Questions 2023

1) Amdahl's Law: what is it and how can you derivate the fomula

Amdahal Law is a crucial principle in parallel theory used to make predictions about the theoretical speedup obtainable when multiple processing unit are used.

It states that if a program has a portion f of non parallelizable work then the speedup is limited to $\frac{1}{f}$, no matter how many processing unit you have at your disposal.

$T_{seq} = fT_{seq} + (1-f)T_{seq}$

To prove it we assume, that a program is divided in a portion f of non parallelizable work and the remaining portion $1-f$ of parallelizable works.
Hence the best theoretical time for the parallelizable part is:

$$\frac{(1-f)T_{seq}}{n}$$

Then we have that:

$$speedup(n) = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{f \cdot T_{seq} + (1-f)\frac{T_{seq}}{n}} = \frac{T_{seq}}{T_{seq}(f + \frac{1-f}{n})} = \frac{1}{f + \frac{1-f}{n}}$$

And we obtain that:

$$\lim_{n\to\infty} \frac{1}{f + \frac{1-f}{n}} = \frac{1}{f}$$

Which means that, with **the processing unit tending to infinite, the speedup is limited by the sequential portion of the work** □

2) Gustafsson's Law: what is it and how can you derivate the formula

Gustafson Law is another crucial principle of parallel theory and it addresses the shortcomings of Amdahal's law.

Amdahal's law assume that the problem size remains constant, while **Gustafson recognize that programmers tends to exploit the resources they have by also increasing the workload/size of the problem.**

Say that we have:

- n workers
- f non-parallelizable work

- $1 - f$ parallelizable work

Gustafsson found that the speedup gained using parallelism is

$$speedup(n) = f + n(1 - f) = f + n - nf = n - f(n - 1)$$

Basically **Gustafsson propose that the size of the problem grows proportionally to the number of processing units: in the same time we can solve a bigger instance of the problem.**

3) What is the Actor Model?

The actor model is model of concurrent computation that treats an actor as the basic building block of concurrent computation.

The actor model adopts the philosophy that everything is an actor.
This is similar to the everything is an object philosophy used by some object-oriented programming languages.

In response to a message it receives, an actor can:

- make local decisions
- create more actors
- send more messages
- determine how to respond to the next message received.

Actors may modify their own private state, but can only affect each other indirectly through messaging (removing the need for lock-based synchronization)

The actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

The said communication is also **one-way asynchronous**: once a message has been sent, it is the responsibility of the receiver.
Because of the indeterminism, it's important that the actions are **quasi-commutative**: the order in which messages are sent between actors must not affect the final outcome of the system.

Note that due to the said features, the Actor model supports **fault-tolerance** (actors can be restarted/moved to another machine) and **scalability** (new actors can be created).

4) What's Spark? Are there other frameworks? Talk about GraphX

Apache Spark is a unified analytics engine for large-scale data processing.
It is based on the Hadoop Approach: it does support MapReduce, but also SQL queries,

data streaming, machine learning and graph algorithms.

The main component is the driver program, which runs the `main()` and creates the `SparkContext`, which is the entry point to any Spark functionality.
The driver program is responsible for dividing the work into tasks and schedule them across the clusters.

Spark support two kinds of shared variables across tasks:

- **broadcast**: read-only variables that are cached on each machine
- **accumulators**: write-only variables that can only be increased

As said Spark has been influenced by Hadoop, which is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

GraphX

GraphX is a distributed graph processing framework built on top of Spark.
It supports the most common graph operations (vertices/edges counting, reverse, subgraphing, ...) and has built-in algorithms (PageRank, Connected Components, Shortest Path, ...)

GraphX is implemented using the concept **TLAV** (Think Like a Vertex) and it leverages two models:

- Bulk Synchronous Parallel (BSP) Model
- Actor Model

Think Like a Vertex

The vertex is the primary unit of computation, it maintains a state (label, attributes, ...) and it can send messages to its neighbours, which can in turn update their state with the info received with those messages.

Actor Model

See before

Bulk Synchronous Parallel Model

The BSP Model is a parallel computing model that divides the computation into supersteps, where each superstep consists in:

- a computation phase
- a communication phase

- a synchronization phase
  Thanks to the last phase no superstep can overlap.

Mind that BSP model adopt the TLAV philosophy: the compute units are the vertices, they can communicate and exchange messages with each other and synchronize at the end of every superstep.

5) Stream and Data Parallelism

We can outline two approaches to parallelism:
Data Parallelism: parallelism resulting from the division of an initial task in subtask, each subtask is then computed in parallel and the results are combined to obtain the result of the original task.
It is effective to reduce the latency of a program.

Stream Parallelism: parallelism resulting from the parallel execution of different tasks available at different times on an input stream.
It is effective to increase throughput.

