L9 – PARALLEL ALGORITHM

BMCS3003 DISTRIBUTED SYSTEMS AND PARALLEL COMPUTING

CONTENTS

FUNDAMENTALS OF PARALLEL COMPUTING

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time (strong scaling), and/or
  - To solve bigger problems (weak scaling), and/or
  - To achieve better solutions (advancing science)

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

DATA SHARING

- Data sharing can be a double-edged sword
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage and datapaths
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization

SYNCHRONIZATION

- Synchronization == Control Sharing
- Barriers make threads wait until all threads catch up
- Waiting is lost opportunity for work
- Atomic operations may reduce waiting
  - Watch out for serialization

- Important: be aware of which items of work are truly independent

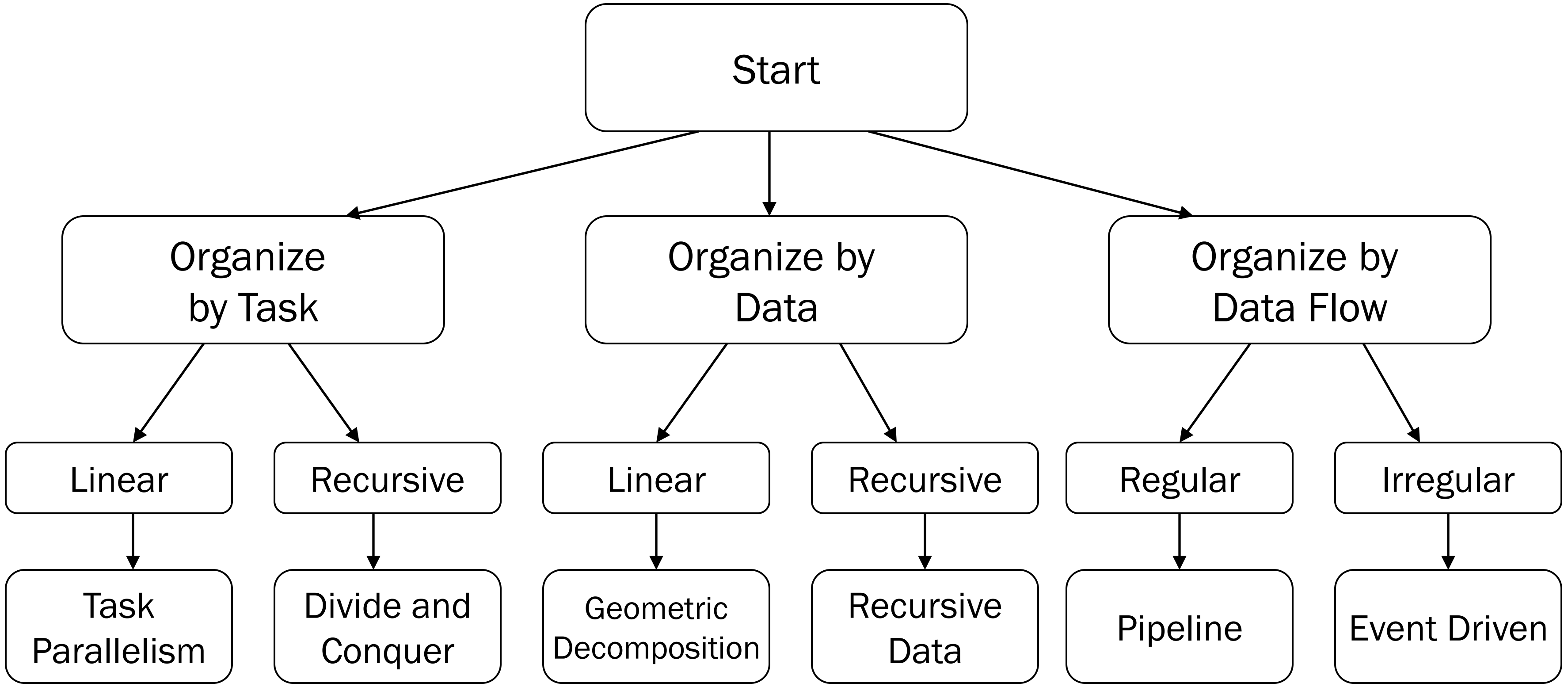
PARALLEL ALGORITHMS

Tasks and Decomposition

Processes and Mapping

Processes Versus Processors

ALGORITHM STRUCTURE

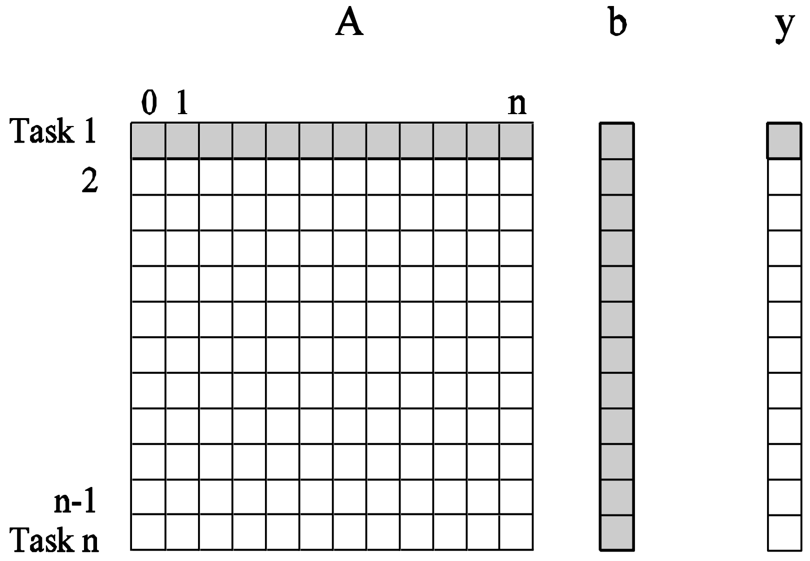


Mattson, Sanders, Massingill, *Patterns for Parallel Programming*

PRELIMINARIES: DECOMPOSITION, TASKS, AND DEPENDENCY GRAPHS

- 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 intermediate sizes.
- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next. Such a graph is called a *task dependency graph*.

