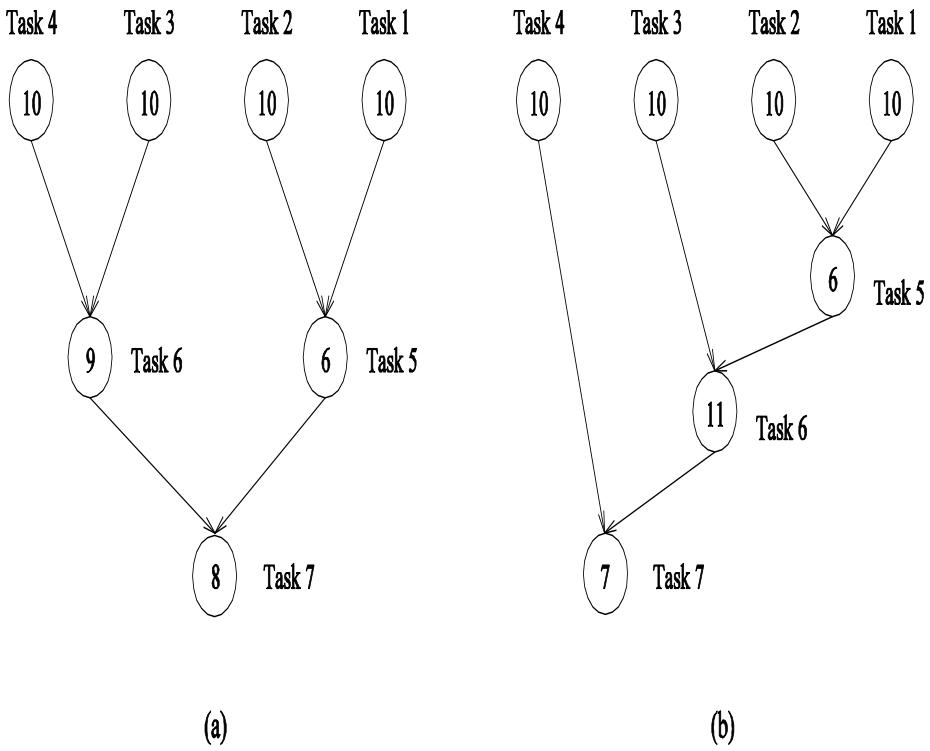
- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.

- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution.

- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program.

- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

- 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 between any pair of zero in-degree to zero out-degree node is known as the *critical path*.
- The length of the longest path in a task dependency graph is called the *critical path length*.



- Parallel time cannot be made arbitrarily small by making the decomposition finer in granularity.
  - A parallel algorithm will inherently have a limited number of decomposable tasks
  - For example, in the case of multiplying a dense matrix with a vector, there can be no more than **(n^2)** concurrent tasks.

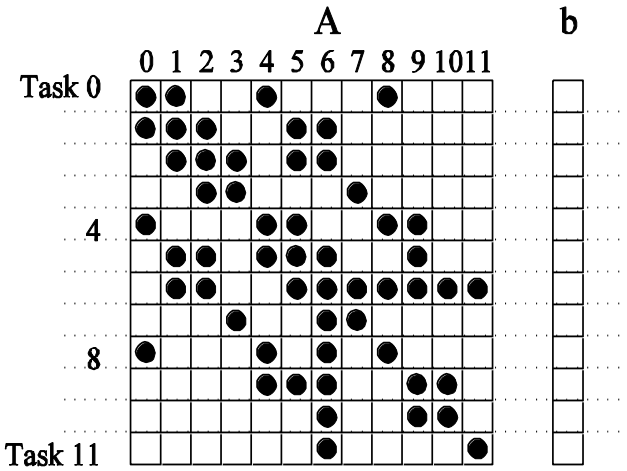
- Tasks interaction is another limiting factor on parallel performance

- *Task interaction graph*: an undirected graph that captures the pattern of interactions among tasks

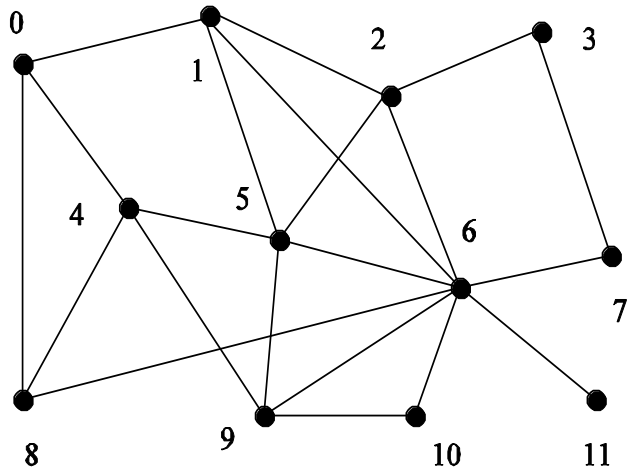
- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.

- The edge-set of a task-interaction graph is a superset of the edge-set of a task-dependency graph. Why?

- Consider the problem of multiplying a sparse matrix ***A*** with a vector ***b***. The following observations can be made:
  - **Notes**
    - Decomposition is as before; each `y[i]` computation is a task.
    - Only non-zero elements of matrix ***A*** participate in the computation, in this case.
    - We also partition ***b*** across tasks; `b[i]` is held by Task `i`.



(a)



(b)

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.
  - Thus, a parallel algorithm must also provide a mapping of tasks to processes.

- **Note:** mapping is from tasks to processes, as opposed to processors.
  - Because typical programming APIs do not allow easy binding of tasks to physical processors.
  - We aggregate tasks into processes and rely on the system to map these processes to physical processors.

- Processes (no in UNIX sense): logical computing agents that perform tasks
  - Task + task data + task code required to produce the task's output

- Processors: physical hardware units that perform tasks

- An appropriate mapping must minimize parallel execution time by:

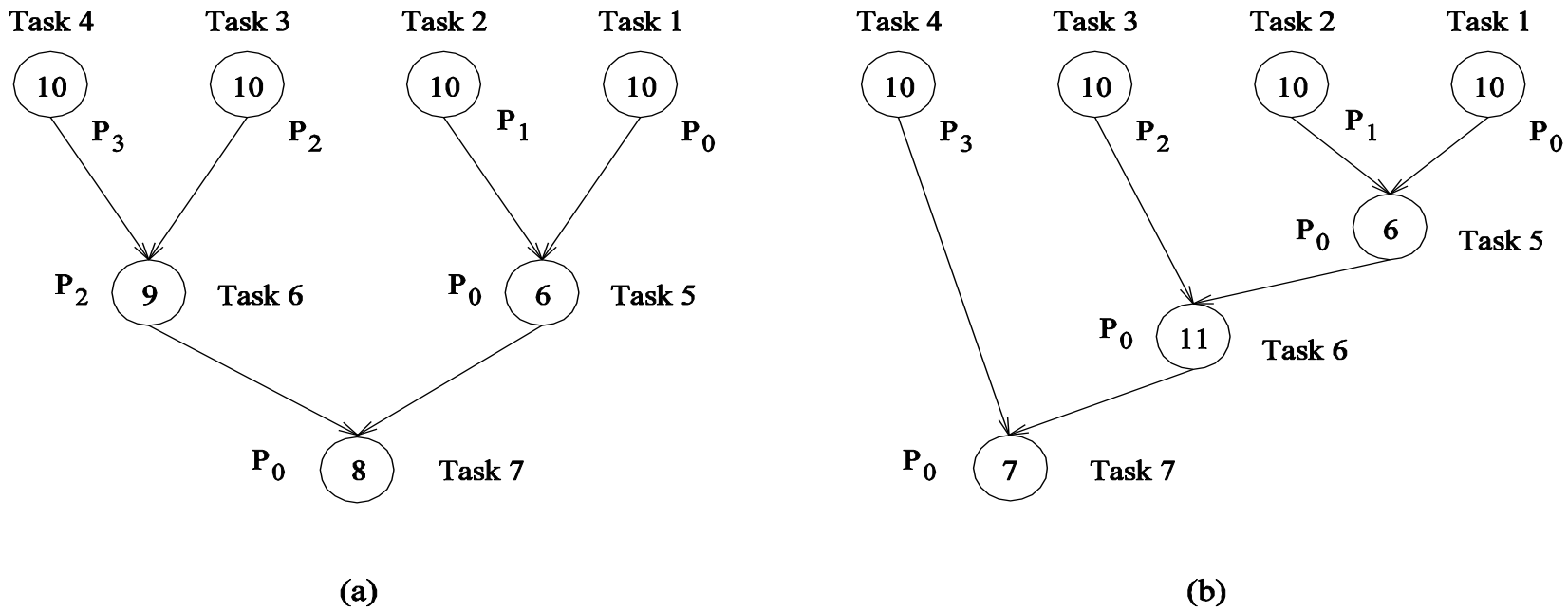
  1. Mapping independent tasks to different processes.

  2. Assigning tasks on critical path to processes as soon as they become available.

  3. Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

- These criteria often conflict with each other.
  - E.g., a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

- Can you think of other such conflicting cases?



Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

- 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
- So how does one decompose a task into various subtasks?
- There is no single recipe that works for all problems!
- Commonly used techniques that apply to broad classes of problems:
  - Recursive decomposition
  - Data decomposition
  - Exploratory decomposition
  - Speculative decomposition
  - Hybrid decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.

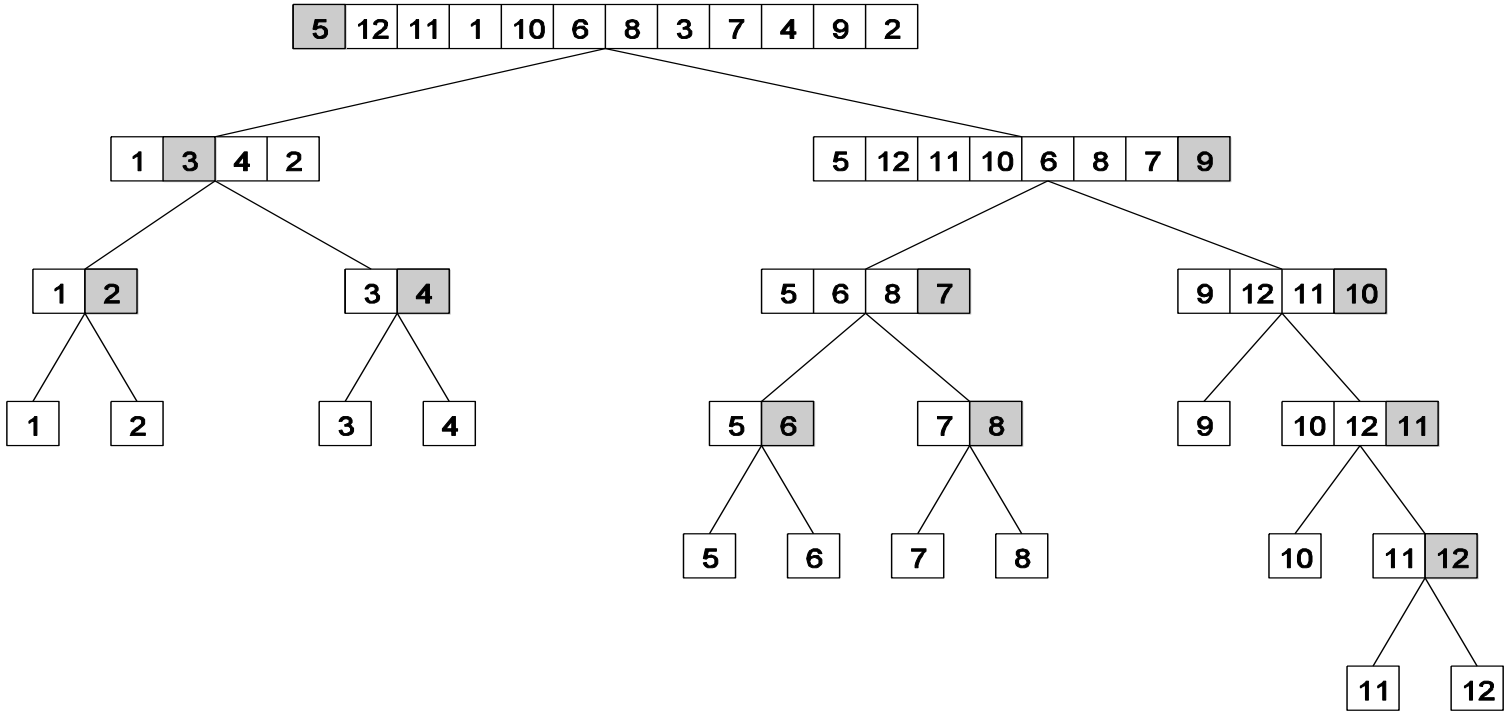
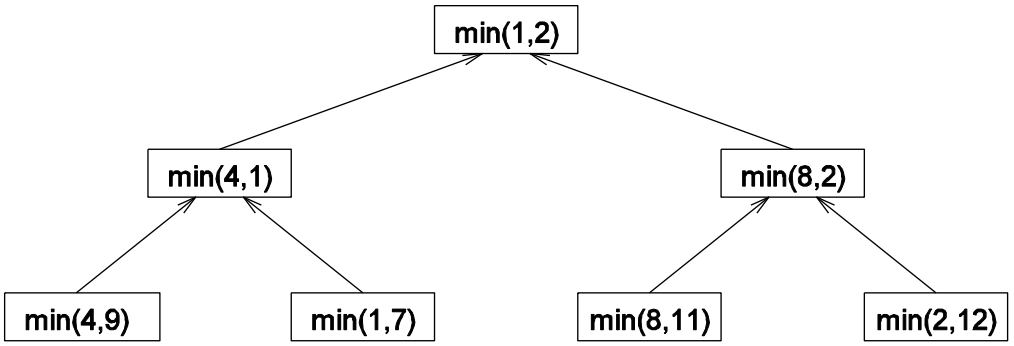


Figure 3.8 The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.

```
1. procedure SERIAL_MIN(A,n)
2. begin
3. min =A[0];
4. for i:= 1 to n - 1 do
5.         if (A[i] < min) min := A[i];
6. endfor;
7. return min;
8. end SERIAL_MIN
```

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.     min := A [0]  ;
5. else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.     if (lmin  < rmin) then
9.                 min := lmin;
10.    else
11.                min := rmin;
12.    endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

The code in the previous foil can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of finding the minimum number in the set {4, 9, 1, 7, 8, 11, 2, 12}. The task dependency graph associated with this computation is as follows:



- 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?
    - All these—leading to different data decomposition methods
- How do induce a computational partitioning?
  - Owner-computes rule

- Partitioning the output data
  - Applied when each element of the output data can be computed independently of the others, as a function of the input

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

(a)

$$\begin{aligned} \text{Task 1: } C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ \text{Task 2: } C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ \text{Task 3: } C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ \text{Task 4: } C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

(b)

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

(b) Partitioning the frequencies (and itemsets) among the tasks

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 1

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	C, D	1
B, D, E, F, K, L	D, K	2
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

