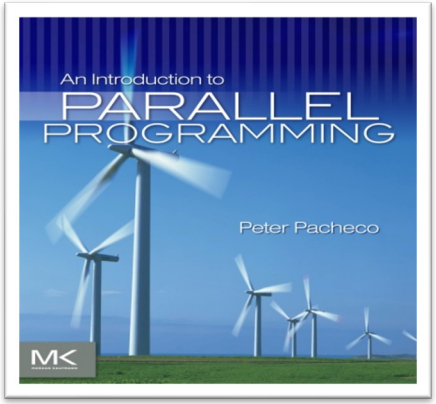
14013204-3 - PARALLEL COMPUTING



Parallel Program Design

Roadmap

- Parallel Program Design
  - Main coordination aspects and Related Topics
    a. Work decomposition
    b. Parallel programming patterns
    c. Work Mapping (Load Balancing)
    d. Performance Tunning

Parallel Performance -- Review

- **Performance**
  - Depends on the total elapsed time of an algorithm's execution.
  - Less elapsed time means higher performance.

- **Speedup**
  - performance metric comparing two elapsed time values.
  - In parallel computing, these two values are usually generated by the execution of a serial algorithm and its parallelized version.
  - **Speedup = Serial Execution Time / Parallel Execution Time**
  - Example: a serial algorithm takes 100 seconds to complete, and the parallel version takes 40 seconds, the speedup is "2.5x".

Parallel Performance -- Review

- **Efficiency**
  - A metric derived from speedup by adding awareness of the utilized hardware.
  - **Efficiency = Speedup / # of cores**
  - So, if speedup is “2.5x” on a 4-core machine, efficiency is 0.625 or 62.5%.

- **Scalability**
  - It refers to the speedup of an algorithm given different numbers of cores/processors.
  - The efficiency metric is good for quantifying scalability, because if efficiency holds constant as the number of cores changes, we have linear scaling (great scalability).

PARALLEL PROGRAM DESIGN

Designing Parallel Programs

- The process of converting a serial program or algorithm into a parallel program is often called **parallelization**.
- In sequential programs we focus only on **computation**
- In the parallel program we focus on **computation + coordination**
- Coordination involves different aspects.
- Computation part use **the same programming language** as the sequential code but need to use **coordination technology** to manage the coordination aspects.

Designing parallel program

- Parallel Program should be designed to tackle the **issues of locality**, **synchronizations, overhead**, **load balancing**, **scalability**, and several other challenges in parallel computing.

- By **optimizing locality**, **minimizing synchronizations**, **balancing workloads**, **ensuring scalability**, **reducing communication overhead**, and carefully managing data movement, parallel algorithms can achieve higher efficiency, speedup, and scalability.

- Successful parallel algorithm design requires **a careful balance between these factors** to get the full potential of parallel and distributed computing systems.

Designing parallel program

- **Main coordination aspects**:
  - **Work decomposition:** divide the work among the available PEs.
    1. **Data** decomposition. (Data Parallelism)
    2. **Functional** decomposition. (Task Parallelism)
  - **Work mapping:** assign (allocate) the work to PEs.
    - **Load Balancing**: dividing the work among the processes/threads so that each PE has roughly the same amount of work
    1. **Static work allocation**: Block allocation, Cyclic allocation.
    2. **Dynamic work allocation**: work queue.
  - **Communication and Synchronization the management of communication and synchronization.**
- These aspects are **interrelated components**.

Designing parallel program

- Will cover these related topics to the coordination aspects:
  - **Patterns:** Master/worker, Data Pipelining, Divide and Conquer.
  - **Identify and reduce different sources of performance issues (Performance Tuning):**
    - Dependences.
    - Granularity.
    - Data locality
- **PE (Processing Element)** is an abstraction of a core. We use PE in the programming language context, and core in hardware level discussions.

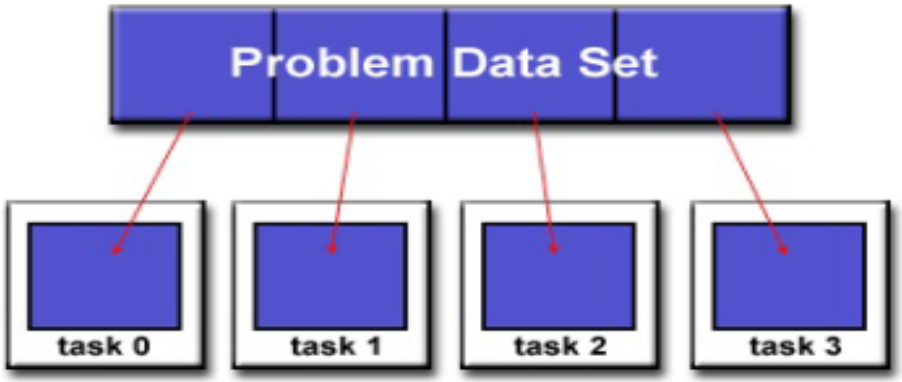
Work decomposition

- Break the problem into **discrete "chunks"** of work that can be distributed to multiple PEs. This is known as **decomposition** or **partitioning**.
- The **equal division** of tasks among the PEs is the **load balancing**.
- Two main widely used approaches to decomposition (Type of Parallelism).
  1. Task Parallelism
  2. Data Parallelism
- In many parallel applications, **both** data decomposition and task decomposition are used together.
- The choice to use **data decomposition** and/or **task decomposition** depends on the **nature of the problem**, the **characteristics of the data**, and **the computational tasks** involved in the parallel algorithm.