EXAMPLE: MULTIPLYING A DENSE MATRIX WITH A VECTOR



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: While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other. 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?*

EXAMPLE: DATABASE QUERY PROCESSING

Consider the execution of the query:

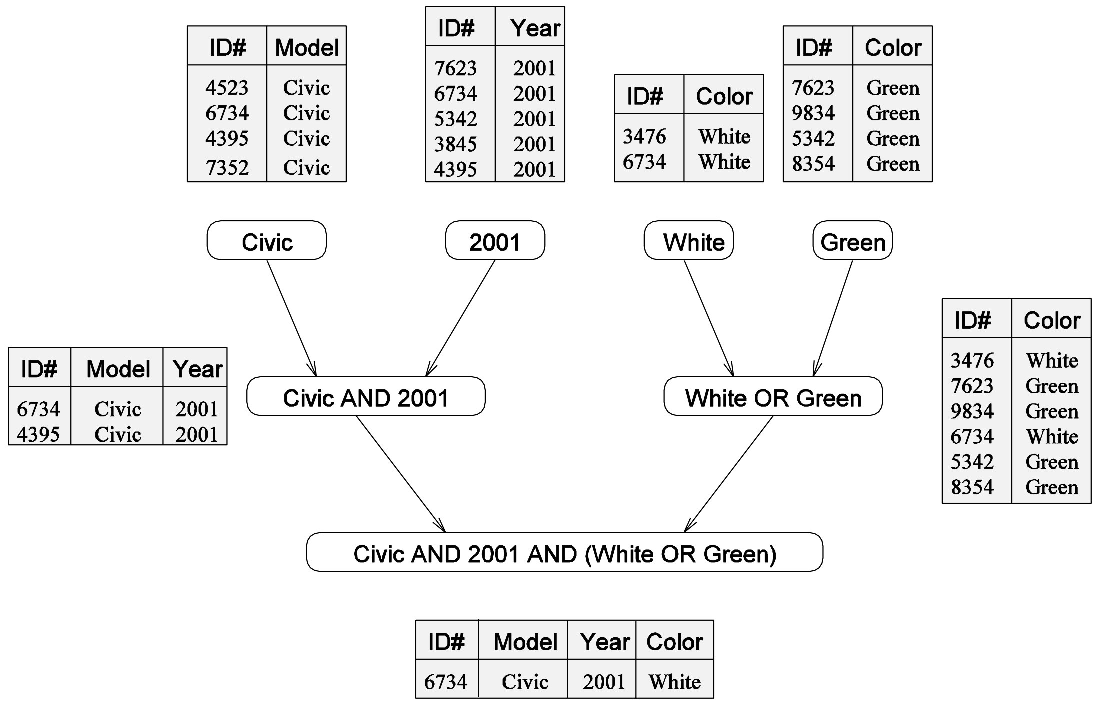
MODEL = ``CIVIC" AND YEAR = 2001 AND
(COLOR = ``GREEN" OR COLOR = ``WHITE)

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

EXAMPLE: DATABASE QUERY PROCESSING

The execution of the query can be divided into subtasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

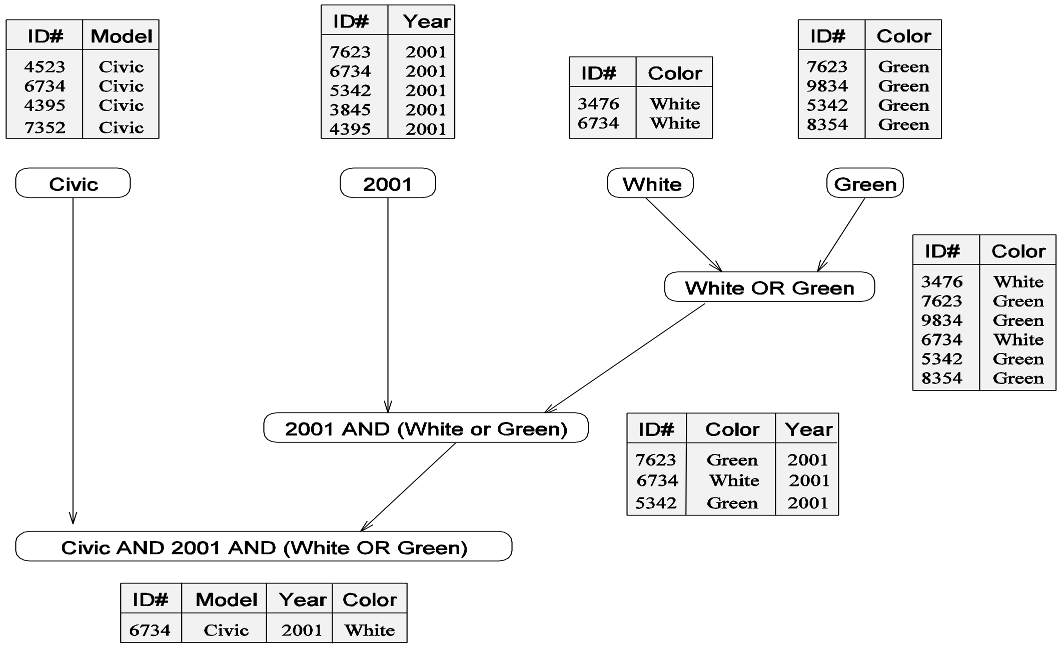
ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

Decomposing the given query into a number of tasks. Edges in this graph denote that the output of one task is needed to accomplish the next.

EXAMPLE: DATABASE QUERY PROCESSING

Note that the same problem can be decomposed into subtasks in other ways as well.


Civic
2001
White
Green
White OR Green
2001 AND (White or Green)
Civic AND 2001 AND (White OR Green)

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

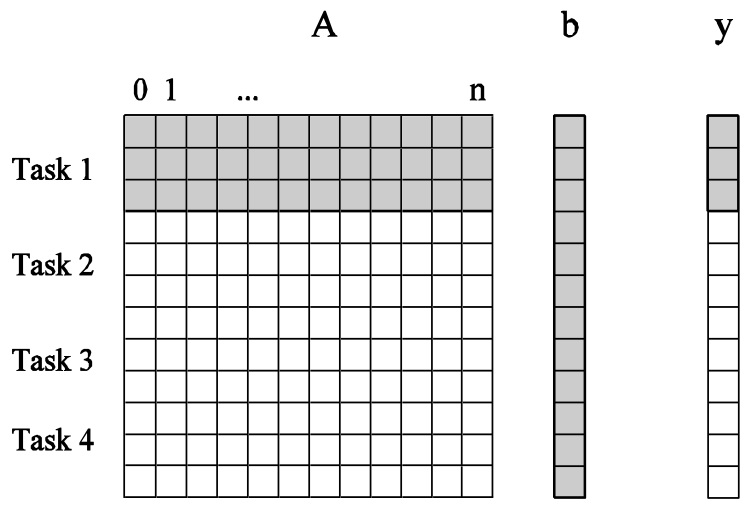
ID#	Model	Year	Color
6734	Civic	2001	White

An alternate decomposition of the given problem into subtasks, along with their data dependencies.

Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

GRANULARITY OF TASK DECOMPOSITIONS

- The number of tasks into which a problem is decomposed determines its granularity.
- Decomposition into a large number of tasks results in fine-grained decomposition and that into a small number of tasks results in a coarse grained decomposition.