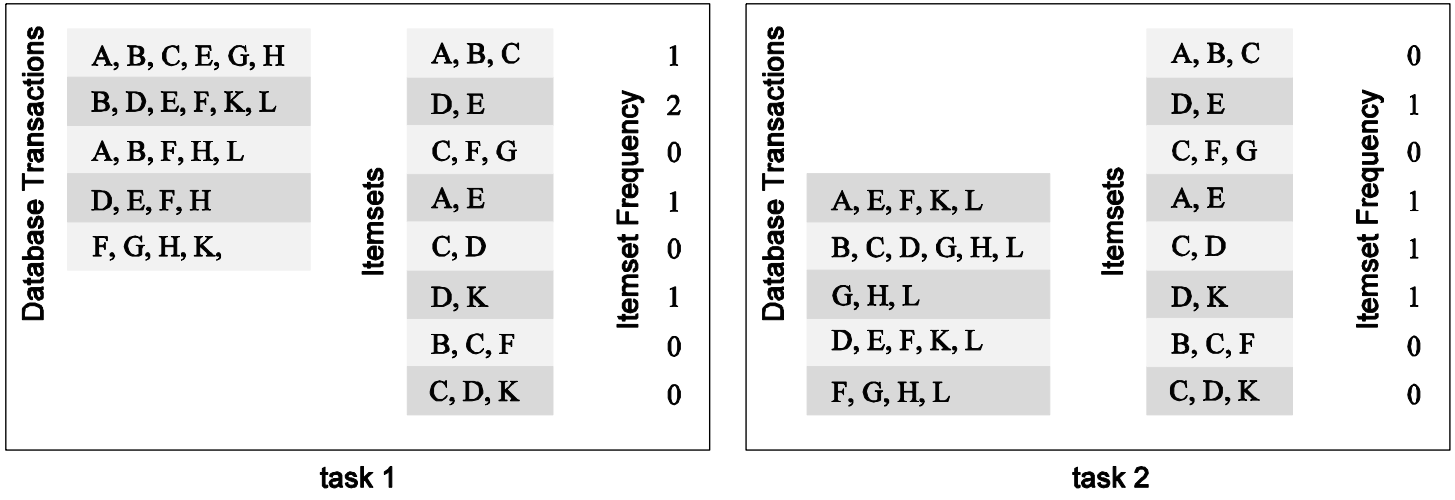
task 2

- From the previous example, the following observations can be made:
  - If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
  - If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

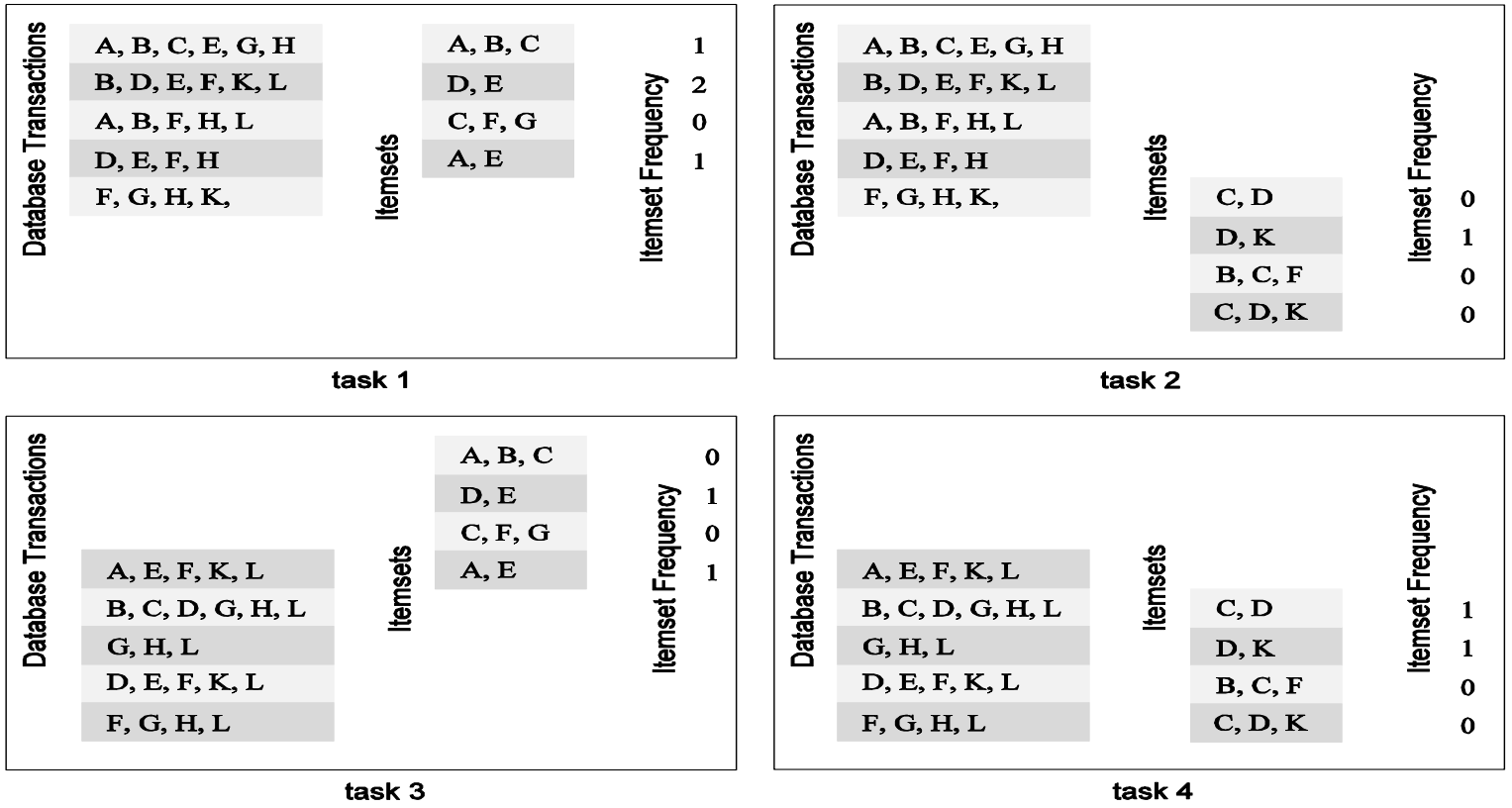
- Output data partitioning is applicable if each output can be naturally computed as a function of the input.
- In many algorithms it is not possible or desirable to partition the output data
  - e.g., the problem of finding the minimum in a list, sorting a given list, etc.
- In such cases, it is sometimes possible to partition the input data, and then use this partitioning to induce concurrency
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.

- In the frequency counting example, the input (i.e., the transaction set) can be partitioned.
  - This induces a task decomposition in which each task generates partial counts for all itemsets. These are combined subsequently for aggregate counts.

Partitioning the transactions among the tasks

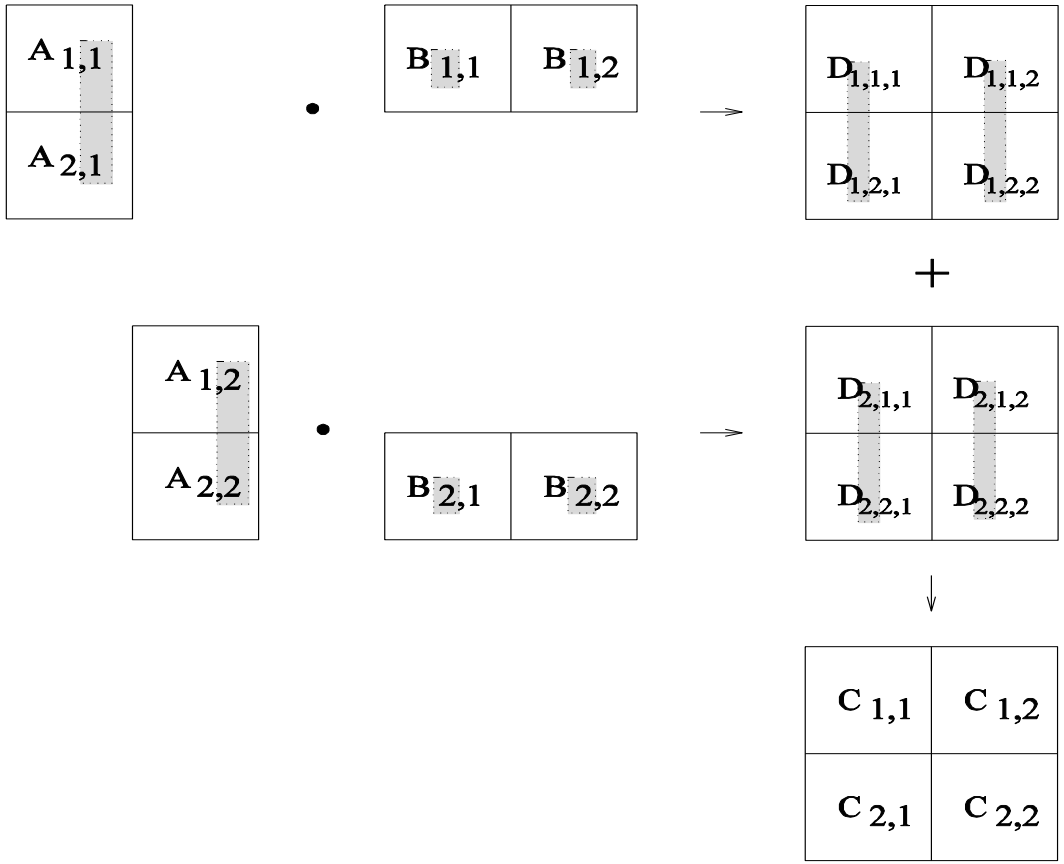


Partitioning both transactions and frequencies among the tasks



- 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.

- Revisiting the dense matrix multiplication example.



Stage I

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

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: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

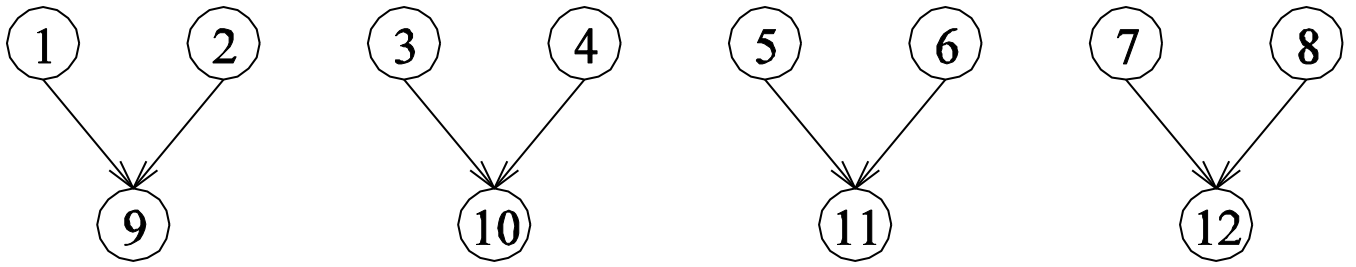
Task 09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

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

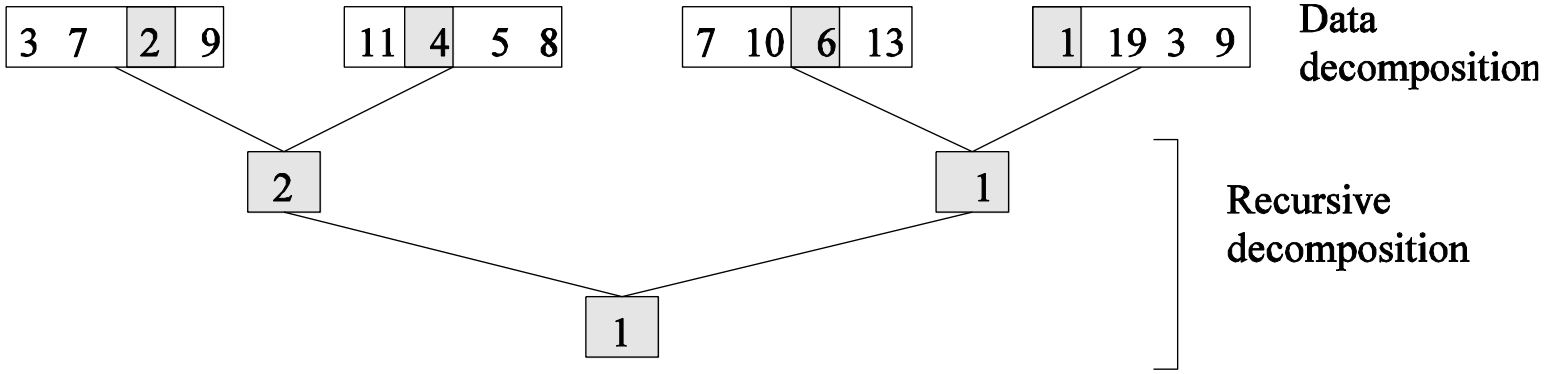
Task 11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

Task 12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

The task dependency graph for the decomposition (shown in previous foil) into 12 tasks is as follows:



- 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 alone or in conjunction with other decomposition methods
  - Hybrid decomposition