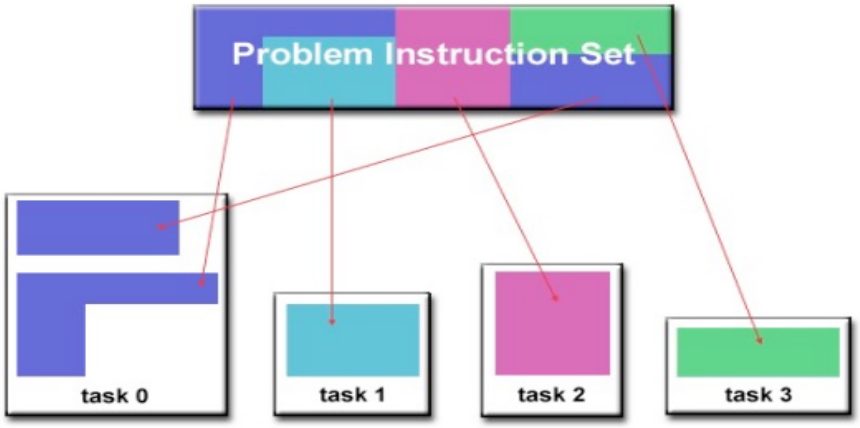
Work decomposition

1. Data (Domain) decomposition:
The **data** associated with the problem to be solved **is divided among the PEs**. each data item is subjected to more or less **the same sequence of instructions.**

- This type of parallelism is called **data parallelism.**



2. Functional (Task) decomposition:
involves **breaking down complex computational problems** into **smaller, independent tasks**, each of which can be executed concurrently by Pes.
This type of parallelism is called **task parallelism.**



Parallel programming patterns

- To exploit parallel architectures efficiently, **computation needs to be partitioned into smaller sub-tasks, which are mapped to PEs** in a way that
  - sets a suitable balance between **computational load** to exploit the parallel cores, and the **preservation of data locality** to reduce the overheads associated with the communication.
- A general trend in the parallel languages' community is to define **the parallel programming pattern** in which applications can be classified.
  - These patterns represent **approaches for structuring the parallel program**.
  - They are defined to manage **the work decomposition and the load balancing components of parallelism.**

Parallel programming patterns

- Most widely used **patterns**:
  1. **Master/Workers (Task Farming)**
  2. **Divide and Conquer.**
  3. **Data Pipeline**
  4. There are others e.g., loop-parallelism

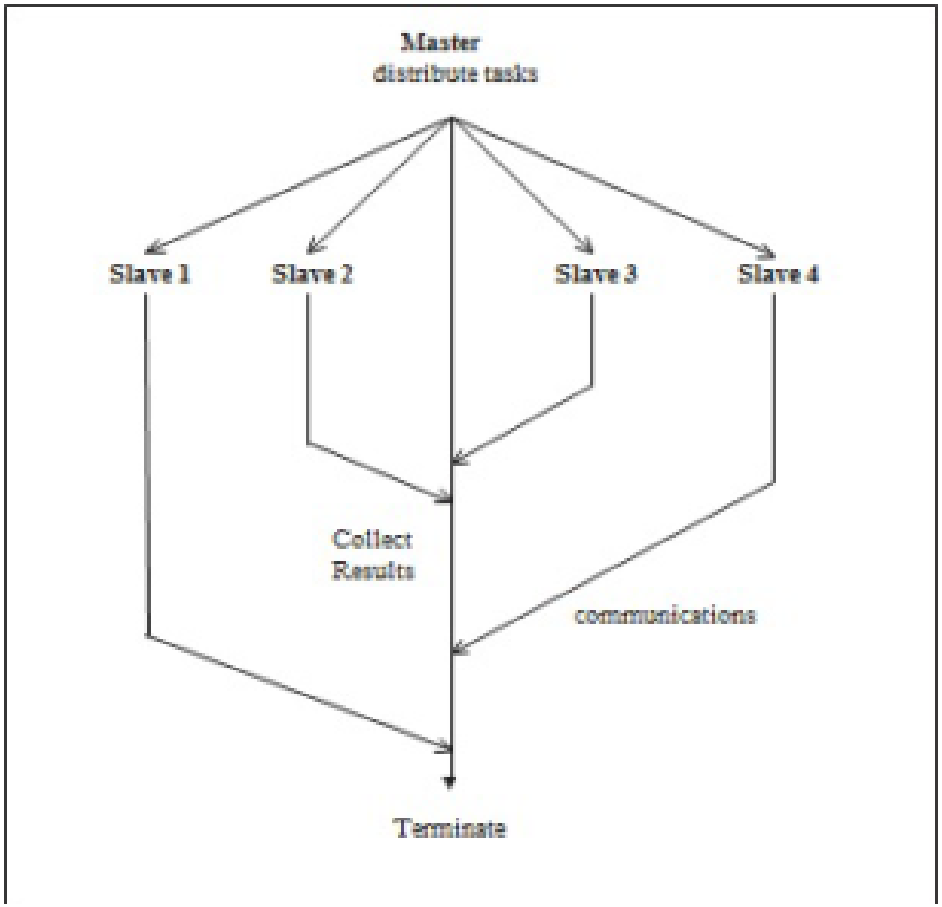
- With all these patterns, the **load balancing** is essential to improve the performance and can be achieved either **statically or dynamically**.
- static and dynamic balancing of load is achieved through static and dynamic work allocation which will be covered in the following point.