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.

DEGREE OF CONCURRENCY

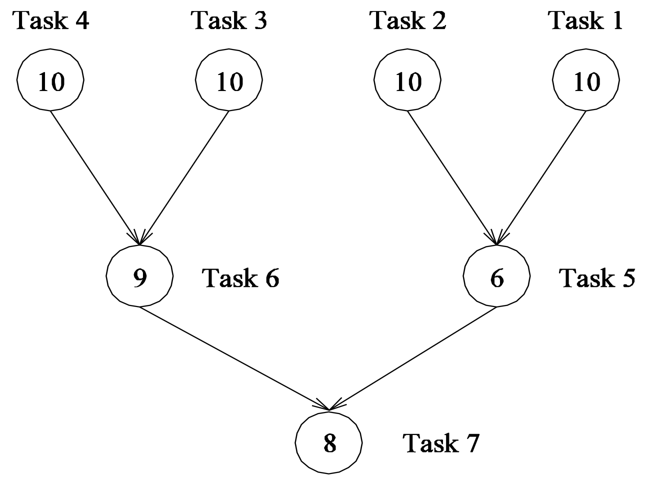
- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution. *What is the maximum degree of concurrency of the database query examples?*
- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program. *Assuming that each tasks in the database example takes identical processing time, what is the average degree of concurrency in each decomposition?*
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

CRITICAL PATH LENGTH

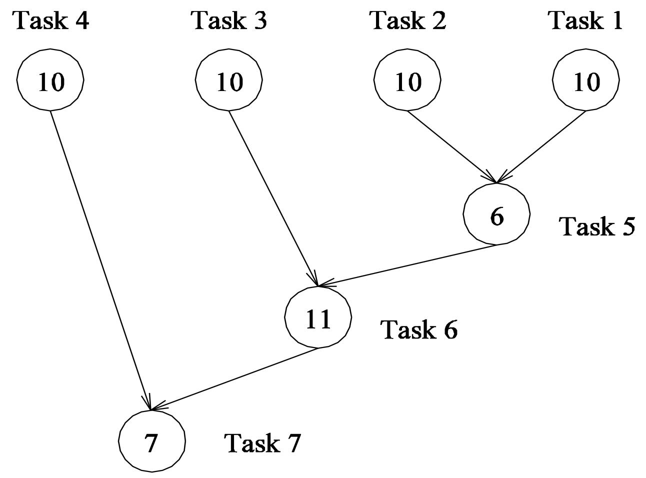
- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.
- The longest such path determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the critical path length.

CRITICAL PATH LENGTH

Consider the task dependency graphs of the two database query decompositions:



(a)



(b)

What are the critical path lengths for the two task dependency graphs? If each task takes 10 time units, what is the shortest parallel execution time for each decomposition? How many processors are needed in each case to achieve this minimum parallel execution time? What is the maximum degree of concurrency?

LIMITS ON PARALLEL PERFORMANCE

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.
- There is an inherent bound on how fine the granularity of a computation can be. *For example, in the case of multiplying a dense matrix with a vector, there can be no more than* ***(n^2)*** *concurrent tasks.*
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.

PROCESSES AND MAPPING

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.

- For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

Note: We refer to the mapping as being from tasks to processes, as opposed to processors. This is because typical programming APIs, as we shall see, do not allow easy binding of tasks to physical processors. Rather, we aggregate tasks into processes and rely on the system to map these processes to physical processors. We use processes, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.

PROCESSES AND MAPPING

- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.
- Mappings are determined by both the task dependency and task interaction graphs.
- Task dependency graphs can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).
- Task interaction graphs can be used to make sure that processes need minimum interaction with other processes (minimum communication).

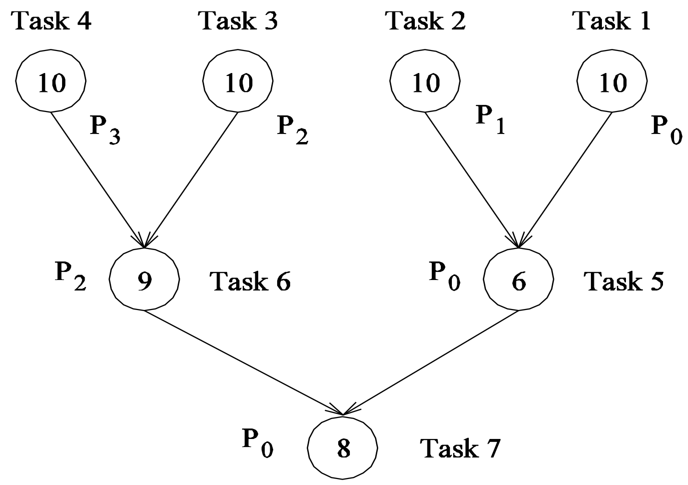
PROCESSES AND MAPPING

An appropriate mapping must minimize parallel execution time by:

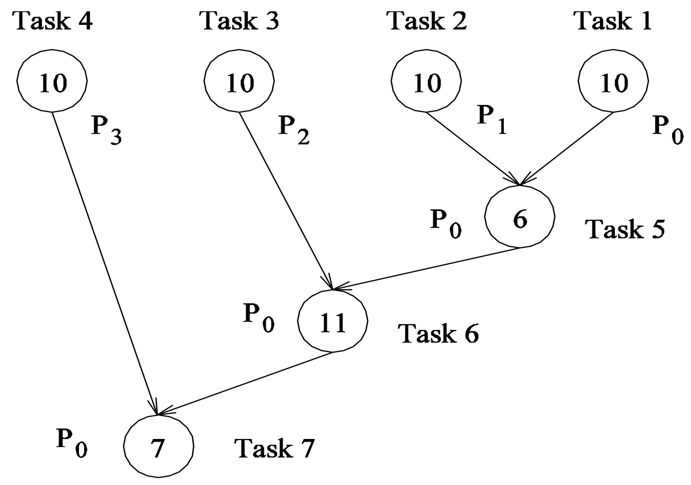
- Mapping independent tasks to different processes.
- Assigning tasks on critical path to processes as soon as they become available.
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

Note: These criteria often conflict with each other. For example, 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?