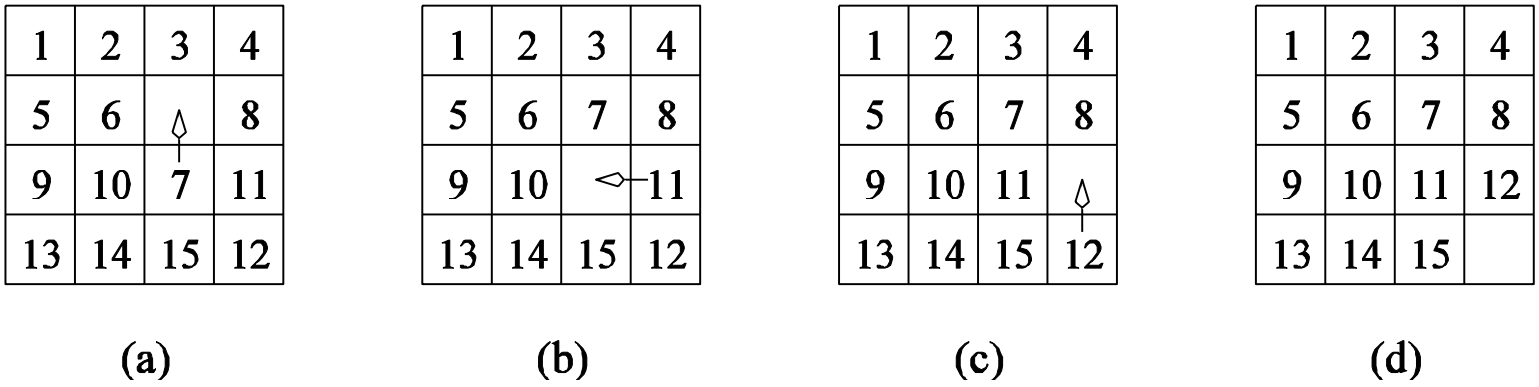


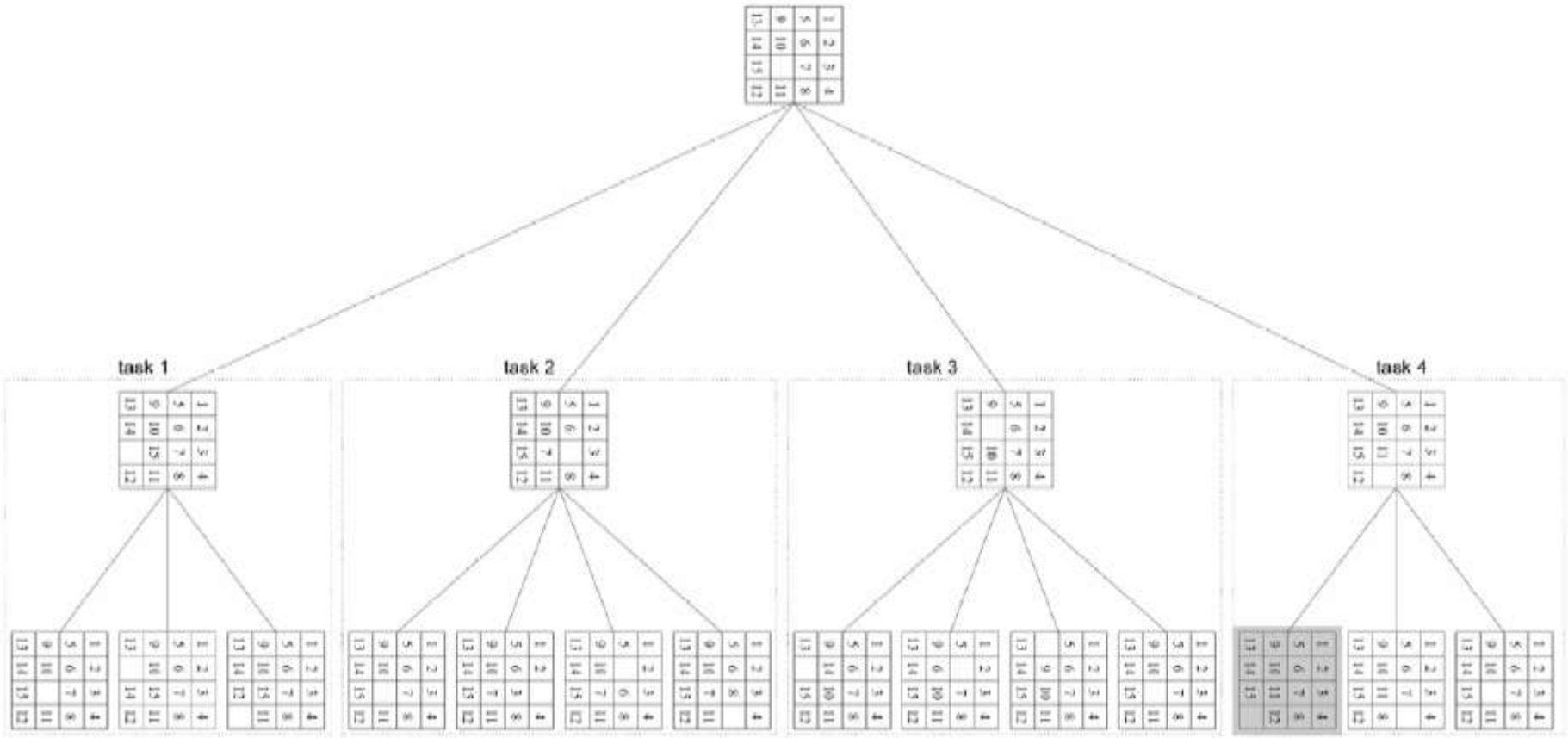
- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.

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

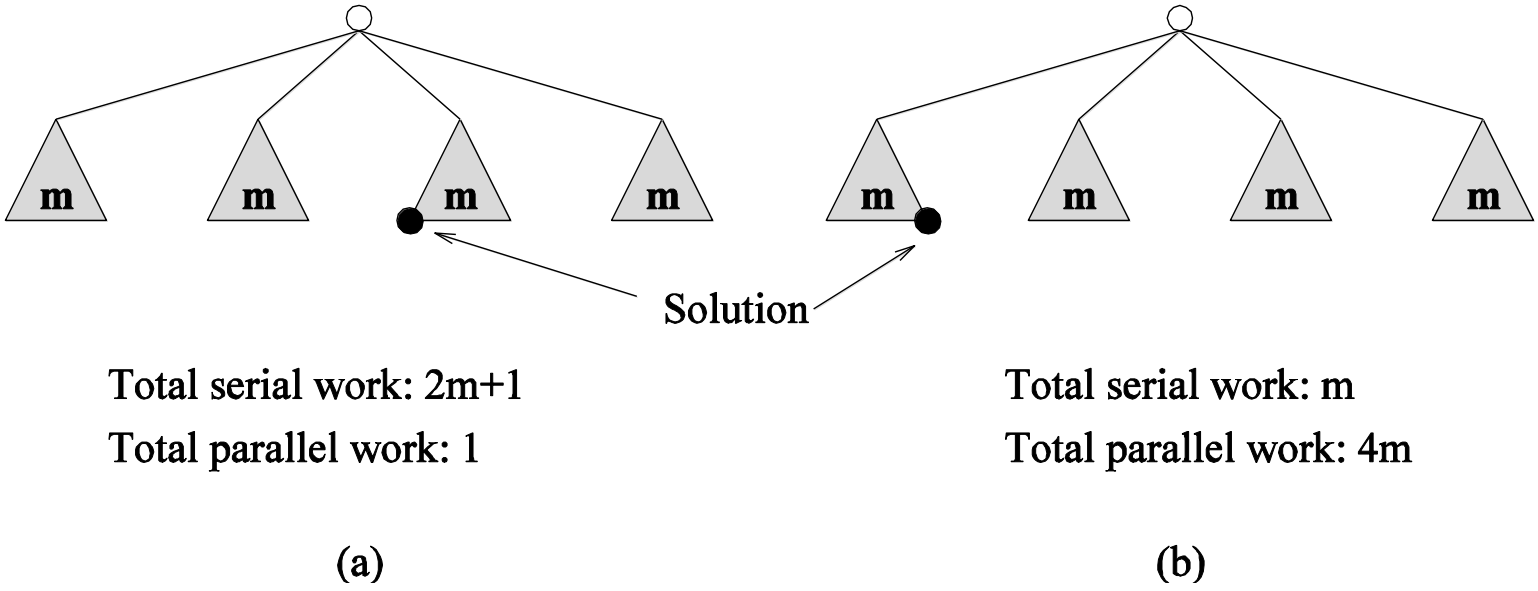
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.

- Used to decompose computations that correspond to a search of a space of solutions
  - Examples theorem proving, game playing, etc.
- Example: solution of the 15-puzzle problem
- We show a sequence of three moves that transform a given initial state (a) to desired final state (d).




task 1
task 2
task 3
task 4

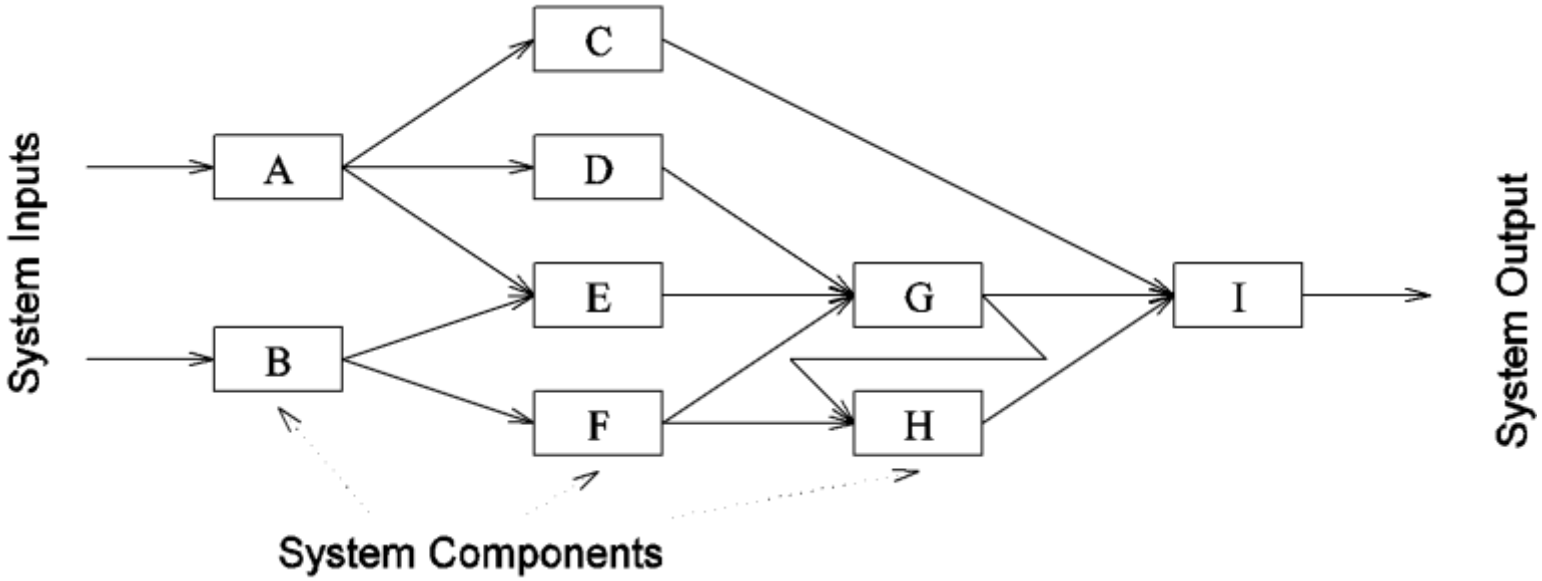
- In many instances of exploratory decomposition, the decomposition technique may change the amount of work done by the parallel formulation.
- This change results in super- or sub-linear speedups.



- 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

- Performs more or the same aggregate work (but not less) than the sequential algorithm

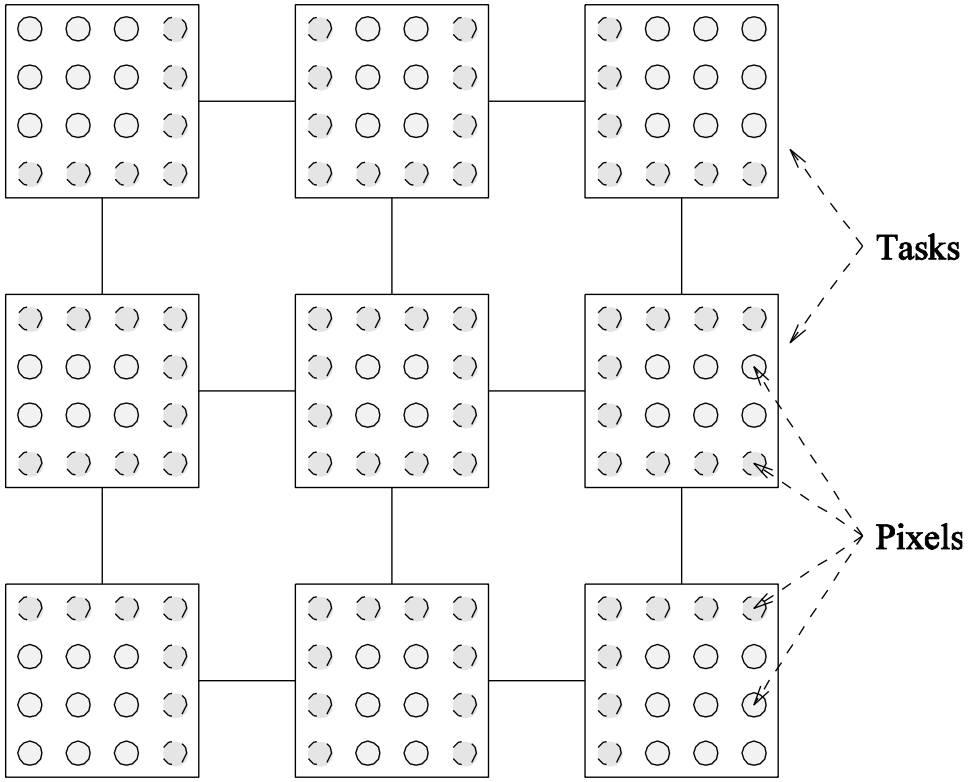

System Inputs
A
B
C
D
E
F
G
H
I
System Output
System Components

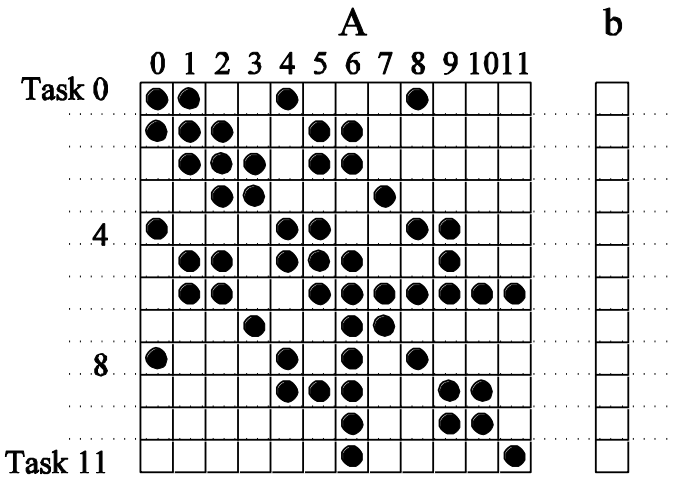
- If predictions are wrong…
  - Work is wasted
  - work may need to be *undone*
    - state-restoring overhead
      - memory/computations

- However, it may be the only way to extract concurrency!

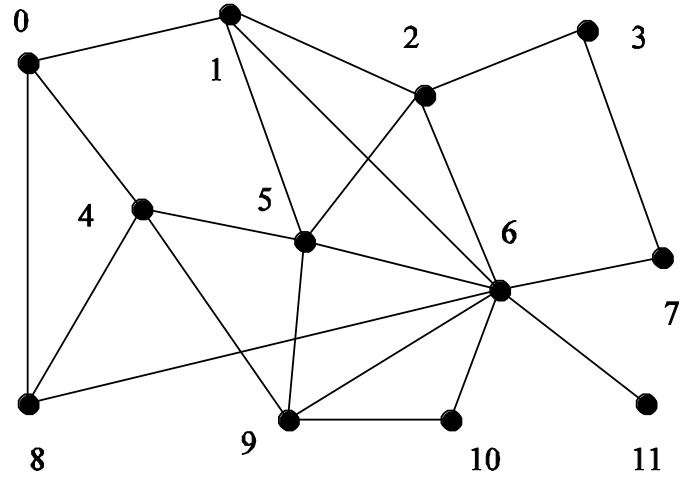
  - Are the tasks available *a priori*?
    - Static vs dynamic task generation
  - How about their computational requirements?
    - Are they uniform or non-uniform?
  - Do we know them a priori?
  - How much data is associated with each task?
- Characteristics of inter-task interactions
  - Are they static or dynamic?
  - Are they regular or irregular?
  - Are they read-only or read-write?