Master/Worker

- A model for **problem decomposition**.
- The program comprises two entities: the **master**, and multiple **workers** or **slaves**.

1. The master's job is to decompose the problem into a number of tasks and distribute them amongst the workers. Subsequently, when the computation has been completed, the master collects the partial results, so that the final computation result can be produced.
2. The slaves (or workers) execute tasks in a very simple cycle: they receive a message with the task, then process the task before returning the results to the master.

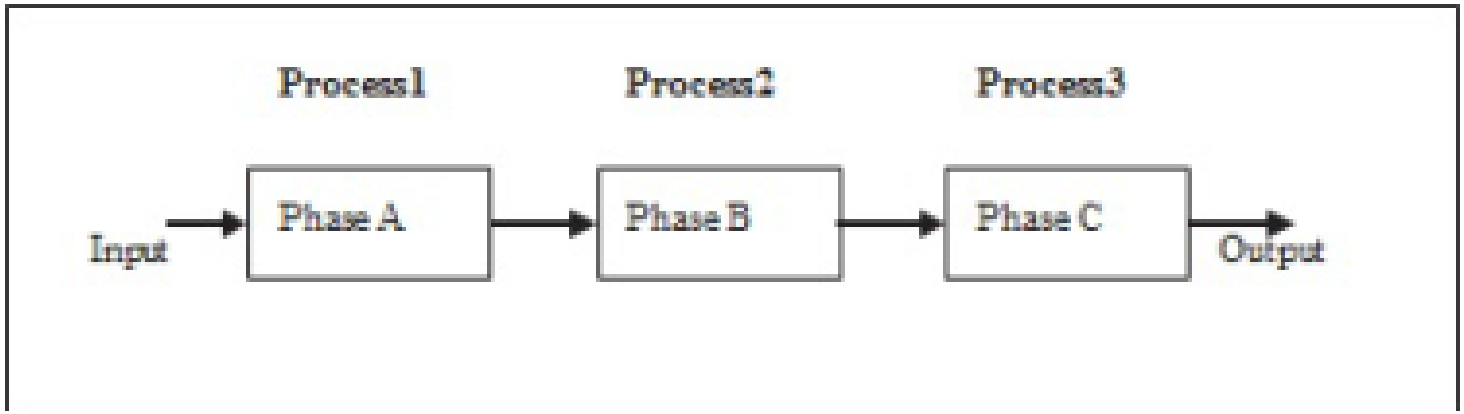
Master/Worker



Data Pipelining

- Based on **functional decomposition** and is one of the simplest pattern and one of the most commonly applied.
- The **different tasks** of the algorithm, which operate **concurrently**, are identified.
- Then, each PE executes its small portion of the total algorithm.
- The **processes** are arranged as **different stages of the Pipeline**, each one of which is responsible for executing a particular task.
- This type of parallelism is sometimes called **data flow parallelism** because the data flows between one stage of the pipeline and the next.
- It is commonly used in applications such as **image processing**.