PROCESSES AND MAPPING: EXAMPLE



(a)

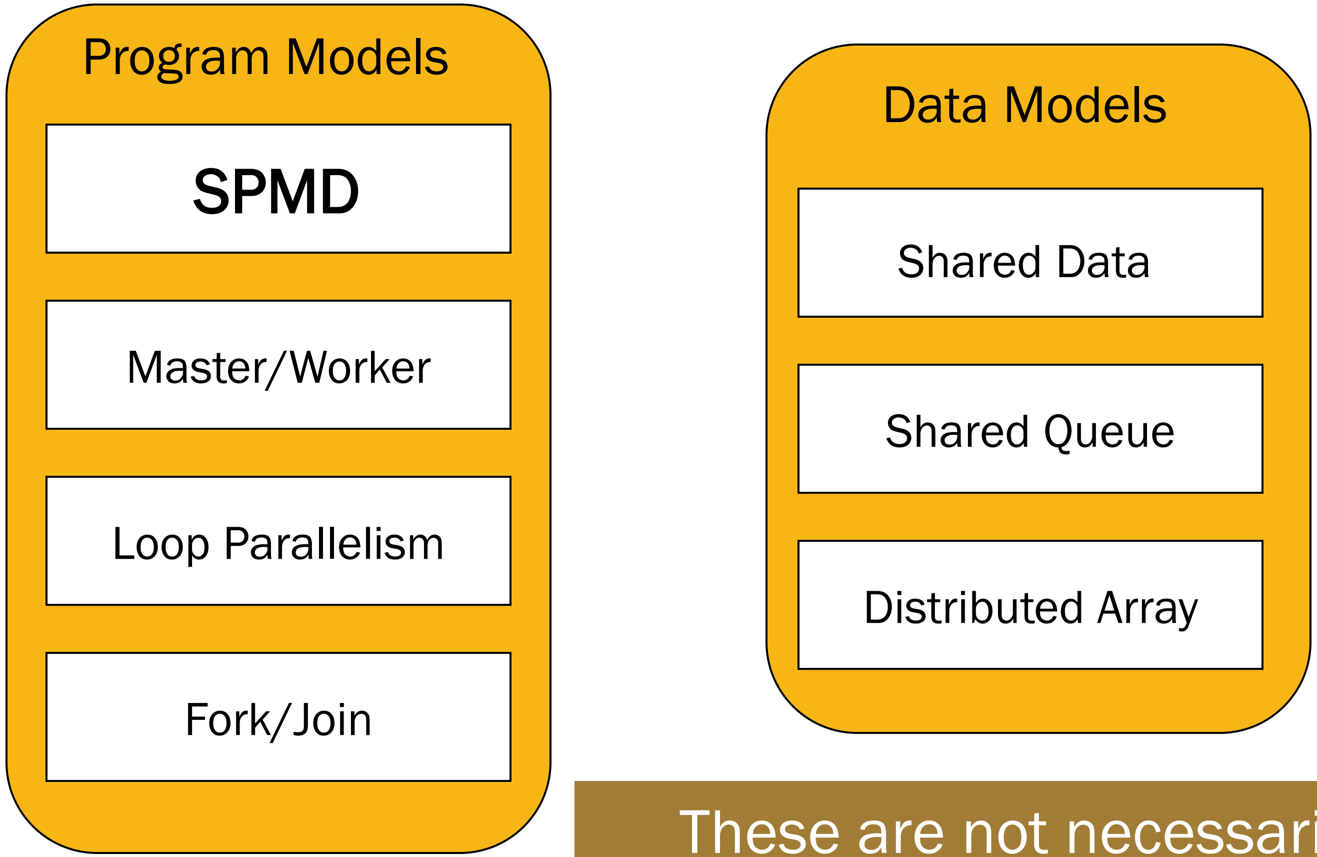


(b)

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.

MODELS OF PARALLEL ALGORITHM

PARALLEL PROGRAMMING CODING STYLES – PROGRAM AND DATA MODELS



These are not necessarily mutually exclusive.

PROGRAM MODELS

- SPMD (Single Program, Multiple Data)
  - All PE's (Processor Elements) execute the same program in parallel, but has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code
  - This is essentially the CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- Master/Worker
- Loop Parallelism
- Fork/Join

PROGRAM MODELS

- SPMD (Single Program, Multiple Data)
- Master/Worker (OpenMP, OpenACC, TBB)
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
- Loop Parallelism (OpenMP, OpenACC, C++AMP)
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
- Fork/Join (Posix p-threads)
  - Most general, generic way of creation of threads

MORE ON SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD (next to loop parallelism)
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.
- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

TYPICAL SPMD PROGRAM PHASES

- Initialize
  - Establish localized data structure and communication channels
- Obtain a unique identifier
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- Distribute Data
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- Run the core computation
  - More details in next slide…
- Finalize
  - Reconcile global data structure, prepare for the next major iteration

CORE COMPUTATION PHASE

- Thread IDs are used to differentiate behavior of threads
  - Use thread ID in loop index calculations to split loop iterations among threads
    - Potential for memory/data divergence
  - Use thread ID or conditions based on thread ID to branch to their specific actions
    - Potential for instruction/execution divergence

Both can have very different performance results and code complexity depending on the way they are done.

MAKING SCIENCE BETTER, NOT JUST FASTER

OR... IN OTHER WORDS:
THERE WILL BE NO NOBEL PRIZES OR TURING AWARDS AWARDED FOR “JUST RECOMPILE” OR USING MORE THREADS

TERMS AND NOTATIONS

Sequential Runtime	T_1
Sequential Fraction	s
Parallel Fraction	$p = 1 - s$
Parallel Runtime	T_N
Cost	$C_N = NT_N$
Parallel Overhead	$T_o = C_N - T_1$
Speedup	$S_N = T_1 / T_N$
Efficiency	$E = S_N / N$

FACTORS AFFECTING SPEEDUP

- Sequential Fraction
- Parallel Overhead
  - Unecessary/duplicate work
  - Communication overhead/idle time
  - Time to split/combine
- Task Granularity
- Degree of Concurrency
- Sychronization/Data Dependency
- Work Distribution
- Ramp-up and Ramp-down Time

AMDAHL'S LAW

- Speedup is bounded by

$$(s + p)/(s + p/N) = 1/(s + p/N) = N/(sN + p)$$

- This means more processors $\Rightarrow$ less efficient!
- How do we combat this?
- Typically, larger problem size $\Rightarrow$ more efficient.
- This can be used to "overcome" Amdahl's Law.