- **dithering** - the process of representing intermediate colors by patterns of tiny colored dots that simulate the desired color





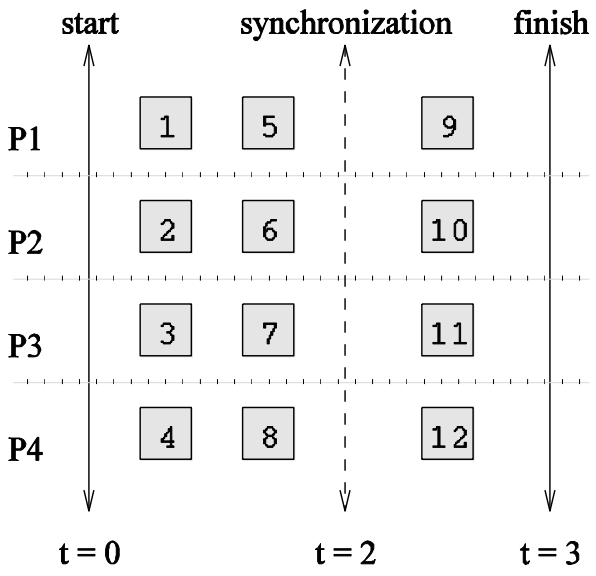
(a)



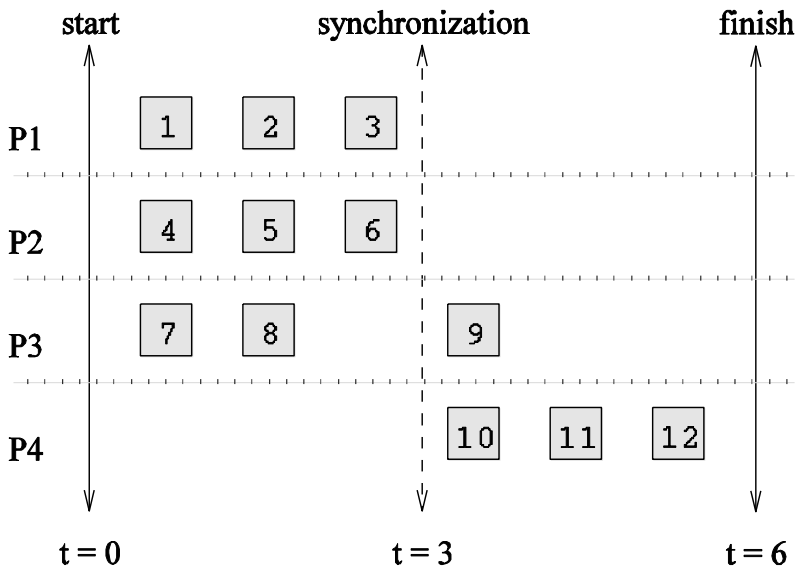
(b)

Mapping: assigning tasks to processes with the objective that all tasks should complete in the shortest amount of elapsed time

- Thus, a mapping scheme must address the major sources of overhead
  - Load imbalance
  - Inter-process communication
    - Synchronization/data-sharing
- Minimizing these overheads often represents contradicting objectives.
  - E.g., assign all interdependent tasks to the same process
- **Note**: Assigning a balanced aggregate load to each process is a necessary but not sufficient condition for reducing idling.


start
synchronization
finish
P1
P2
P3
P4
1
5
9
2
6
10
3
7
11
4
8
12
t = 0
t = 2
t = 3

(a)


start
synchronization
finish
P1
P2
P3
P4
1
2
3
4
5
6
7
8
9
10
11
12
t = 0
t = 3
t = 6

(b)

executing algorithm).

  - Applicable for tasks that are
    - generated statically
    - known and/or have uniform computational requirements

- **Dynamic Mapping**: Tasks are mapped to processes at runtime.
  - Applicable for tasks that are
    - generated dynamically
    - unknown and/or have non-uniform computational requirements

  - use data decomposition
  - their underlying input/output/intermediate data are in the form of arrays

- Block Distribution
  - Used to load-balance a variety of parallel computations that operate on multi-dimensional arrays

- Cyclic Distribution

- Block-Cyclic Distribution

- Randomized Block Distributions

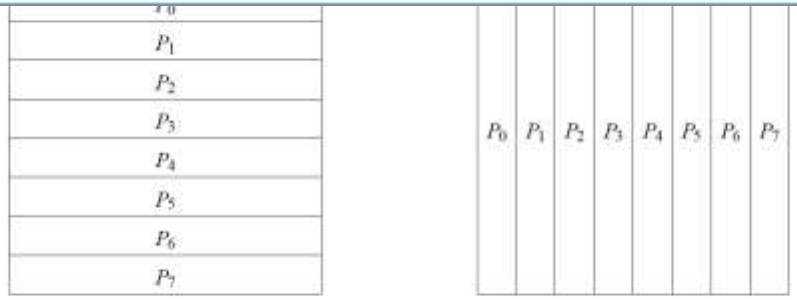


Figure 3.24 Examples of one-dimensional partitioning of an array among eight processes.

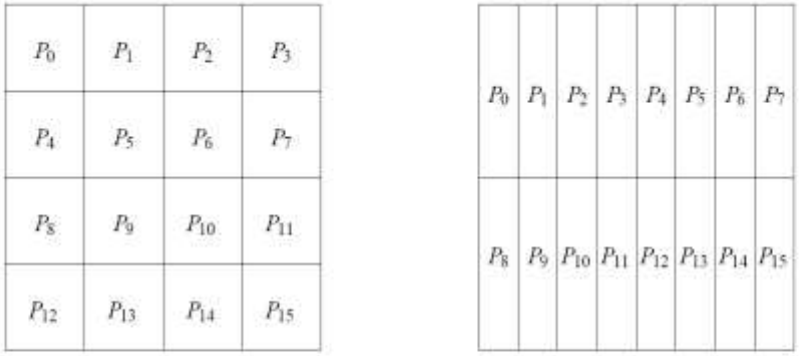


Figure 3.25 Examples of two-dimensional distributions of an array, (a) on a 4×4 process grid, and (b) on a 2×8 process grid.

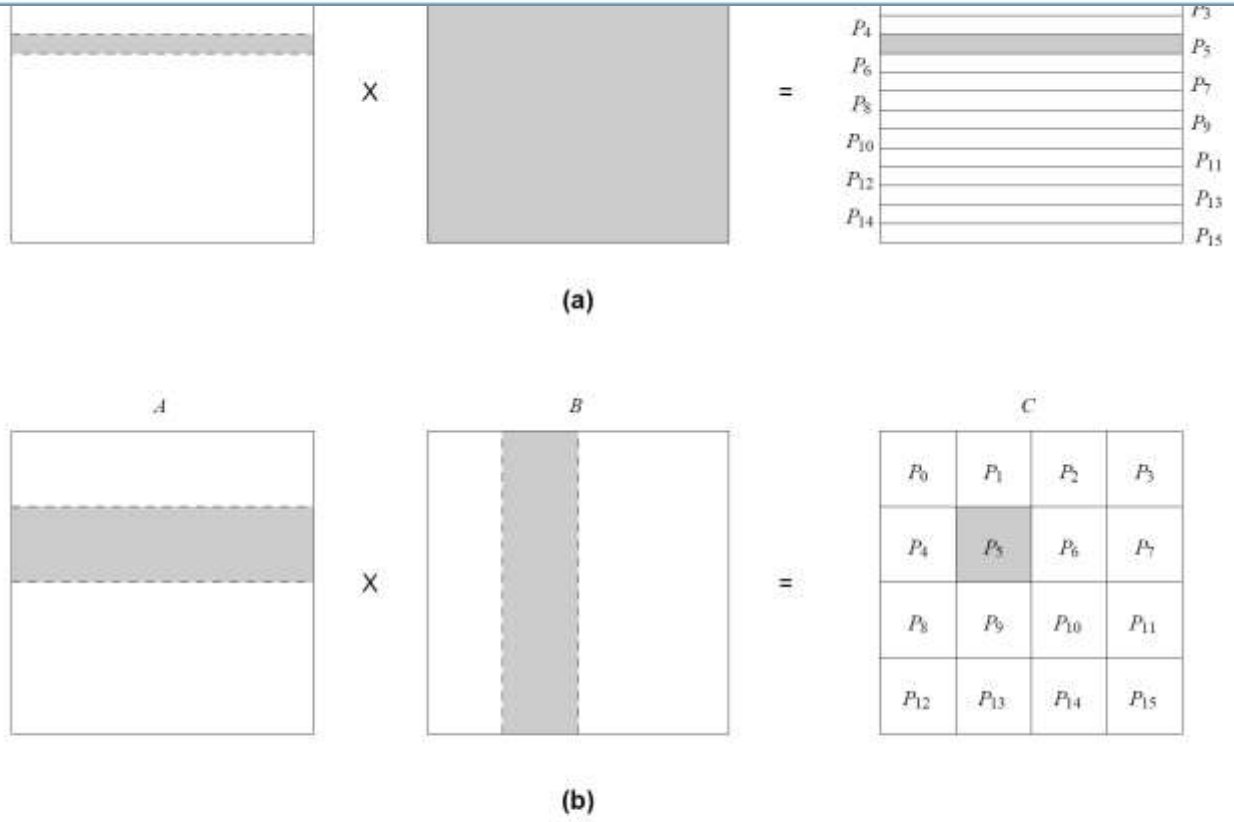


Figure 3.26 Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

  - One-dimensional?
  - Multi-dimensions?

- Which distribution reduces the amount of interaction among processes?
  - One-dimensional?
    - How many data elements are accessed in part (a) on the previous slide?
  - Multi-dimensions?
    - How many data elements are accessed in part (b) on the previous slide?

block decomposition may lead to significant load imbalances.

- A simple example of this is in LU decomposition (or Gaussian Elimination) of dense matrices.
- Block cyclic distribution schemes can be used to alleviate the load-imbalance and idling problems.
- Block cyclic distributions:
  - Partition an array into many more blocks than the number of available processes.
  - Blocks are assigned to processes in a round-robin manner so that each process gets several non-adjacent blocks.

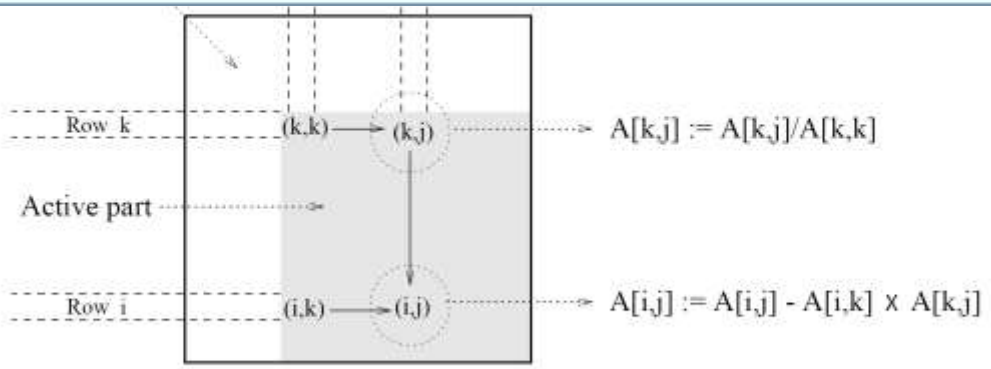


Figure 3.28 A typical computation in Gaussian elimination and the active part of the coefficient matrix during the *k*th iteration of the outer loop.

P_0 T_1	**P_3** T_4	**P_6** T_5
P_1 T_2	**P_4** T_6 T_{10}	**P_7** T_8 T_{12}
P_2 T_3	**P_5** T_7 T_{11}	**P_8** T_9 T_{13} T_{14}

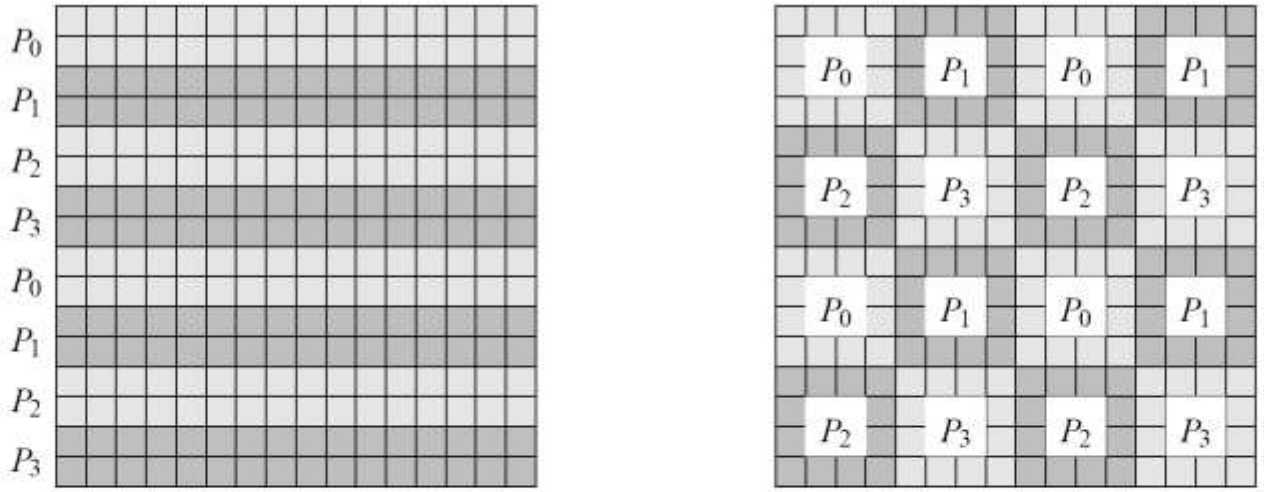
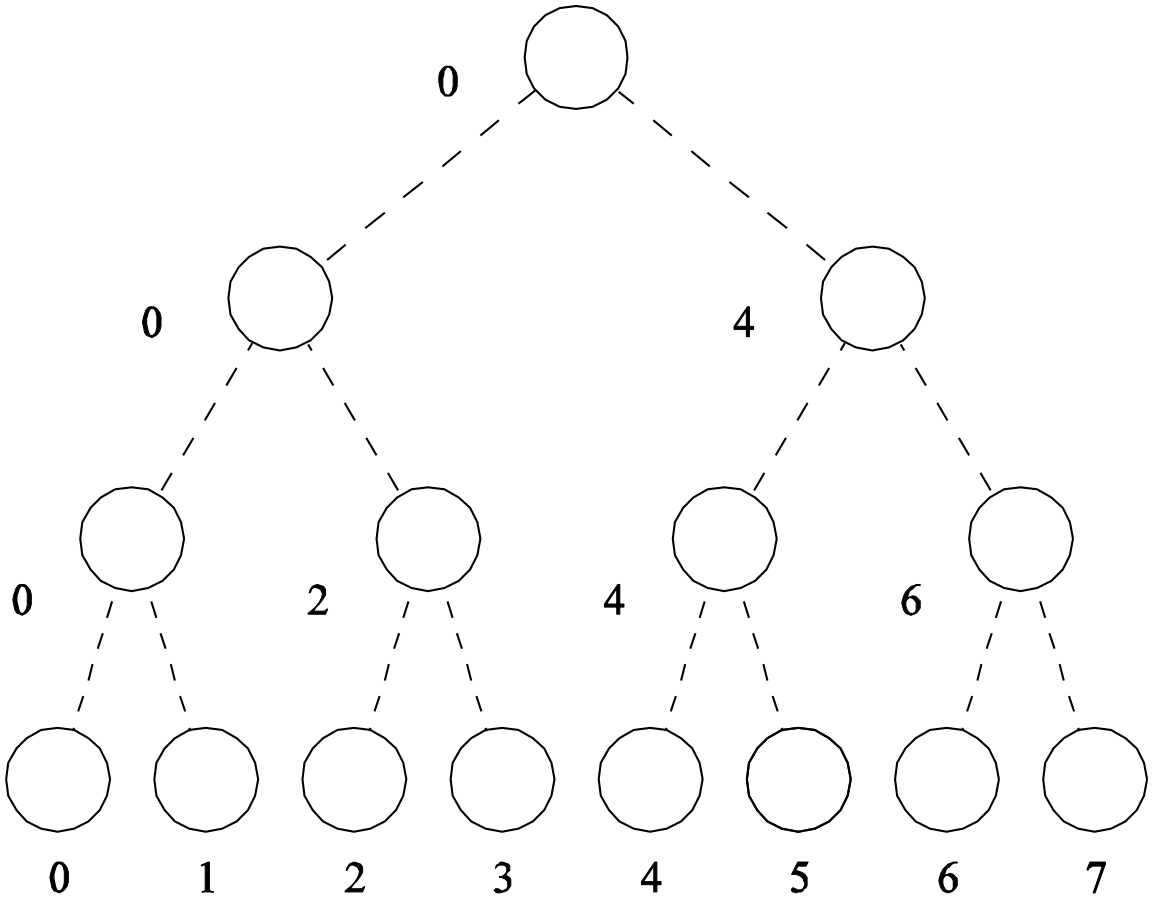


Figure 3.30 Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size 4×4, and it is mapped onto a 2×2 grid of processes in a wraparound fashion.

- Determining an optimal mapping for a general task-dependency graph is an NP-complete problem.