Data Pipelining

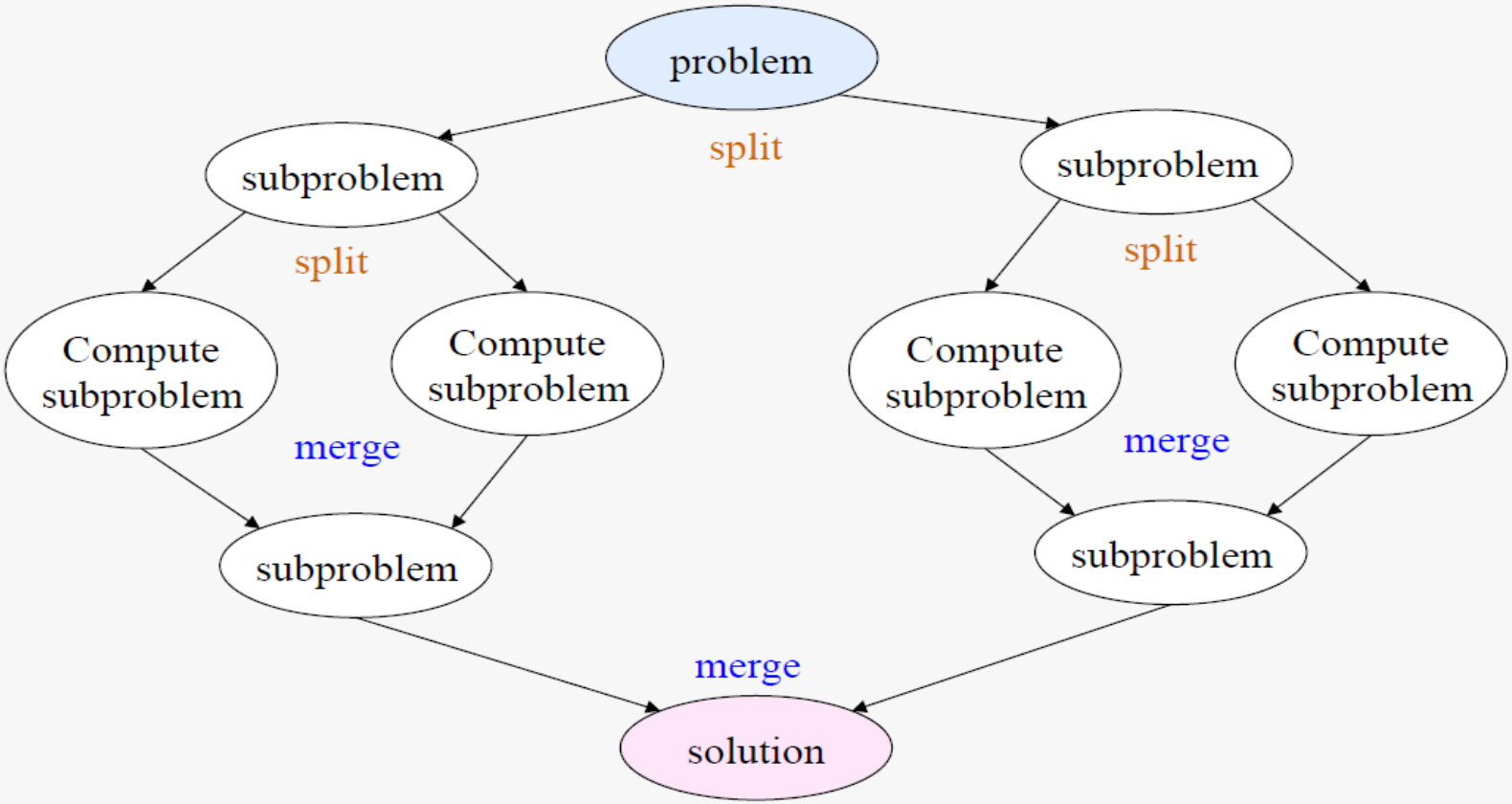


Divide and Conquer

- It is a frequently used approach in the **design and development of sequential algorithms**.
- A **task is broken down recursively** into a number of smaller sub-tasks, which sometimes represent **smaller samples of the original task**, each of which is computed independently, then results are combined.
- In parallel divide-and-conquer, the division of a task into smaller sub-tasks, and a combination of results can be carried out in parallel.
- The divide-and-conquer paradigm consists of **three main phases**: **divide, compute, and combine.**
- These are designed in the form of a **virtual tree**, whereby some of the PEs create sub-tasks, and the results then combined to produce an overall result.
- The computation of sub-tasks is carried out by the **leaf nodes** of the virtual tree.

Divide and Conquer

A problem is structured to be solved in sub-problems independently, and merging them later.



Work mapping : static work allocation

- Important part of parallel programming is splitting work between PEs to **balance the load**.
- Assigning work statically means:
  - Dividing the work between PEs **before** doing the work.
  - Example: array sum
  - In our examples, we have always assigned parts of the array to each PE before they start working.

Work mapping: static work allocation

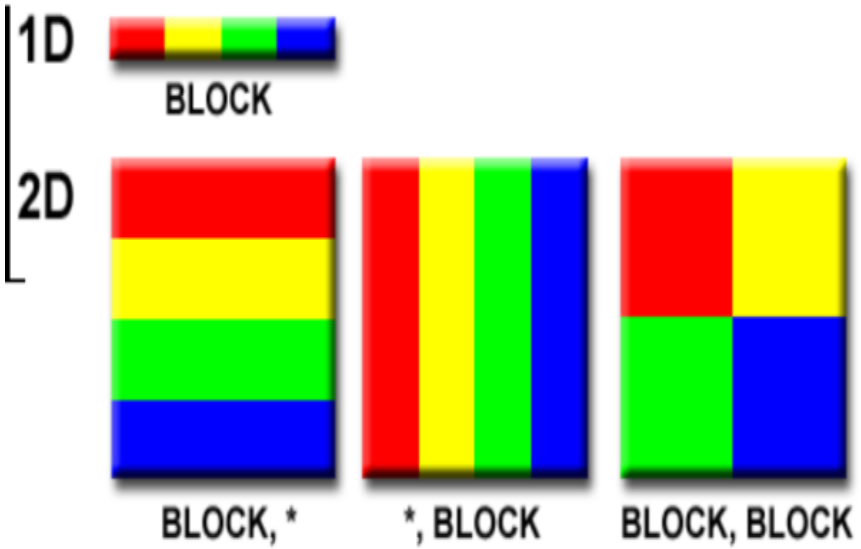
```
int numThreads = 2; //Assume one thread per core, & 2 cores
int sum = 0;
int i = 0;
int middleSum[numThreads];
int threadSetSize = size/numThreads
//Each thread will execute this code with a different threadID
for( i = threadID*threadSetSize; i < (threadID+1)*threadSetSize
    ; i++)
{
middleSum[threadID] += array[i];
}
waitForAllThreads(); //Wait for all threads
//Only thread 0 will execute this code
if (threadID==0) {
for(i = 0; i < numThreads; i++) {
sum += middleSum[i];
}}
```

Work mapping: static work allocation

- There are three main approaches (can be combined):
  1. **Block Allocation:** large dataset is divided into fixed-size blocks, and each block is assigned to a different PE, allowing parallel processing on these data blocks.
  2. **Cyclic Allocation:** assigns data elements in a cyclical manner to PEs. For example, in cyclic data distribution, elements are assigned to PEs in a **round-robin** fashion. In cyclic task distribution, tasks are allocated to PEs sequentially, and the pattern repeats. This approach ensures that PEs work on different parts of the dataset or tasks, promoting load balancing.
  3. **Block-Cyclic Allocation:** combines aspects of **both** block allocation and cyclic allocation. The dataset is divided into blocks, and these blocks are distributed to PEs in a cyclic manner. This method aims to **balance** the advantages of block-wise organization and cyclical distribution, providing a balanced workload and efficient data access patterns.