SPEEDUP

$$\text{Speedup} = \frac{\text{Execution time before improvement}}{\text{Execution time after improvement}}$$

- There are three types of problems to be solved using the following Amdahl's Law equation:

$$\text{Speedup} = \frac{1}{(1 - \text{fraction enhanced}) + (\text{fraction enhanced}/\text{factor of improvement})}$$

- Let Speedup be denoted by "S", fraction enhanced be denoted by "f_E", and factor of improvement be denoted by "f_I". Then we can write the above equation as
- $S = ((1 - f_E) + (f_E \ / \ f_I))^{-1}$.
- The three problem types are as follows:
- 1. Determine S given f_E and f_I
- 2. Determine f_I given S and f_E
- 3. Determine f_E given S and f_I

PROBLEM TYPE 1 – PREDICT SYSTEM SPEEDUP

- If we know f_E and f_I, then we use the Speedup equation (previous slide) to determine S.
- **Example:** Let a program have 40 percent of its code enhanced (so f_E = 0.4) to run 2.3 times faster (so f_I = 2.3). What is the overall system speedup S?
  - *Step 1*: Setup the equation: $S = ((1 - f_E) + (f_E / f_I))^{-1}$
  - *Step 2*: Plug in values & solve $S = ((1 - 0.4) + (0.4 / 2.3))^{-1}$
    - $= (0.6 + 0.174)^{-1} = 1 / 0.774$
    - = **1.292**

PROBLEM TYPE 2 – PREDICT SPEEDUP OF FRACTION ENHANCED

- If we know f_E and S, then we solve the Speedup equation (previous slide) to determine f_I , as follows:
- Example: Let a program have 40 percent of its code enhanced (so f_E = 0.4) to yield a system speedup
- 4.3 times faster (so S = 4.3). What is the factor of improvement f_I of the portion enhanced?

Case #1:

- Can we do this? In other words, let's determine if by enhancing 40 percent of the system, it is possible to make the system go 4.3 times faster …
- Step 1: Assume the limit, where f_I = infinity, so $S = ((1 - f_E) + (f_E / f_I))^{-1} \rightarrow S = 1 / (1 - f_E)$
- Step 2: Plug in values & solve $S = ((1 - 0.4))^{-1} = 1 / 0.6 = 1.67$.
- Step 3: So S = 1.67 is the maximum possible speedup, and we cannot achieve S = 4.3 !!

PROBLEM TYPE 2 – PREDICT SPEEDUP OF FRACTION ENHANCED (CONTINUE)

Case #2:

A different case: Let’s determine if by enhancing 40 percent of the system, it is possible to make the system go 1.3 times faster …

- Step 1: Assume the limit, where f_I = infinity, so S = ((1 – f_E) + (f_E / f_I))$^{-1}$ → S = 1 / (1 – f_E)
- Step 2: Plug in values & solve S = ((1 – 0.4))$^{-1}$ = 1 / 0.6 = 1.67 .
- Step 3: So S = 1.67 is the maximum possible speedup, and we can achieve S = 1.3 !!
- Step 4: Solve speedup equation for f_I :

1/S = (1 – f_E) + (f_E / f_I)	[invert both sides]
1/S – (1 –f_E) = f_E / f_I	[subtract (1 – f_E)]
(1/S – (1 – f_E))$^{-1}$ = f_I / f_E	[invert both sides]
f_E · (1/S – (1 – f_E))$^{-1}$ = f_I	[multiply by f_E]

- Step 5: Plug in values & solve: f_I = f_E · (1/S – (1 – f_E))$^{-1}$

  = 0.4 · (1/1.3 – (1 – 0.4))$^{-1}$

  = 0.4 / (0.769 – 0.6) = 2.367

- Step 6: Check your work: S = ((1 – f_E) + (f_E / f_I))-1 = (0.6 + (0.4/2.367))-1 = 1.3 (Correct)

PROBLEM TYPE 3 – PREDICT FRACTION OF SYSTEM TO BE ENHANCED

- If we know f_I and S, then we solve the Speedup equation (previous slide) to determine f_E , as follows:
- Example: Let a program have a portion f_E of its code enhanced to run 4 times faster (so f_I = 4), to yield a system speedup 3.3 times faster (so S = 3.3). What is the fraction enhanced (f_I)?
- Step 1: Can this be done? Assuming f_I = infinity, S = 3.3 = ((1 – f_E))$^{-1}$ so minimum f_E = 0.697 Yes, this can be done for maximum f_I, so let's solve the equation to determine actual f_E
- Step 2: Solve speedup equation for f_E :

	S = ((1 – f_E) + (f_E / f_I))$^{-1}$	[state the equation]
	3.3 = ((1 – f_E) + (f_E / 4))$^{-1}$	[plug in values]
	(1 – f_E) + f_E /4= 1/3.3 = 0.303	[invert both sides]
	1 – 0.75 f_E = 0.303	[regroup]
	0.75 f_E = 1 – 0.303 = 0.697	[commutativity]
	f_E = 0.697 / 0.75 = 0.929	[divide by 0.75]

- Step 3: Check your work: S = ((1 –f_E) + (f_E / f_I))$^{-1}$ = (0.071 + (0.929/4))$^{-1}$ = 3.3 (Correct)

GUSTAFSON'S VIEWPOINT

- Gustafson noted that typically the serial fraction does not increase with problem size.
- This view leads to an alternative bound on speedup called scaled speedup.

$$(s + pN)/(s + p) = s + pN = N + (1-N)s$$

This may be a more realistic viewpoint.