Two-Tier Composition Model
It has been observed that it has better performance to apply stream parallelism "before" the data parallelism: the data parallelism lives on top the stream parallelism.

6) Collective Communication in MPI

In MPI, collective communication is a method of communication that involves the participation of (all) processes in a communicator.
Collective communication routines provide a higher-level abstraction for communication and synchronization among processes.
Collective communication calls may use the same communicators as point-to-point communication, but MPI guarantees that messages generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication.
Collective communication routines are blocking, meaning that they do not return until all processes in the communicator have completed the operation.

We notice the calls:

- `MPI_BCast(...)`: broadcast
- `MPI_Barrier(MPI_Comm comm)`: synchronize all the processes inside a communicator
- ...

In **Hadoop**, collective communication refers to the ability of multiple nodes in a cluster to communicate and coordinate with each other to perform a computation.
Collective communication is important for many data mining and data analysis algorithms that require iterative computation.

7) Advantages of using Map Fusion

An idea to improve a composition of sequences of map-reduce comes from the theoretical result named map fusion, which it aims to move as less data as possible to give the final result.

The map operation applies a function to every element of a collection of data, producing another collection of data.
If we have a series of map operations we have a sequence of intermediate collections that are formally useless to store.
To reduce this problem we can, if possible, fuse some maps, in order to not store/pass the intermediate collection.

As an example consider the following code:

```
numbers = [1,2,3,4]
result = numbers.map(_ +1).map(_ *2)
```

The consecutive applications of `map` are inefficient and they can be fused to obtain:

```
numbers = [1,2,3,4]
result = numbers.map(x -> (x+1)*2)
```

The advantages are then clear:

- **less space required** (no instance of the intermediate collection is stored)
- **less time used** (there is no waste of time creating and deallocating the intermediate collection)
- **less communication overhead** if the process is parallel and/or distributed (the intermediate collection is never sent to anyone as it never exists)

It is always positive to fuse maps when possible?
No, we can identify cases where having the intermediate collection is beneficial to the computation to the final collection, even with the "overhead" of creating and looping more-than-once through the collection.
Consider an array of elements, we want to apply to every element a number of hash functions.
In this case fusing all the hash function may be more complex than computing them singularly.

8) Scatter and Gather in MPI

- `MPI_Scatter(...)` distribute a message (e.g. an array) among multiple processes

- `MPI_Gather(...)` the inverse of `MPI_Bcast(...)`, a process receives `n` messages from other processes
    - `MPI_Bcast(...)` broadcast message(s) to all the processes

9) What is the Role of the Communicator in MPI

In the Message Passing Interface we have three main entities:

- **Communicator**
- **Group**
- **Process**

The **Communicator** is an object that encompasses a group of processes that have the ability to communicate with one another, achieving point-to-point communication.
A **Group** is a group of processes.
A **Process** has a rank (ID) and they can send/receive messages to/from other processes by providing a rank and a tag to uniquely identify the message.

10) Spark Shared Variables

Spark supports two kinds of shared variables across tasks:

- **broadcast:** Read-only variables that are cached on each machine. Broadcast variables allow programmers to reduce the communication cost, for example they can be used to give a copy of a large input dataset to each node
- **accumulators:** Write-only variables that can only be increased by the tasks. They can be used for reduce operations, so that the driver can retrieve the final result.

11) Map Fusion's relation with Spark's Transformations

A transformation can be applied on a RDD in order to generate a new RDD.
A sequence of transformations behaves exactly as a sequence of mappings, so fusing them can improve performances.

12) Job Stealing

Job stealing is a popular technique of scheduling tasks between processing elements.
Basically a PE can take tasks from other PEs if they don't have anything to do anymore.

To use this approach we need:

1. **a way to assign activities to PEs**
2. **a way to list the PEs that still have something to compute** so that PEs can know who they can steal from

    1. one approach is to have a centralized entity to which PEs can report updates. Easy to implement but it can be a SPOF and be a bottleneck
    2. the best approach is to steal at random: the idle PE will choose randomly which PE will be his victim, and hopefully he will have something to be stolen
3. **a protocol for the stealing**: the victim must handle the stealing request and also be able to handle more PEs trying to steal tasks at the same time

This technique can be very effective but it is difficult to implement as there are many aspects to be defined and handled, and there is the concrete risk to spend too much on the stealing and too low on the executing the tasks.

13) Formulate a Dynamic Scheduling Technique which finds compromises between Job Stealing and Auto Scheduling

Job Stealing: see before

Auto Scheduling: tasks are not assigned to the processing units, but instead the idle processing units will ask for a task to compute to a centralized or distributed concurrent activity repository.
This works great for activities with quite different execution times and/or processing elements with different compute capabilities.
There is the slight hiccup of dependencies that may be handled.