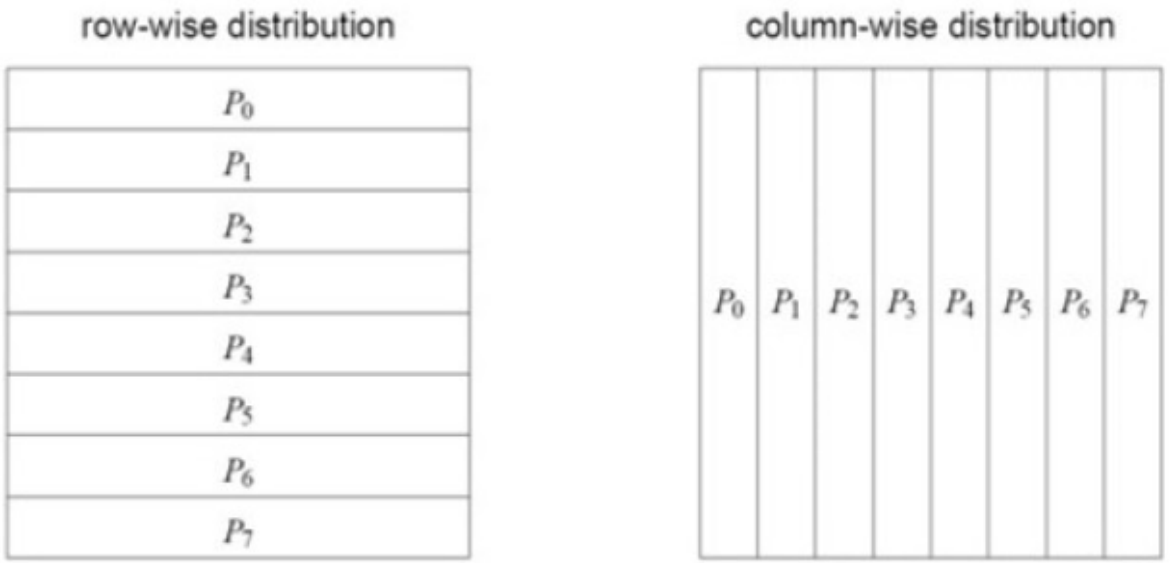
Work mapping: static work allocation

- **Block Allocation:**
  - Divide data into blocks and assign each block to a PE.
  - Usually used when **Data is regular-shaped**.
  - When **the data size is fixed**.
    - E.g. an array of fixed size



Work mapping: static work allocation

- **Block Allocation:**
- For 1-dimension arrays, a block is a set of consecutive indices Like an array sum.
- Benefit: memory and cache locality.

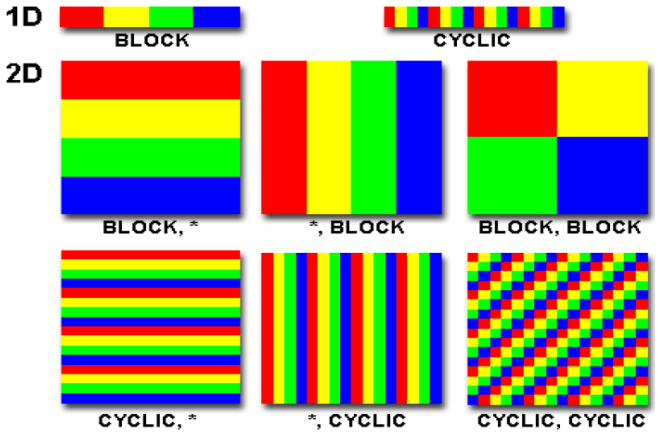


Work mapping: static work allocation

- **Block Allocation:**
- Block distribution can be generalized to higher dimensions.