EXAMPLE

HISTOGRAM

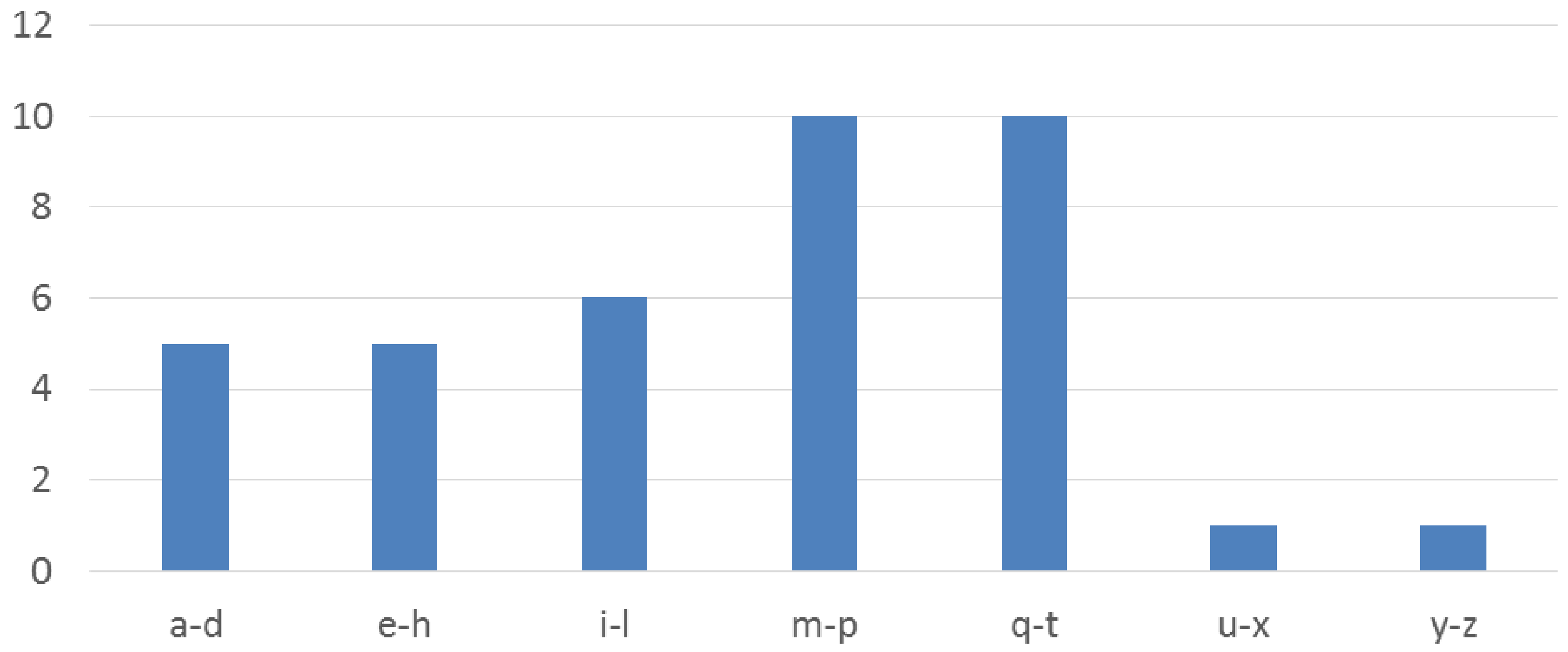
HISTOGRAM

- A method for extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …

- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment

A TEXT HISTOGRAM EXAMPLE

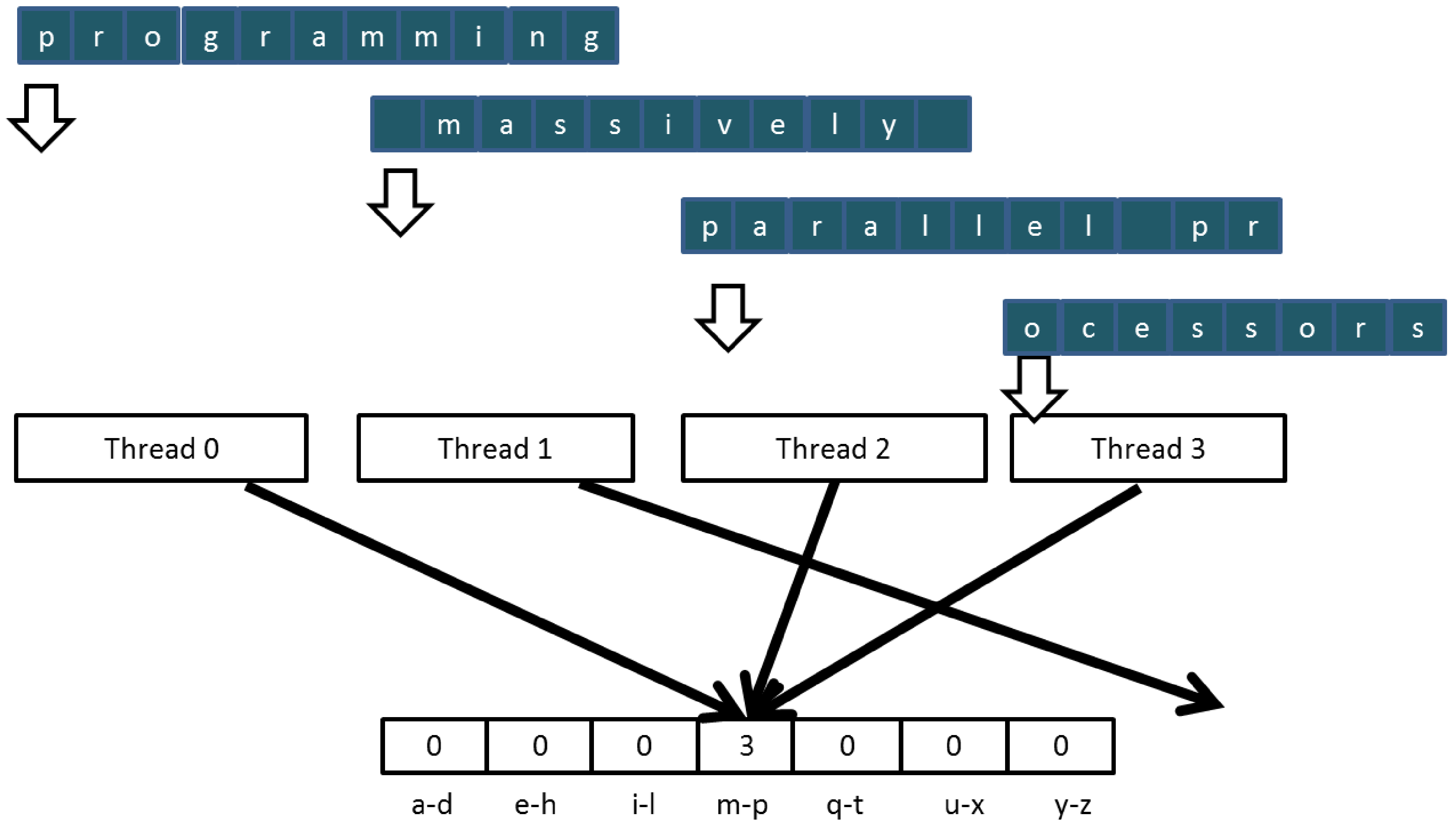
– Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, …
– For each character in an input string, increment the appropriate bin counter.
– In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