- Excellent heuristics exist for structured graphs.

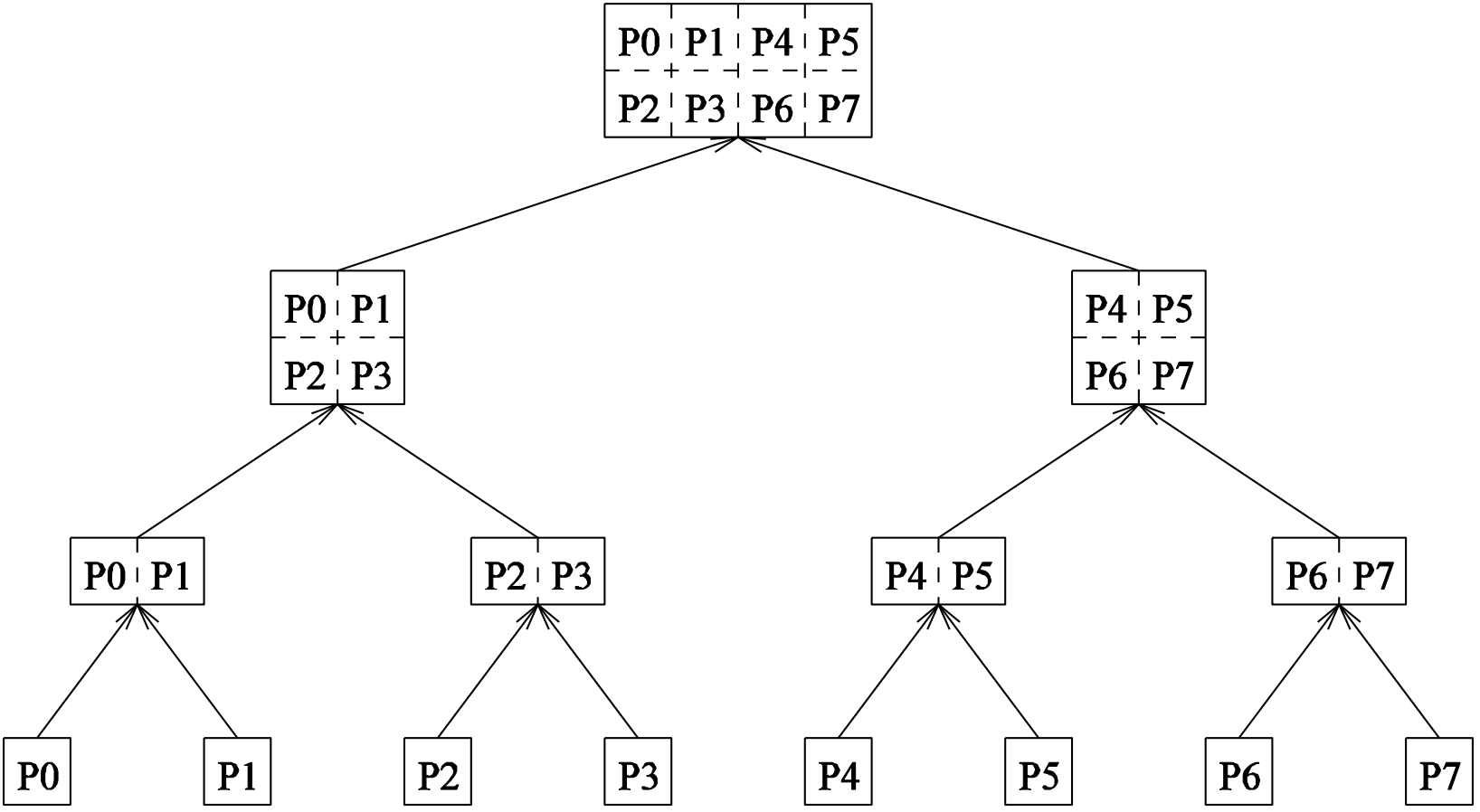
– Occurs in practical problems with recursive decomposition



- For example, the task mapping of the binary tree (quicksort) cannot use a large number of processes.

- For this reason, task mapping can be used at the top level and data partitioning within each level.

the lower level.



balancing, since load balancing is the primary motivation for dynamic mapping.

- Dynamic mapping schemes can be *centralized* or *distributed*.

- When a process runs out of work, it requests the master for more work.
- When the number of processes increases, the master may become the bottleneck.
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling.
- Selecting large chunk sizes may lead to significant load imbalances as well.
- A number of schemes have been used to gradually decrease chunk size as the computation progresses.

- This alleviates the bottleneck in centralized schemes.

- There are four critical questions:
  - How are sending and receiving processes paired together,
  - Who initiates work transfer,
  - How much work is transferred, and
  - When is a transfer triggered?

Restructure computation so that data can be reused in smaller time windows.

- **Minimize volume of data exchange**: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated.

- **Minimize frequency of interactions**: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible.

- **Minimize contention and hot-spots**: Use decentralized techniques, replicate data where necessary.

communications, multithreading, and prefetching to hide latencies.

- Replicating data or computations.
- Using group communications instead of point-to-point primitives.
- Overlap interactions with other interactions.

a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- **Data Parallel Model**: Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.
  - Usually based on data decomposition followed by static mapping
  - Uniform partitioning of data followed by static mapping guarantees load balance
  - Example algorithm: dense matrix multiplication
- **Task Graph Model**: Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.
  - Typically used to solve problems where amount of data associated with a task is large relative to computation
  - Static mapping usually used to optimize data movement costs
  - Example algorithm: parallel quicksort, sparse matrix factorization

allocate it to worker processes. This allocation may be static or dynamic.

- **Pipeline / Producer-Consumer Model**: A stream of data is passed through a succession of processes, each of which perform some task on it.

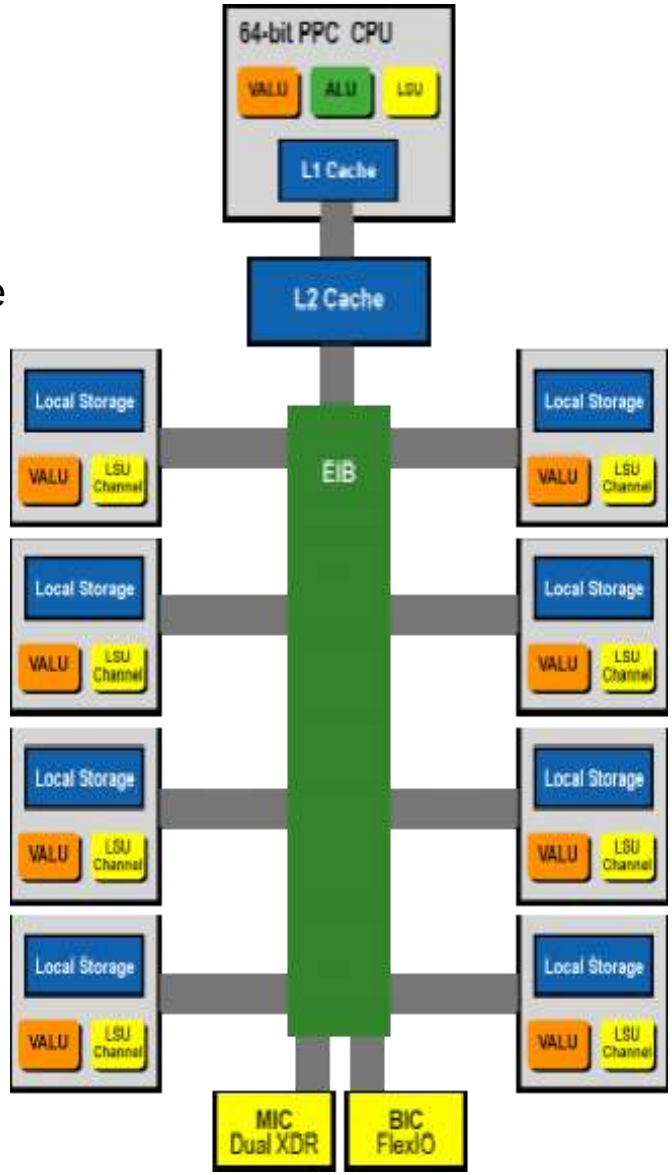
- **Hybrid Models**: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

- **The work Pool Model:**

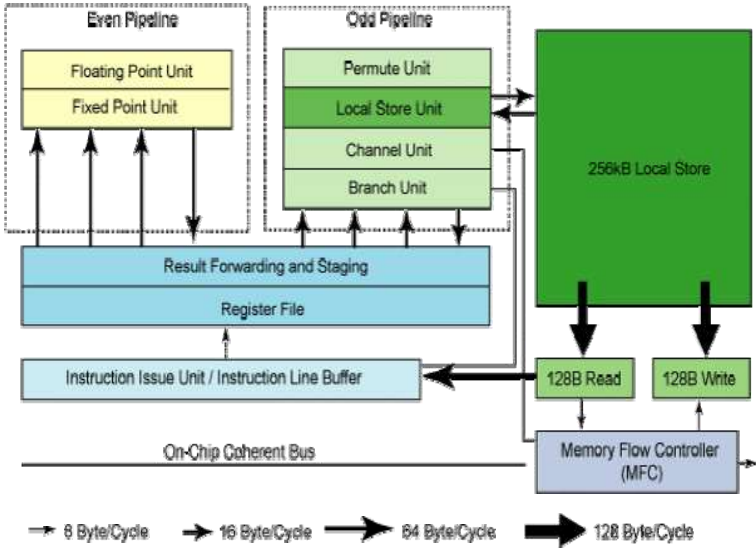
by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process. There is no desired premapping of tasks onto processes. The mapping may be centralized or decentralized. Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure. The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool. If the work is generated dynamically and a decentralized mapping is used, then a termination detection algorithm would be required so that all processes can actually detect the completion of the entire program (i.e., exhaustion of all potential tasks) and stop looking for more work.

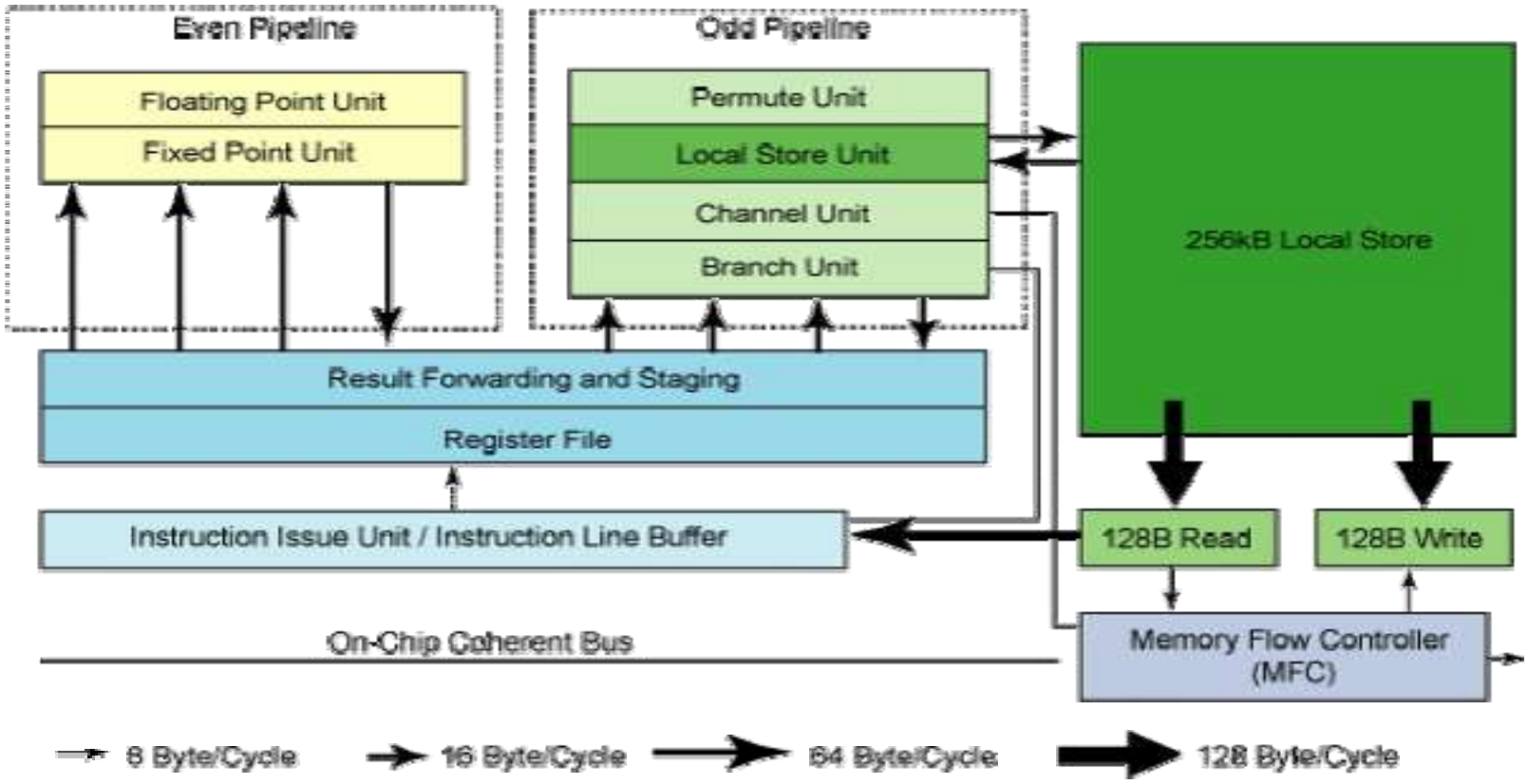
- Example : Parallelization of Loops by chunk scheduling

- Heterogeneous chip multiprocessor
  - 64-bit Power Architecture with cooperative offload processors, with direct memory access and communication synchronization methods
  - 64-bit Power Processing Element Control Core (PPE)
  - 8 Synergistic Processing Elements (SPE)
  - Single instruction, multiple-data architecture, supported by both the vector media extensions on the PPE and the instruction set of the SPE's (i.e. gaming/media and scientific applications)
  - High-bandwidth on-chip element interconnection bus (EIB),
- PPE
  - Main control unit for the entire Cell
  - 32KB L1 instruction and data caches
  - 512KB L2 unified cache
  - Dual threaded, static dual issue
  - Composed of three main units
    - Instruction Unit (IU)
      - Fetch, Decode, Branch, Issue, and Completion
    - Fixed-Point Execution Unit
      - Fixed-Point instructions and load/store instructions
    - Vector Scalar Unit
      - Vector and floating point instructions