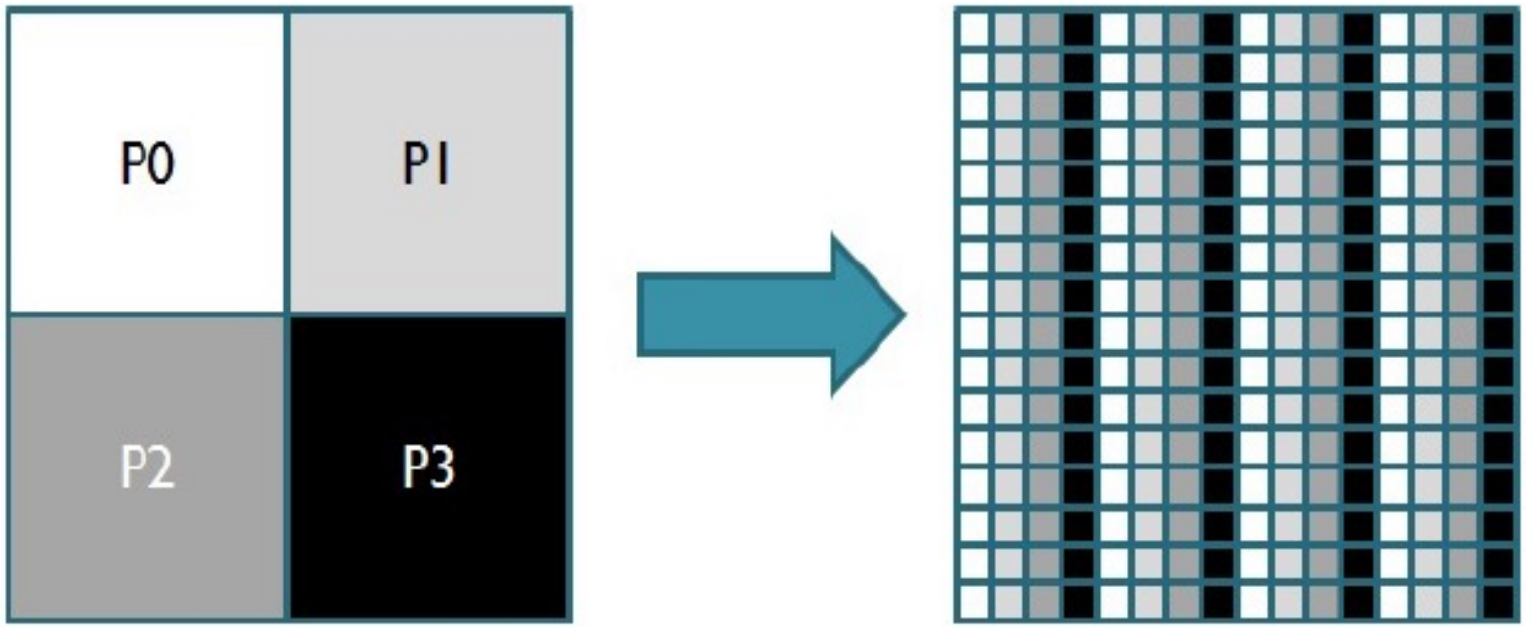
Work mapping: static work allocation

- **Cyclic Allocation:**
  - Allocate data elements to PEs in a **round-robin** manner
  - Why?
    - **Data is regular** and can be divided statically.
    - **Work for each data element varies**, BUT we know how much work each data element needs.
  - Example: If the amount of work differs for different entries of a matrix, a block distribution can lead to **load imbalances**.



Work mapping: static work allocation

- Cyclic Allocation: (4 PEs example)

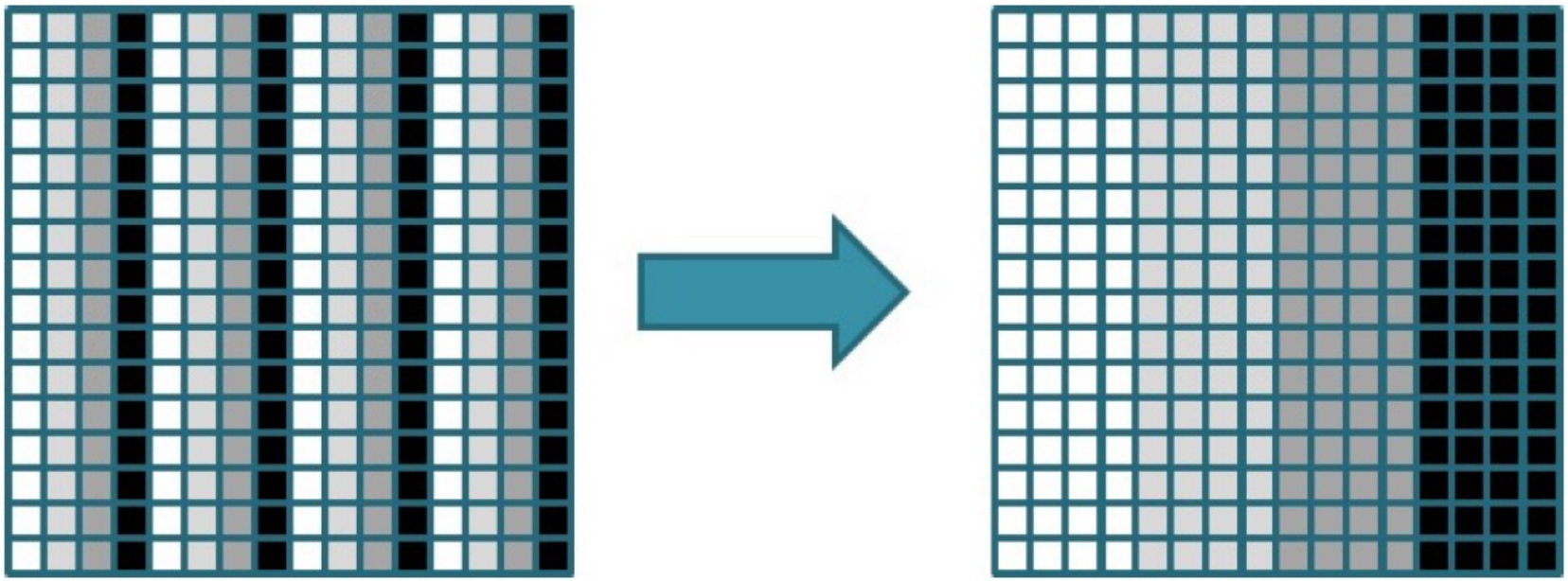


Work mapping: static work allocation

- **Block-Cyclic Allocation:**
  - Block allocation may lead to poor **load balance**.
  - Cyclic allocation may provide poor **data locality** (e.g., new cache line needed for each new data element).
  - For good performance, need to consider both **load balance**, and **data locality**
  - Solution, **block-cyclic allocation**.
    - Partition an array into many more blocks than the number of available PEs.
    - Assign blocks to PEs in a round-robin manner.
    - Each PE gets several non-adjacent blocks.