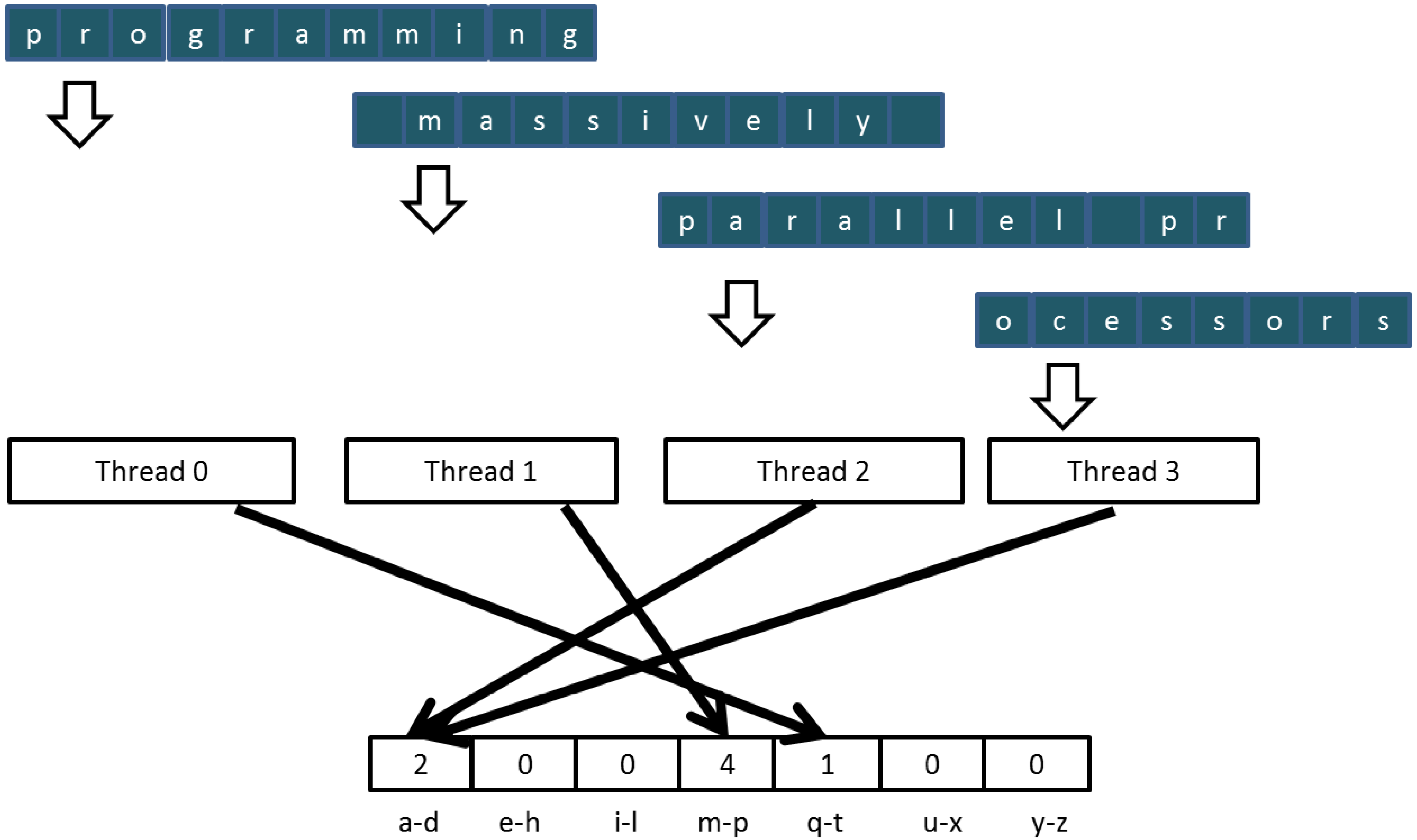
A SIMPLE PARALLEL HISTOGRAM ALGORITHM

- Partition the input into sections
- Have each thread to take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter

SECTIONED PARTITIONING (ITERATION #1)

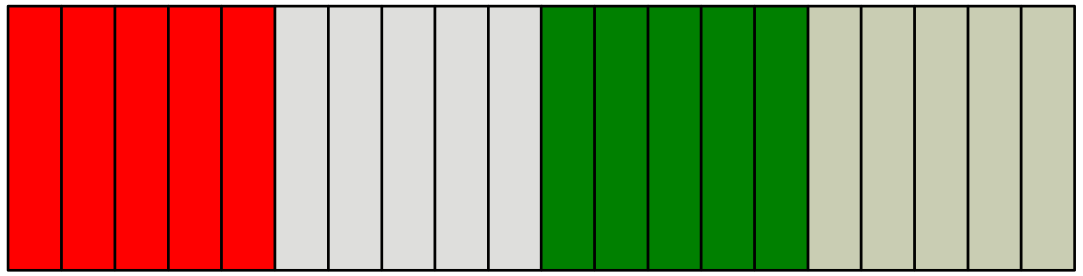


SECTIONED PARTITIONING (ITERATION #2)