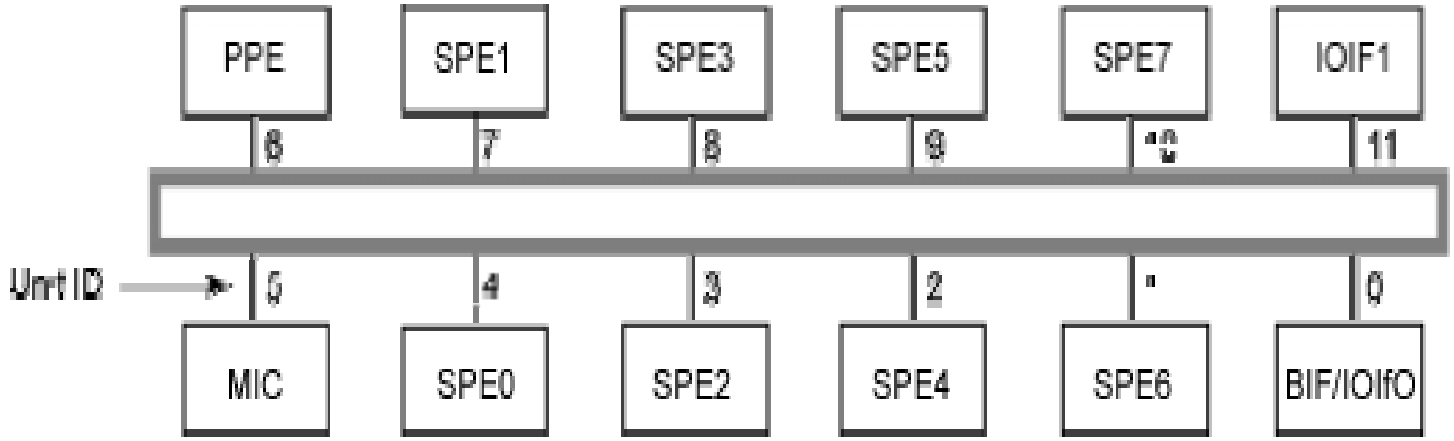
- SIMD instruction set architecture, optimized for power and performance of computational-intensive applications
- Local store memory for instructions and data
  - Additional level of memory hierarchy
  - Largest component of SPE
  - Single-port SRAM, capable of reading and writing through both narrow 128-bit and wide 128-byte ports
- Data and instructions transferred between local store and system memory by asynchronous DMA commands, controlled by memory flow controlled on SPE
  - Support for 16 simultaneous DMA commands
- Programmable DMA options
  - SPE instructions insert DMA commands into queues
  - Pool DMA commands into a single “DMA list” command
  - Insert commands in DMA queue from other SPE processors
- 128 entry unified register file for improved memory bandwidth, and optimum power efficiency and performance
- Dynamically configurable to provide support for content protection and privacy

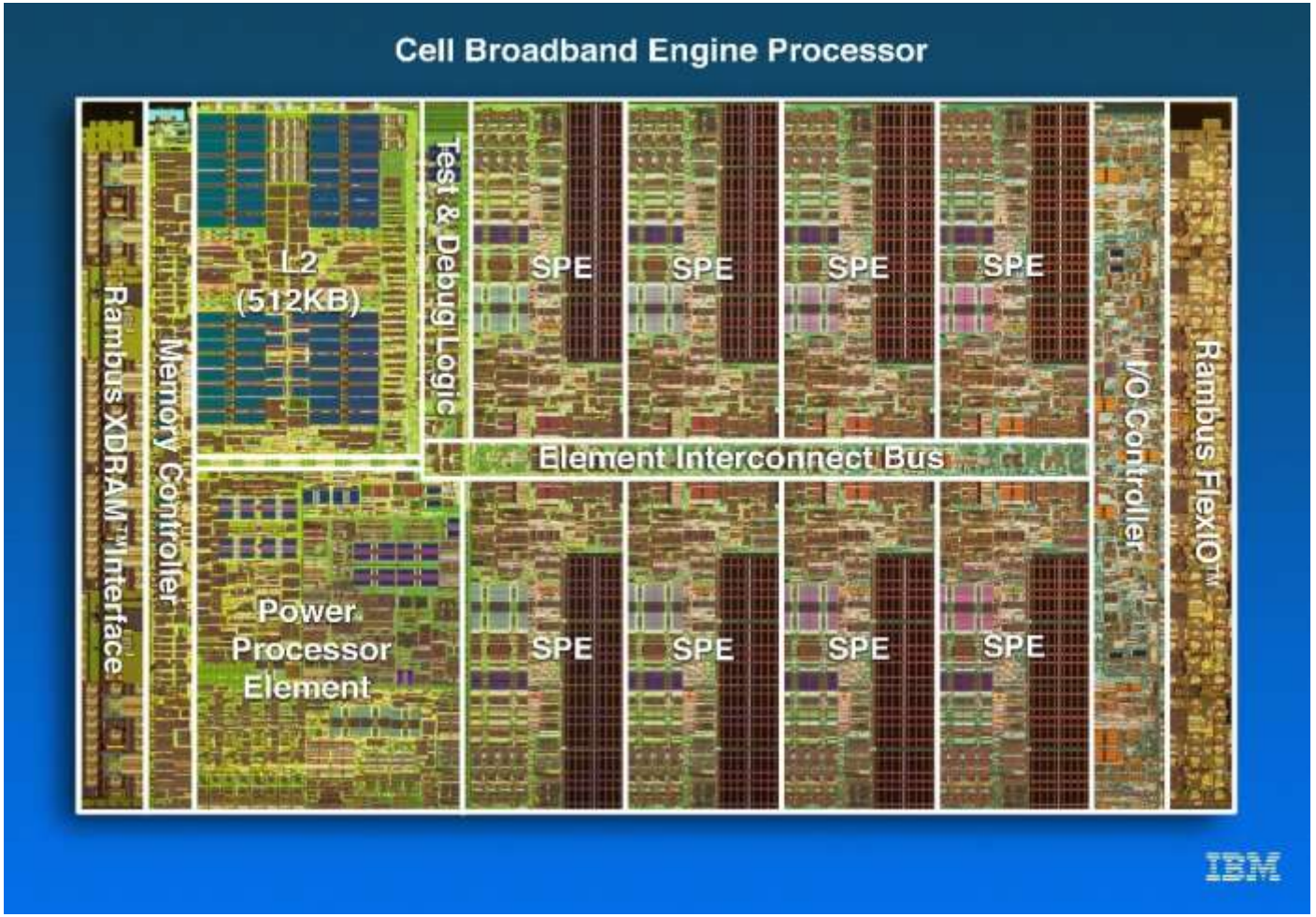


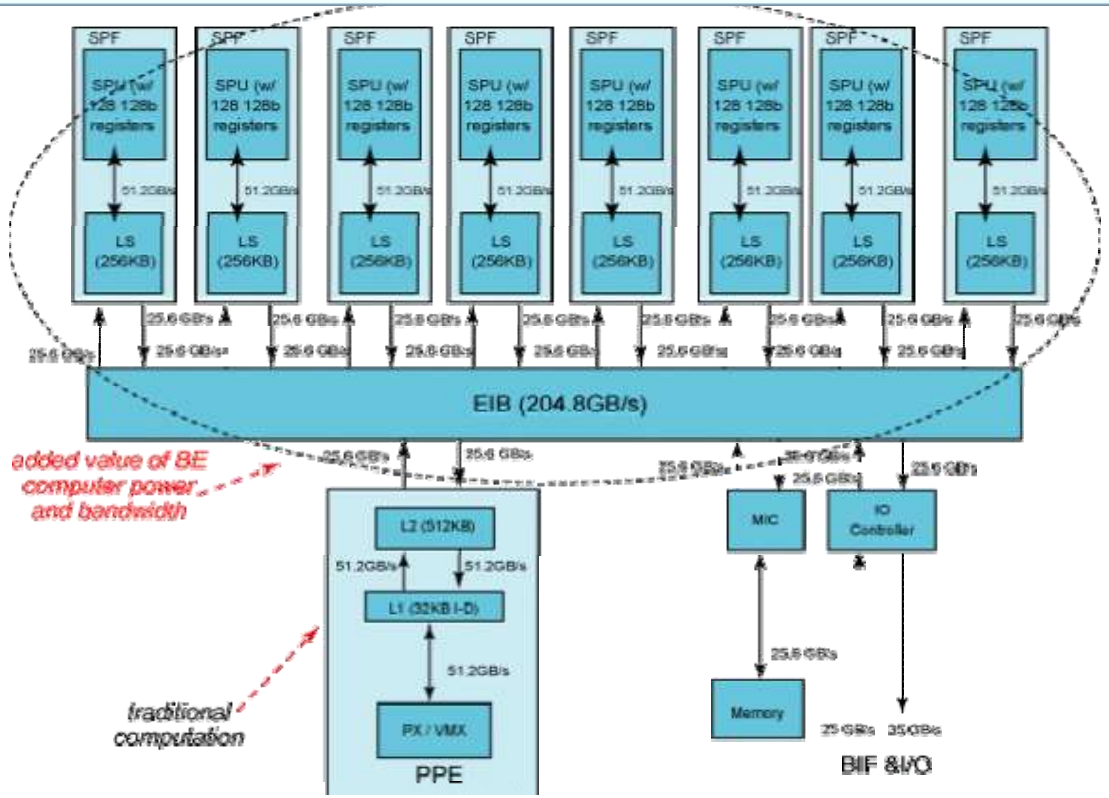

Even Pipeline
Floating Point Unit
Fixed Point Unit
Odd Pipeline
Permute Unit
Local Store Unit
Channel Unit
Branch Unit
256kB Local Store
Result Forwarding and Staging
Register File
Instruction Issue Unit / Instruction Line Buffer
128B Read
128B Write
On-Chip Coherent Bus
Memory Flow Controller (MFC)
8 Byte/Cycle
16 Byte/Cycle
64 Byte/Cycle
128 Byte/Cycle

- Provides integration between communicating SPE's within the system architecture
- Utilizes the Power protection model and address translation, incorporating the SPE's into the system protection and address translation requirement
- Allows data transfer and synchronization capabilities to interface with the system Element Interconnect Bus (EIB)
- Supports the communication interface between the PPE and SPE elements, serving as a high performance data transfer engine between local store memories and Cell system memory
- Supports the communication interface between the PPE and SPE elements, serving as a high performance data transfer engine between local store memories and Cell system memory
- Parallel data transfer and processing are capable due to the offloading of data transfers from compute elements onto dedicated data transfer engines
- Asynchronous high performance data transfer along with parallel data processing on the PPE and SPE's eliminate the need to explicitly interleave data processing and transfer at program level.

- Element Interconnect Bus (EIB)
- High-bandwidth on-chip element interconnection bus (EIB), allowing on-chip interactions between processor elements.
- Coherant bus allowing single address space to be shared by the PPE and SPE's for efficient communication and ease of programming
- Peak bandwidth of 205.8GB/s
    - 4 x 16B wide channels




Cell Broadband Engine Processor
Rambus XDRAM™ Interface
Memory Controller
L2 (512KB)
Test & Debug Logic
SPE
SPE
SPE
SPE
I/O Controller
Rambus FlexIO™
Element Interconnect Bus
Power Processor Element
SPE
SPE
SPE
SPE
IBM


SPF
SPU (w/ 128 128b registers
LS (256KB)
EIB (204.8GB/s)
added value of BE computer power and bandwidth
traditional computation
L2 (512KB)
L1 (32KB I-D)
PX / VMX
PPE
MIC
IO Controller
Memory
BIF &I/O

- 2 x 32-bit channels, with 3.2Gbit/s data transfer per pin
- 25.6GB/s peak data transfer
  - potentially scalable to 6.4Gbit/s, offering a peak rate of 51.2GB/s
- “FlexPhase” Technology
  - signals can arrive at different times, reducing need for exact wire length
- Configurable to attach to variable amounts of main memory
- Memory Interface Controller (MIC) controls the flow of data between the EIB and the XDR RAMBUS

- RAMBUS RRAC FlexIO used for I/O
  - 7 transmit, 5 receive 8-bit ports
    - 35GB/s transmit peak
    - 25GB/s receive peak
  - Used as a high speed interconnect between SPE's of different Cells when a multiple Cell architecture is in place
    - 4 Cells can be seamlessly connected given the current architecture, an additional 4 with an added switch

- "single source" parallelizing, simdizing compiler
  - generates multiple binaries targeting both the PPE and SPE elements from one single source file
  - allows programmers to develop applications for parallel architectures with the illusion of a single shared memory image
- Compiler-controlled software-cache, memory hierarchy optimizations, and code partitioning techniques assume all data resides in a shared system memory
  - enables automatic transfer of data and code
  - preserves coherence across all local SPE memories and system memory

- In order to obtain optimum performance within the Cell both the SPE's LS and Cell's SIMD dataflow must be taken into account by the programmer, and ultimately the programming model
- Function Offload Model
  - SPE's used as accelerators for critical functions designated by the PPE
    - original code optimized and recompiled for SPE instruction set
    - designated program functions offloaded to SPE's for execution
    - when PPE calls designated program function it is automatically invoked on the SPE
  - Programmer statically identifies which program functions will execute on PPE and which will be offloaded to the SPE
    - separate source files and compilation for PPE and SPE
  - Prototype single-source approach using compiler directives as special offload hints has been developed for the Cell
    - challenge to have compiler automatically determine which functions execute on PPE and SPE
      - allows applications to remain compatible with the Power Architecture generating both PPE and SPE binaries
        - Cell systems load SPE-accelerated binaries
        - Non-Cell systems load PPE binaries

- SPE's act as interfaces between PPE and external devices
  - Use memory-mapped SPE-accessible registers as command/response FIFO between PPE and SPE's
  - Device memory can be mapped by DMA, supporting transfer size granularity as small as a single byte

- Computational Acceleration Model
  - SPE centralized model, offering greater integration of SPE's in application execution and programming than function offload model
  - Computational intensive code performed on SPE's rather than PPE
  - PPE acts as a control center and service engine for the SPE software
    - Work is partitioned among multiple SPE's operating in parallel
      - manually done by programmers or automatically by compiler
      - must include efficient scheduling of DMA operations for code and data transfers between PPE and SPE
      - utilize shared-memory programming model or a supported message passing model
  - Generally provides extensive acceleration of intense math functions without requiring significant re-write or re-compilation

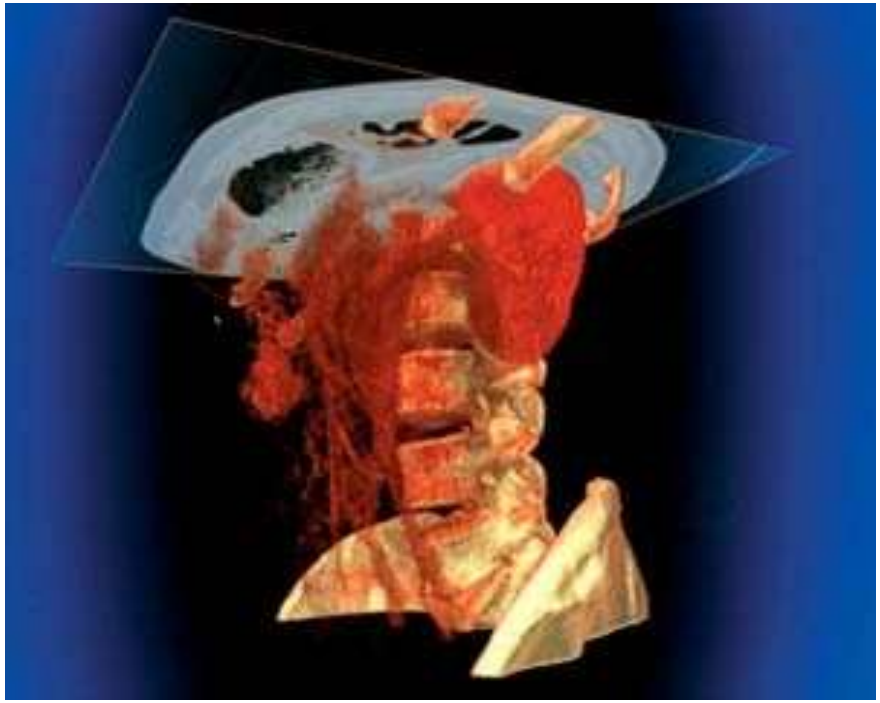
- Construct serial or parallel pipelines using the SPE's
  - Each SPE applies a particular application kernel to the received data stream
  - PPE acts as a stream controller, with SPE's acting as data stream processors
- Extremely efficient when each SPE performs an equivalent amount of work