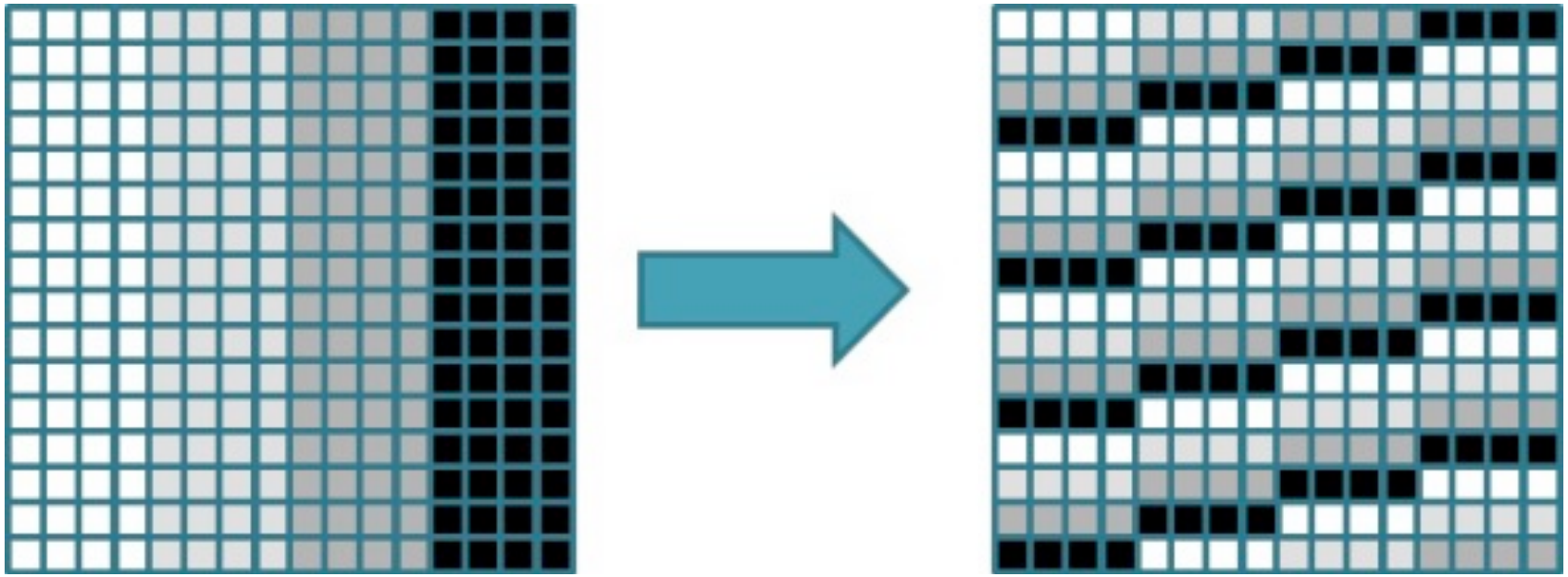
Work mapping: static work allocation

- **Block-Cyclic Allocation:**
  - **Block-cyclic allocation (4 PEs example)**



Work mapping: static work allocation

- **Block-Cyclic Allocation:**
  - Block-cyclic allocation (4 PEs example)
  - better locality, better balance, allocation not regular



Work mapping: static work allocation

Example: n = 12 components of a vector.

PE	Components											
	Block				Cyclic				Block-Cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Work mapping: static work allocation

What did we learn?

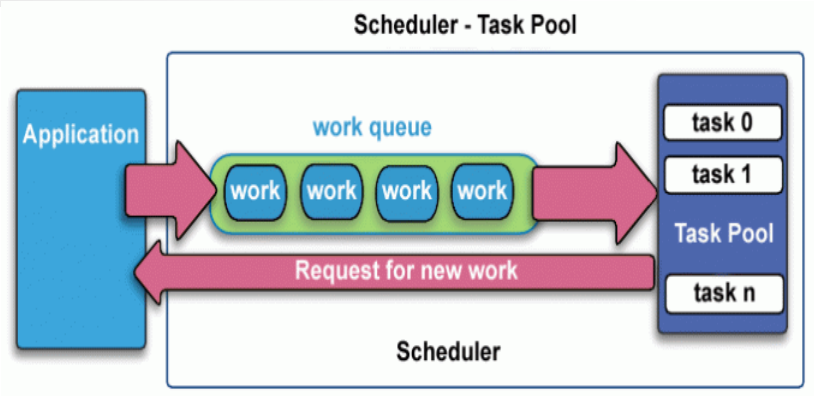
- Need to know size of data
- Block allocation improves data locality
- Knowing access pattern of program can help reduce overheads
- Knowing amount of work for each data element can help improve load balance: cyclic allocation
- Block and cyclic allocation are general techniques. For each program you develop, you may need to tune them

Work mapping: dynamic work allocation

- Allocation of work at *runtime* based on the **current system state**, **workload**, **availability of PEs**.
- Why?
  - **Size of data unknown** (may even be created at runtime).
  - AND/OR **amount of work for each data element is unknown**.
- Also called: **dynamic load balancing**, since balancing the load is the primary motivation for dynamic mapping.
- Main technique
  - work queues (FIFO)
  - work queues (can be **centralized** or **distributed**).

Work mapping: dynamic work allocation

- **Work Queue** holds a list of jobs.
- Each PE:
  - Takes a work from the work queue.
  - Finishes the work.
  - Takes another work from the work queue.
- **Important issue:**
  - synchronization and communication.
  - who initiates work transfer?
  - how much data is transferred?
  - When does program exit?
  - Need to write code carefully to check exit conditions are met.