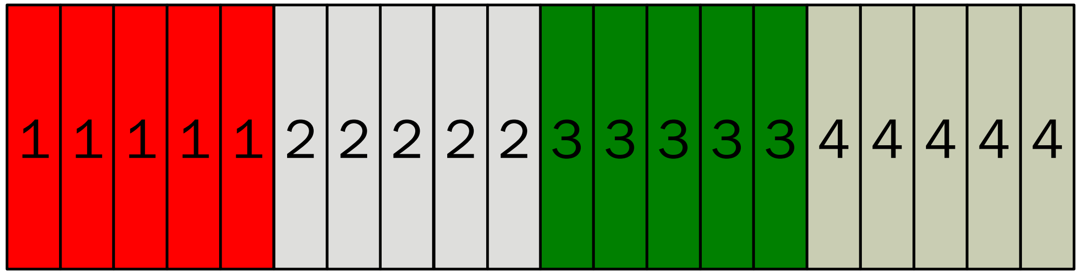
INPUT PARTITIONING AFFECTS MEMORY ACCESS EFFICIENCY

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

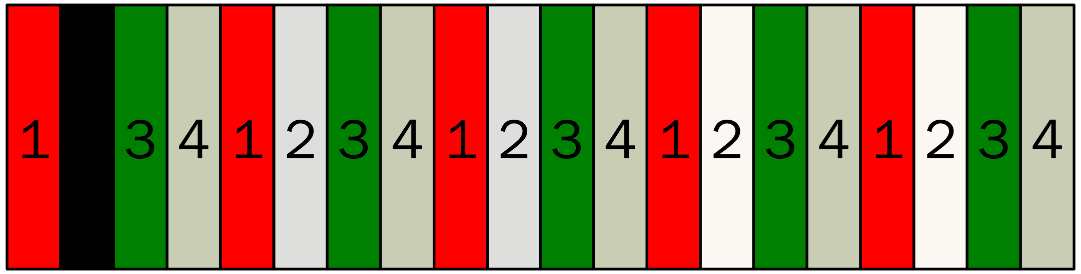


INPUT PARTITIONING AFFECTS MEMORY ACCESS EFFICIENCY

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized

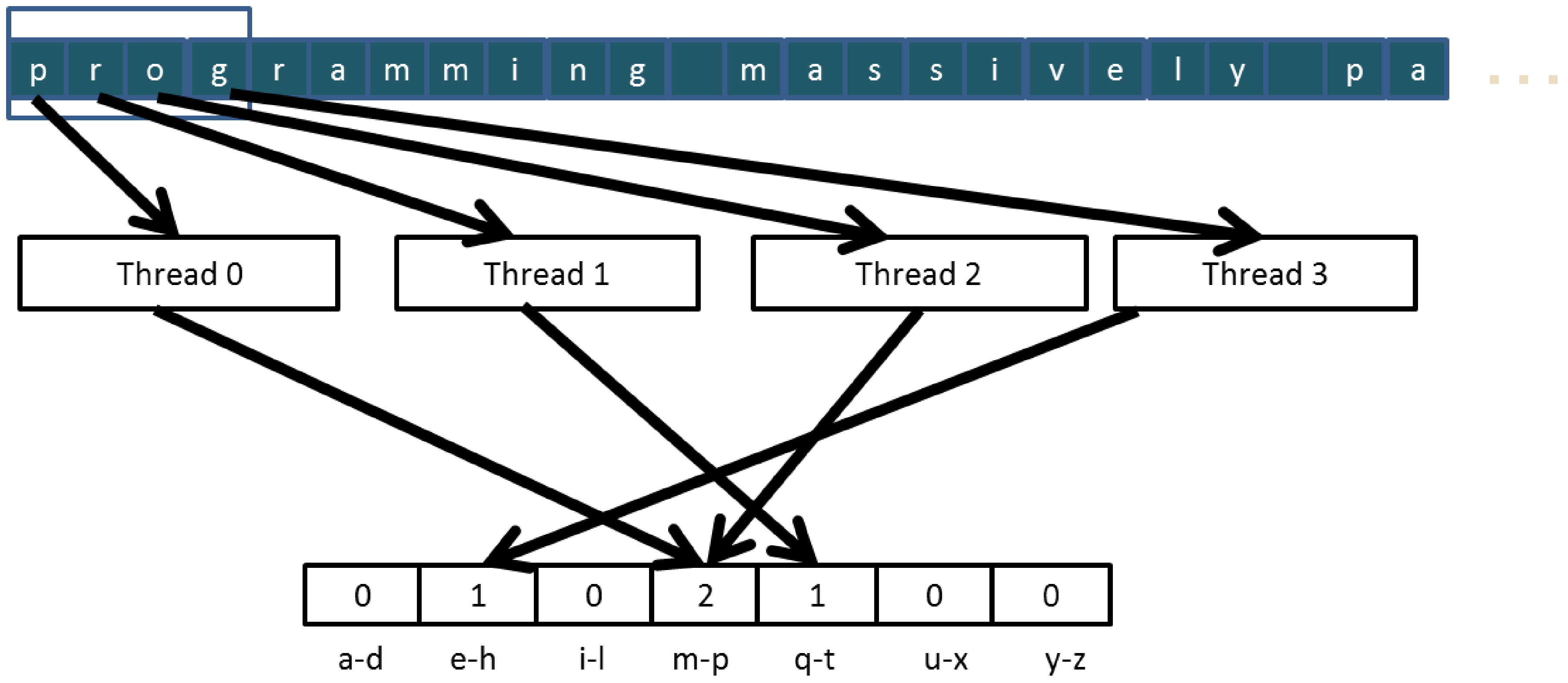


- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced

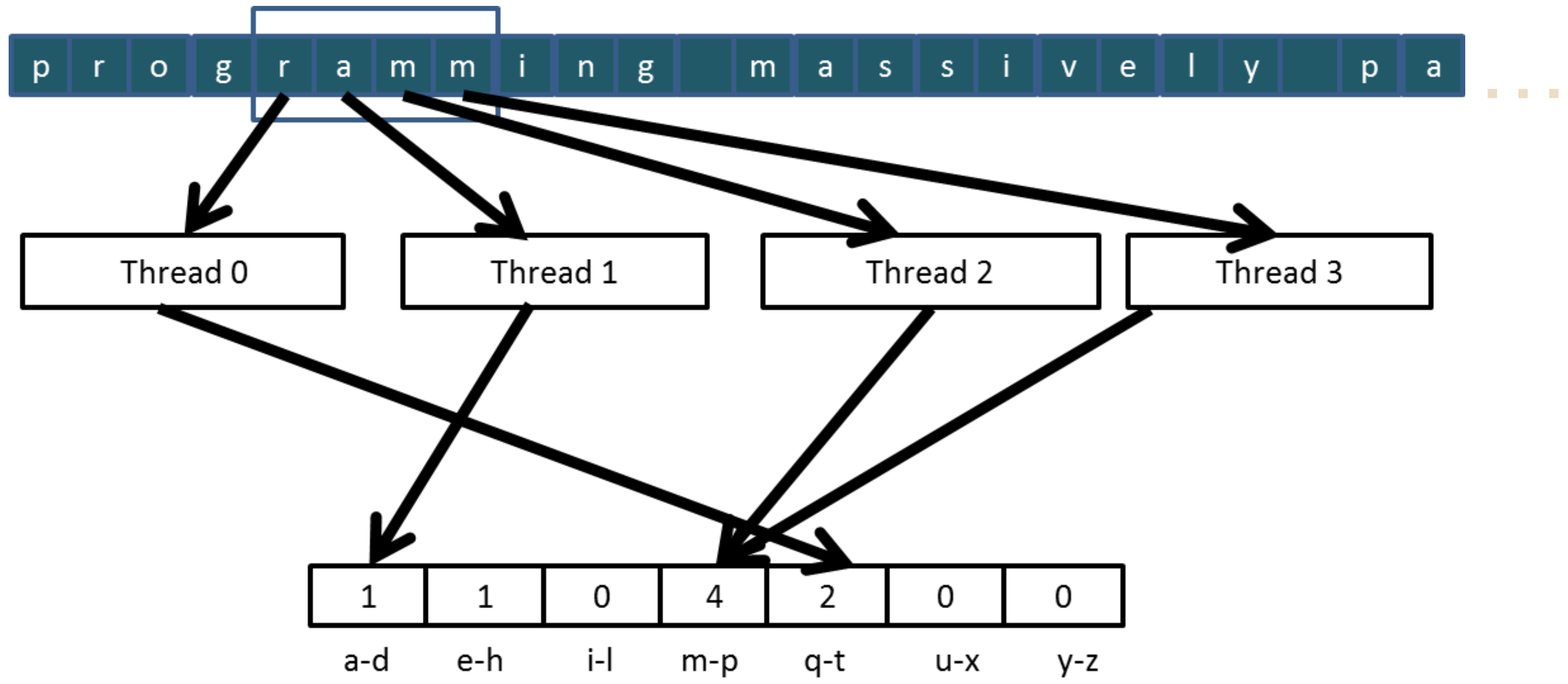


INTERLEAVED PARTITIONING OF INPUT

– For coalescing and better memory access performance



INTERLEAVED PARTITIONING (ITERATION 2)


p r o g r a m m i n g m a s s i v e l y p a
Thread 0
Thread 1
Thread 2
Thread 3
1 1 0 4 2 0 0
a-d
e-h
i-l
m-p
q-t
u-x
y-z