- Shared Memory Multiprocessor Model
  - Utilizes the two different instruction sets of the PPE and SPE to execute applications
    - provides great benefits to performance when utilizing just a single instruction set for a program would be very inefficient
  - Since DMA operations are cache coherent, combining DMA commands with SPE load/store commands provides the illusion of a single shared address space
    - conventional shared-memory loads are replaced by a DMA from system memory to an SPE's LS, and then a load from the LS to the SPE's register file
    - conventional shared-memory stores are replaced by a store from an SPE's register file to it's LS, then a DMA from the SPE's LS to system memory
  - Affective addresses in common with both PPE and SPE are used for all “pseudo” shared-memory load/store commands
  - Possibilities exist for a compiler or interpretor to manage all SPE LS's as local caches for instruction and data

SPE are possible
  - Interaction among threads as in a conventional SMP
- Similar to thread or lightweight task models of conventional operating systems
  - extensions exist to include processing units with different instruction sets as exists within the PPE and SPE's
- Tasks are scheduled on both the PPE and SPE in order to optimize performance and utilization of resources
  - abilities of the SPE to run only a single thread is thus hidden from programmer

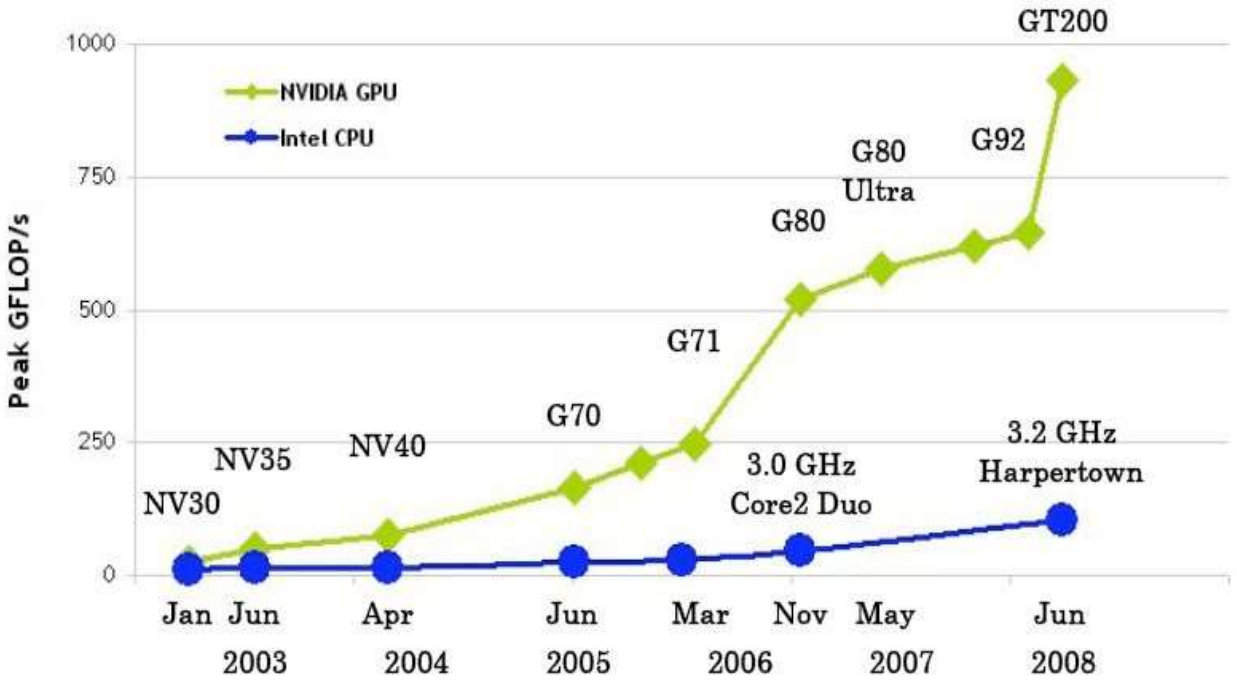
- Games (PS3)
  - Audio
  - Video
  - physics

- Imaging
  - Medical, rendering, feature extraction
  - Mercury – Blade, Turismo
- Televisions (Toshiba)
  - MPEG decode



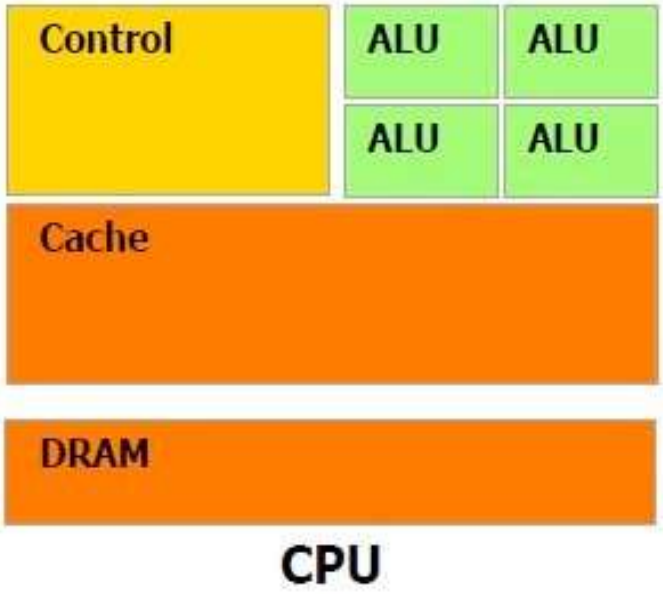
- GPU: Graphics Processing Unit
- Hundreds of Cores
- Programmable
- Can be easily installed in most desktops
- Similar price to CPU
- GPU follows Moore's Law better than CPU

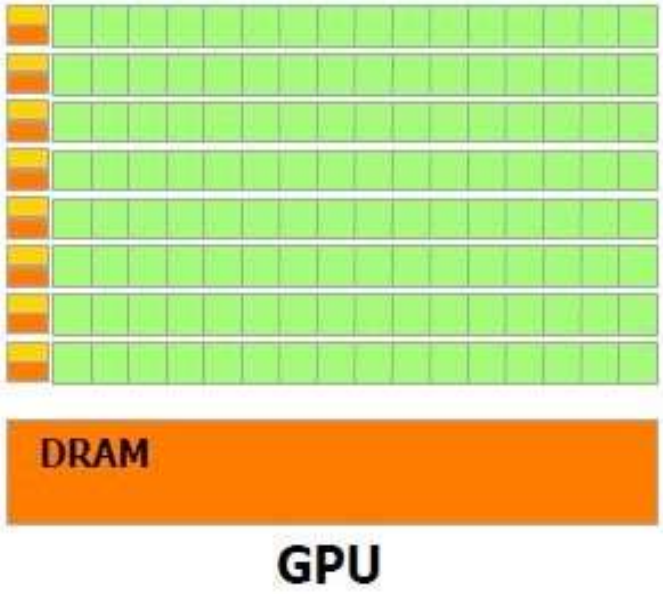
Motivation:



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

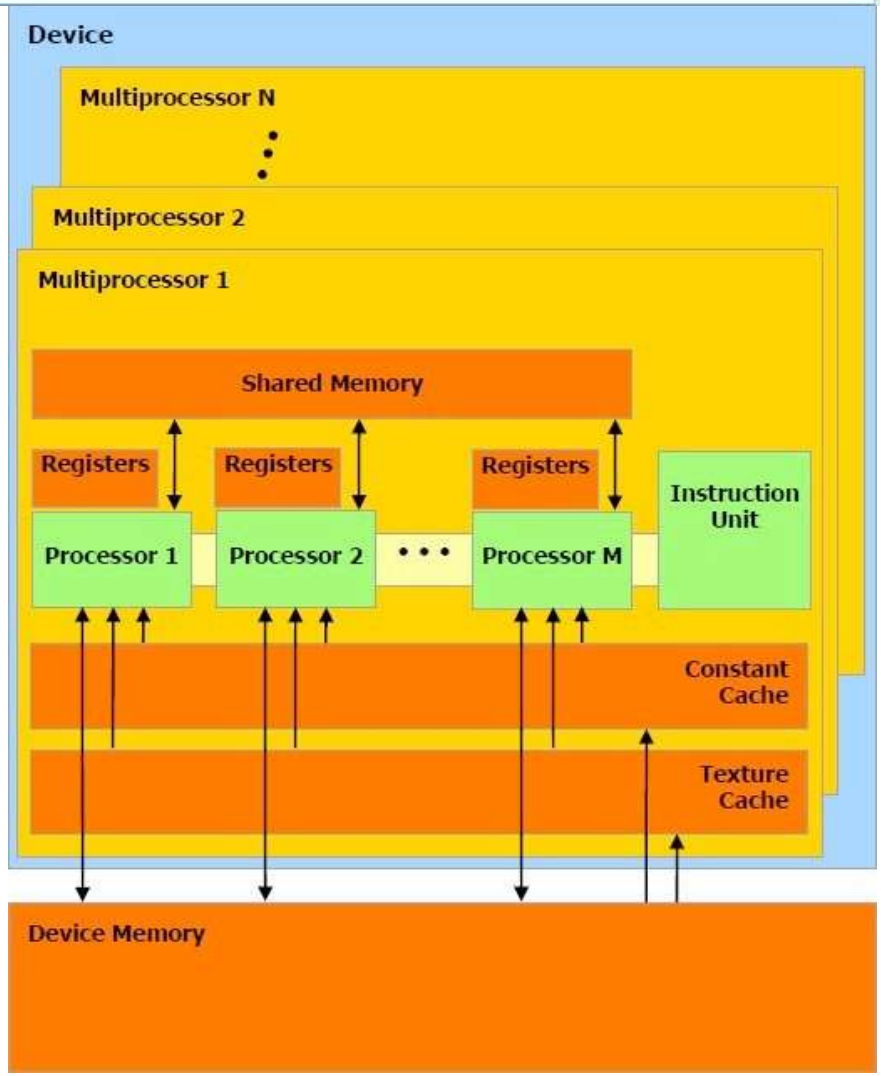
Multiprocessor Structure:





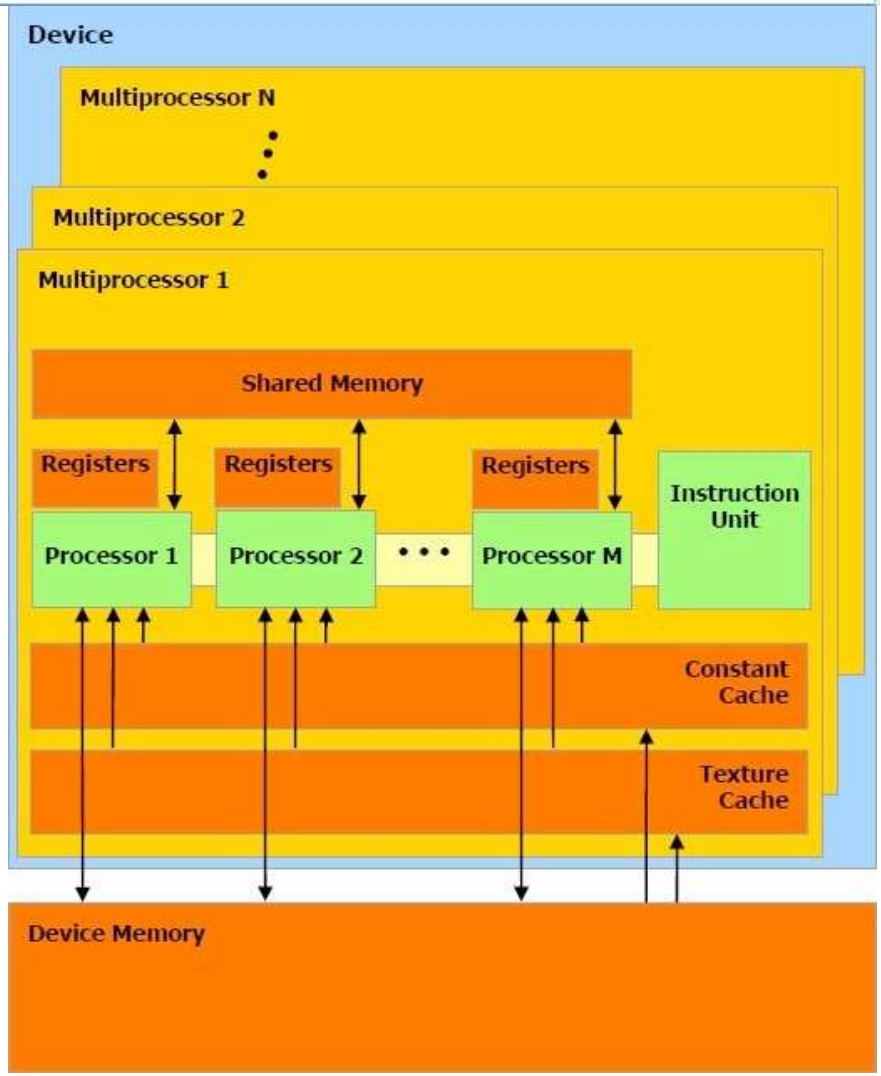
Multiprocessor Structure:

- N multiprocessors with M cores each
- SIMD – Cores share an Instruction Unit with other cores in a multiprocessor.
- Diverging threads may not execute in parallel.



Memory Hierarchy:

- Processors have 32-bit registers
- Multiprocessors have shared memory, constant cache, and texture cache
- Constant/texture cache are read- only and have faster access than shared memory.



Past:

- The GPU was intended for graphics only, not general purpose computing.
- The programmer needed to rewrite the program in a graphics language, such as OpenGL
- Complicated

Present:

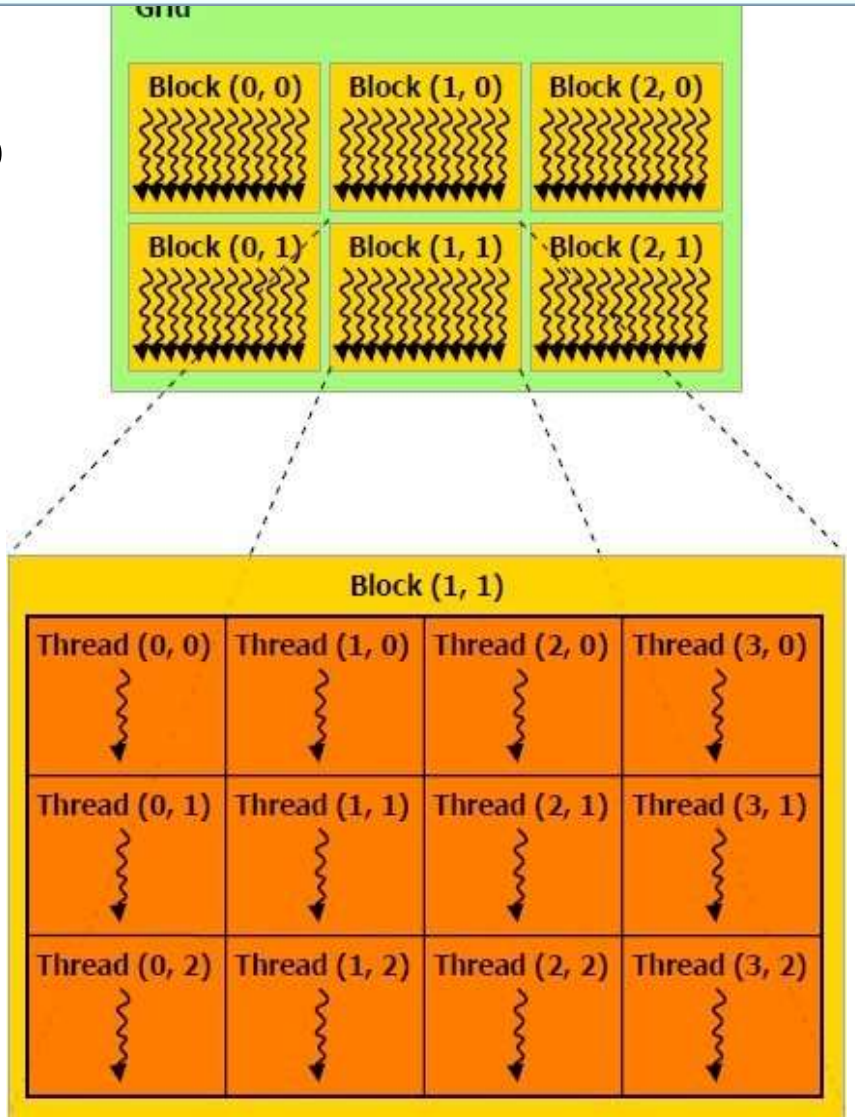
- NVIDIA developed CUDA, a language for general purpose GPU computing
- Simple