Work mapping: dynamic work allocation

Centralized work queue:

- **PEs:**
  - Master: mange a group of available tasks.
  - Slave: depend on master to obtain work (push (active) or steal (passive)).
- **Idea:**
  - When a slave PE has no work, it takes a portion of available work from master.
  - When a new task is generated, it is added to the pool of tasks in the master PE.
- **Potential problem**:
  - Important issue is 'size of work' (granularity-Soft & Hard). If too small, then too much overhead communicating getting work from queue.
  - When many PEs are used, master PE may become bottleneck.
  - Maintain locality .
- **Solution:** do a chunk scheduling (increase granularity): every time a PE runs out of work it gets a group of tasks. or use **distributed work queues**.

Work mapping: dynamic work allocation

- **Multiple (distributed) work queue:**
  - Often preferred solution
  - **Reduce synchronization overhead**
    - PEs don't have to synchronize on single global work queue.
  - **Reduce communication overhead**
    - Keep queue in local memory
  - **How many work queues?**
    - Usually, one per PE.

Work mapping: dynamic work allocation

- **Multiple (distributed) work queue:**
  - What if local queue is empty?
  1. One solution is **work stealing** (passive load distribution).
     - Steal work from another queue
     - Increases synchronization and communication overhead but reduces load imbalance.
  2. Another solution is **work pushing** (active load distribution).
     - A PE with lots of work in its queue actively push work to the queue of other PEs.
     - Might affect data locality .. how?

Dependences

- **Dependences** limit parallelism
- Exists between program statements when the order of statement execution affects the results of the program.
- S1 and S2 can execute in parallel
  - if there are no dependences between S1 and S2.

```
a = 1;
b = 2;
```

- Statements can be executed in parallel.

```
a = 1;
b = a;
```

- Statements cannot be executed in parallel

```
b = a;
a = 1;
```

- Statements cannot be executed in parallel.

```
a = 1;
a = 2;
```

- Statements cannot be executed in parallel.

Dependences

- Avoid them in your parallel program
- Example: Ordering on a pair of instructions:

```
x = 5;
y = y + x;
```

- Distributed memory machine - PE 2 must obtain the value of x from PE 1 after PE 1 finishes its computation.
- Shared memory architecture - PE 2 must read x after PE 1 updates it.
- Three types of dependences:
  1. **True (flow) dependence –RAW**
  2. **Anti-dependence –WAR**
  3. **Output dependence –WAW**

True (flow) dependence –RAW

- **Flow dependency**, also known as a **data dependency** or **true dependency** or **read-after-write.**
- **Statements S1, S2. S2 has a true dependence on S1 if S2 reads a value written by S1.**
- It occurs when an instruction depends on the result of a previous instruction

```
x = 10;
y = x * 2;
```

  - in this example, there is a RAW dependency between the second instruction (y = x * 2) and the first instruction (x = 10) because 'y' depends on the value produced by 'x.'
  - Since instruction 2 is truly dependent upon instruction 1.
- **Ordering must be preserved.**
- **Can not be avoided.**

Anti dependence – WAR

- **Anti dependency**, also known as a **write-after-read.**
- **Statements S1, S2. S2 has an anti-dependence on S1 if S2 writes a value read by S1.**
- occurs when an instruction writes to a data item that is later read by another instruction.

```
sum = a+1;
first_term = sum * scale1;
sum = b+1;
second_term = sum * scale2;
```

- Between lines 2 and 3 on sum.
- Prevents lines 1 and 2 from executing in parallel with lines 3 and 4, due to memory reuse
- **Ordering can be avoided/removed.**
  - Rename sum to, e.g., first_sum and second_sum.
  - Note that this increases memory usage overhead.

Anti dependence – WAR

- **Another Example:**

```
int x = 10;
int y = x * 2;
x = 5;
```

- There is a WAR dependency between the third instruction (x = 5) and the second instruction (y = x * 2) because 'x' is overwritten before 'y' is read.

Output dependence –WAW

- **Output dependency**, also known as a **write-after-write.**
- **Statements S1, S2. S2 has an output dependence on S1 if S2 writes a variable written by S1.**
- Happens when two instructions write to the same data item, and the order of these writes' matters.

```
int x = 10;
x = 5;
```

- There is a WAW dependence between the first and second instructions because they both write to 'x,' and the order of these writes affects the final value of 'x.'

Dependences – More Examples

- Parallelism often occurs in loops.

```
for(i=0; i<100; i++)
    a[i] = i;
```

- No dependences.
- Iterations can be executed in parallel.

```
for(i=0; i<100; i++) {
   a[i] = i;
   b[i] = 2*i;
}
```

Iterations and statements can be executed in parallel.

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Iterations and loops can be executed in parallel.

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

- There is a dependence … on itself!
- Loop is still parallelizable.

```
for( i=0; i<100; i++ )
    a[i] = f(a[i-1]);
```

- Dependence between a[i] and a[i-1].
- Loop iterations are not parallelizable.

Dependences – loop-carried dependence

- A loop-carried dependence is a dependence that is present only if the statements occur in two different instances of a loop.
- Otherwise, we call it a loop-independent dependence.
- Loop-carried dependences limit loop iteration parallelization

```
for(i=0; i<100; i++ )
  for(j=0; j<100; j++ )
    a[i][j] = f(a[i][j-1]);
```

- Loop-independent dependence on i.
- Loop-carried dependence on j.
- Outer loop can be parallelized, inner loop cannot.

```
for( j=0; j<100; j++ )
  for( i=0; i<100; i++ )
    a[i][j] = f(a[i][j-1]);
```

- Inner loop can be parallelized, outer loop cannot.
- Less desirable situation (finer-grain parallelism).
- Loop interchange is sometimes possible.