CUDA:

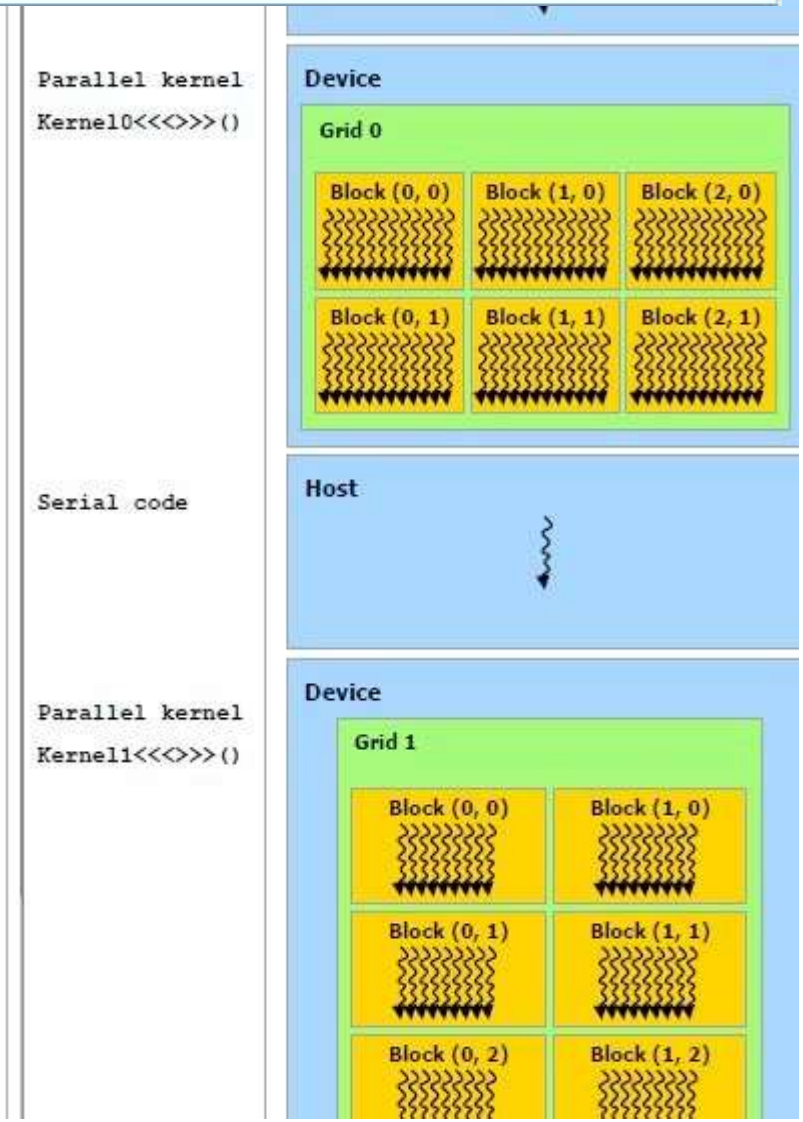
- Compute Unified Device Architecture
- Extension of the C language
- Used to control the device
- The programmer specifies CPU and GPU functions
  - The host code can be C++
  - Device code may only be C
- The programmer specifies thread layout

Thread Layout:

- Threads are organized into *blocks*.
- Blocks are organized into a *grid*.
- A multiprocessor executes one block at a time.
- A *warp* is the set of threads executed in parallel
- 32 threads in a warp



- Heterogeneous Computing:
  - GPU and CPU execute different types of code.
  - CPU runs the main program, sending tasks to the GPU in the form of kernel functions
  - Multiple kernel functions may be declared and called.
  - Only one kernel may be called at a time.



CPU C program

```
void add_matrix_cpu
                (float *a, float *b, float *c, int N)
{
      int i, j, index;
    for (i=0;i<N;i++) {
     for (j=0;j<N;j++) {
        index =i+j*N;
        c[index]=a[index]+b[index];
        }
    }
}
void main()
{
   .....
       add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu
                (float *a, float *b, float *c, int N)
{
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index =i+j*N;
    if( i <N && j <N) c[index]=a[index]+b[index];
}



void main()
{
    dim3 dimBlock (blocksize,blocksize);
    dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
    add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

Tesla C1060
GPU 933
GFLOPS

nForce
Motherboard

Tesla C1070 Blade 4.14
TFLOPS

- Intel: Software is the New Hardware !
- Intel: x86 ISA makes parallel program easier
  - Better flexibility and programmability
  - Support subroutine call and page faulting
  - Mostly software rendering pipeline, except texture filtering

- Lots of x86 cores (8 to 64?)
- Fully coherence L2 cache
- Fully Programmable Rendering Pipeline
- Shared L2, Divided L2
- Cache Control Instructions
- Ring Network
- 4-Way MT

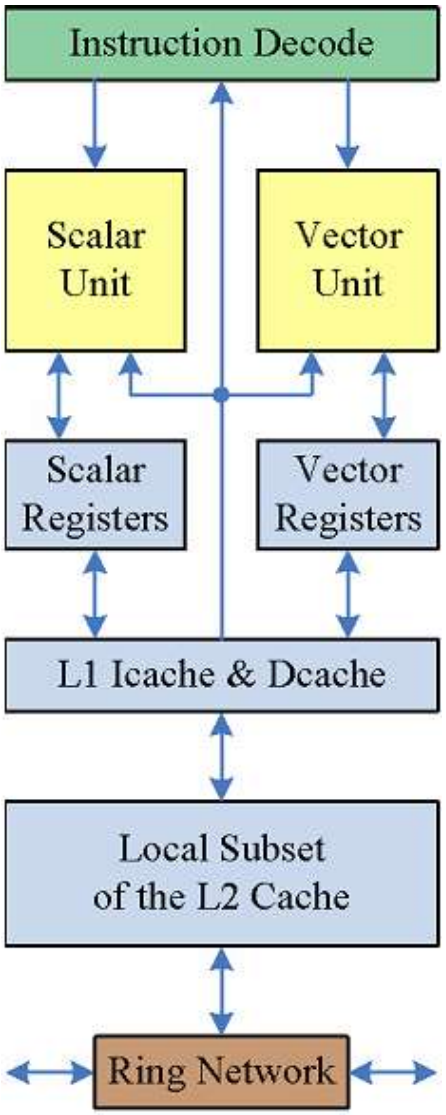
- Larrabee will use the x86 instruction set with Larrabee-specific extensions
- Larrabee will feature cache coherency across all its cores.
- Larrabee will include very little specialized graphics hardware, instead performing tasks like z-buffering, clipping, and blending in software, using a tile-based rendering approach.
- more flexible than current GPUs

- Based on the much simpler Pentium P54C design.
- Each Larrabee core contains a 512-bit vector processing unit, able to process 16 single precision floating point numbers at a time.
- Larrabee includes one major fixed-function graphics hardware feature: texture sampling units.
- Larrabee has a 1024-bit (512-bit each way) ring bus for communication between cores and to memory.
- Larrabee includes explicit cache control instructions.
- Each core supports 4-way simultaneous multithreading, with 4 copies of each processor register.

- 512 bit vector types (8×64bit double, 16×32bit float, or 16×32bit integer)
- Lots of 3-operand instructions, like a=a*b+c
- Most combinations of +,-,*,/ are provided, that is, you can choose between a*b-c, c-a*b and so forth.
- Some instructions have built-in constants: 1-a*b
- Many instructions take a predicate mask, which allows to selectively work on just a part of the 16-wide vector primitive types
- 32bit integer multiplications which return the lower 32bit of the result
- Bit helper functions (scan for bit, etc.)
- Explicit cache control functions (load a line, evict a line, etc.)
- Horizontal reduce functions: Add all elements inside a vector, multiply all, get the minimum, and logical reduction (or, and, etc.).

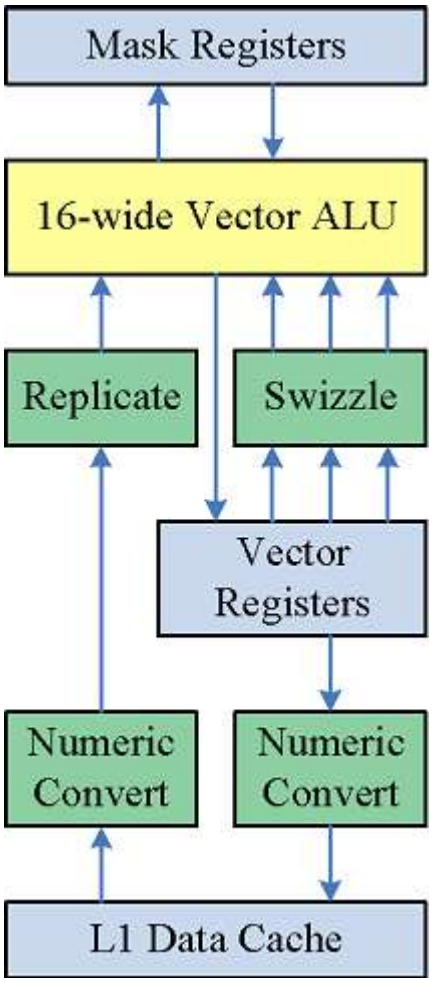
- Scalar Processing Unit
- Vector Processing Unit
- Separated Register File
- Communication via memory

Here is the block diagram of one Larrabee core. Basically, there are two Processing Unit in each cores: scalar processing unit and a vector processing unit. Although the two processing unit are located within the same core, they have completely separate register files. Moreover, there are not direct channel between scalar unit and vector unit, the nearest sharing module is L1 cache. That's to say, the two units can not talk to each other unless using the system memory. The communication overhead maybe varied. If the shared memory is located in cache, the communication cost will be negligible.

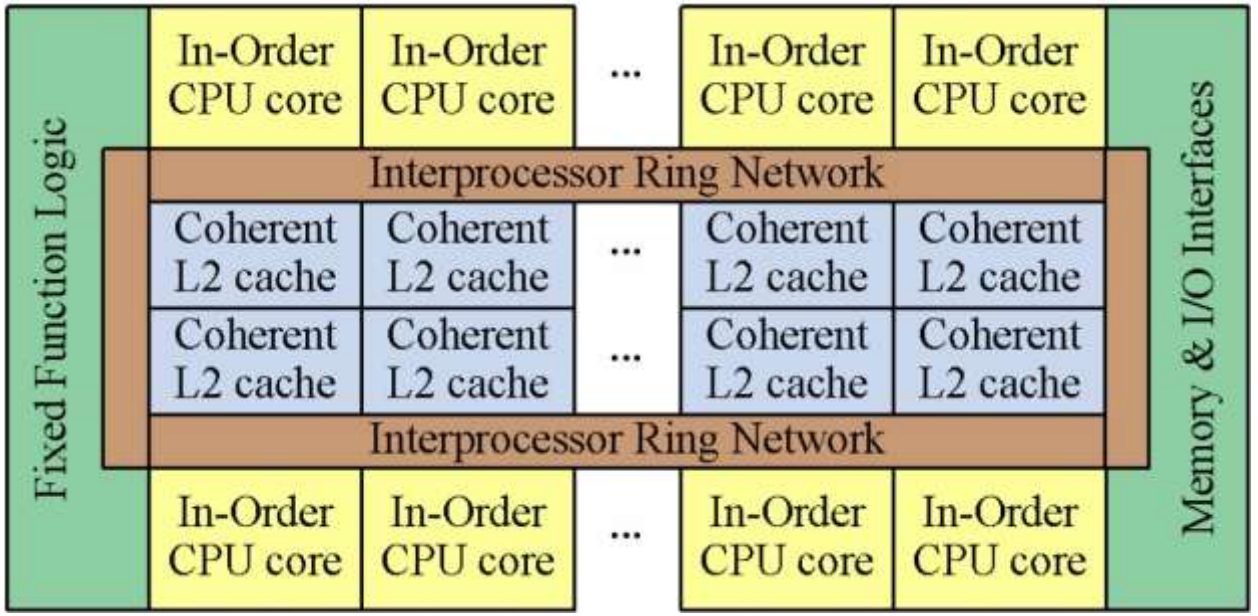


- Derived from Pentium(1990s)
  - 2-issue superscalar
  - In-order
  - Short, inexpensive pipeline execution
- But more than that
  - 64-bit
  - Enhanced multithreading – 4 threads per core
  - Aggressive pre-fetching
  - ISA extension, new cache features.

- 16-wide SIMD, each with 32bits wide data.
- Load/stores with gather/scatter support.
- Mask register enables flexible read and store within packed vector.



- L1
  - 8K I-Cache and 8k D-Cache per thread
  - 32K I/D cache per core
  - 2-way
  - Treated as extended registers
- L2
  - Coherent
  - Shared & Divided
  - A 256K subset for each core



Reason for doing this:

- Simplify the design;
- Cut costs;
- Get to the market faster

- Threads (Hyper-Threading)
  - Hardware-managed
  - As heavy as an application program or OS
  - Up to 4 threads per core
- Fibers
  - Software-managed
  - Chunks decomposed by compliers
  - Typically up to 8 fibers per thread

- Strands
  - Lowers level
  - Individual operations in the SIMD engines
  - One strand corresponds to a thread on GPUs
  - 16 Strands because the 16-lane VPU

- "FULLY" programmable
- Legacy code easy to migrate and deploy
  - Run both DirectX/OpenGL and C/C++ code.
  - Many C/C++ source can be recompiled without modification, due to x86 structures.
  - Crucial to large x86 legacy programs.
- Limitations
  - System call
  - Requires application recompilation

- Architecture level threading:
  - P-threads
  - Extended P-threads to support developers to specify thread affinity.
  - Better task scheduling (task stealing by Bluemofe,1996 & Reinders 2007), lower costs for threads creation and switching
  - Also supports OpenMP

- Larrabe native binaries tightly bond with host binaries.
- Larrabe library handles all memory message/data passing.
- System calls like I/O functions are proxied from the Larrabe app back to OS service.

- Larrabe Native is also designed to implement some higher level programming languages.

- Ct, Intel Math Kernel Library or Physics APIs.

- Complex pointer trees, spatial data structures or large sparse n-dimensional matrices.

- Developer-friendly: Larrabe allows but does not require direct software management to load data into memory.

- Nvidia Geforce
  - Memory sharing is supported by PBSM(Per-Block Shared Memories), 16KB on Geforce 8.
  - Each PBSM shared by 8 scalar processors.
  - Programmers MUST explicitly load data into PBSM.
  - Not directly sharable by different SIMD groups.
- Larrabe
  - All memory shared by all processors.
  - Local data structure sharing transparantly provided by coherent cached memory hierarchy.
  - Don’t have to care about data loading procedure.
  - Scatter-gather mechanism

Parul® University

NAAC GRADE A++

https://paruluniversity.ac.in/