To big of a question and a talkative one, you improvise at the exam.

14) How to use MPI to implement the PageRank?

PageRank is an algorithm used by search engines to rank web pages in their search engine results.

The algorithm assigns a score to each web page based on the number and quality of other web pages that link to it.
The score is used to determine the order in which web pages are displayed in search results.

To implement PageRank using MPI, we can use a distributed memory model where each node in the cluster is responsible for computing the PageRank score for a subset of the web pages.
The nodes communicate with each other to exchange information about the links between web pages and to update the PageRank scores.

The implementation of PageRank using MPI involves several steps.
First, **the web pages are represented as a graph, where each node represents a web page and each edge represents a link between web pages.**
The graph is partitioned into subsets, with each subset assigned to a different node in the cluster.

Next, each node computes the PageRank score for the web pages in its subset. The computation involves iterating over the web pages and updating their PageRank scores based on the scores of the web pages that link to them.

During the computation, **the nodes communicate with each other to exchange information about the links between web pages and to update the PageRank scores.** The communication involves sending and receiving messages between nodes using MPI communication routines.
Finally, after all nodes have completed their computations, the PageRank scores are combined to produce the final ranking of the web pages.

Implementing PageRank using MPI involves partitioning the web pages into subsets, computing the PageRank scores for each subset, communicating between nodes to exchange information and update the scores, and combining the scores to produce the final ranking.

15) What's Macro Data Flow?

Macro Data Flow is a standard to implement parallel programming frameworks. The idea is to denote models where the order of computation is dictated by the data flow, and where we can abstract complex functions.

Rather then compiling skeletons by instantiating the proper process templates the skeletons are first compiled to macro data flow graphs, and then instances of this graph (one instance for each input data set) **are evaluated by a distributed macro data flow interpreter.**

A macro data flow graph is a graph where the nodes contain a function to compute, a list containing inputs and instructions on where to put the output, while an arc denotes that a data item (the output token) is moved from a node to another.
For every input to the program a copy of the program is created and the workers will operate on it in order to run the program.

16) What's Vectorization and What are the 4 Conditions for Loops?

Vectorization is a way to exploit data-level parallelism by performing the same operation on multiple data elements simultaneously.

It is an optimization technique where scalar code is transformed into vectorized code that can take advantage of SIMD (Single Instruction, Multiple Data) instructions.
Many CPUs have "vector" or "SIMD" instruction sets which apply the same operation simultaneously to two, four or more pieces of data.
Vectorization is the process of rewriting a loop so that instead of processing a single element of an array N times the CPU processes (say) 4 elements of the array simultaneously N/4 times.

There are two ways to vectorize a program:

- manual vectorization: explicitly write vector instructions using built-in language constructs
- auto vectorization: the compiler can optimize the code automatically by identifying the loops that can be vectorized.

To be vectorized the loops must satisfy the following requirements:

1. known number of iterations beforehand
2. no external call in the body of the loop
3. no conditional branches
4. no overlapping pointers

17) CAP Theorem and PACELC Theorem

The **CAP Theorem** is a fundamental concept in distributed systems that states that in presence of a network partition a distributed system can only provide either consistency and partition tolerance (CP) or availability and partition tolerance (AP).

Where we define:

- **consistency:** reads are always up to date
- **availability:** always get a response (even if there is no guarantee that it is the most up-to-date value)
- **partition tolerance:** system continues to operate despite network partitions (communication errors) that may occur

The **PACELC Theorem** is an extension of the CAP theorem and stands for **Partition tolerance, Availability, Consistency, Else Latency and Consistency.**
The theorem states that in presence of a network partition a distributed system must choose between availability and consistency (as seen in the CAP theorem), however when the system is running normally, without a network partition, it must choose between latency and consistency.

GitHub (2017 - 2021)

1) Point-to-Point and Collective Communication in FastFlow. When to use collective communication

FastFlow supports both point-to-point and collective communication.
Point-to-point communication involves sending messages between two processes, while collective communication involves a group of processes communicating at the same time.

In FastFlow, point-to-point communication is implemented using message passing, where messages are sent and received using communication channels.

FastFlow provides several communication primitives for point-to-point communication, including send, receive, and probe.
These primitives can be used to implement a wide range of parallel algorithms, such as parallel sorting and matrix multiplication.
Collective communication in FastFlow is implemented using a set of communication patterns, such as map, reduce, and scatter.
These patterns provide a higher-level abstraction for communication and synchronization among processes, making it easier to implement parallel algorithms.

Collective communication in FastFlow is optimized for multi-core and distributed systems, and can be used to implement a wide range of parallel algorithms, such as graph algorithms and machine learning algorithms.
When to use collective communication in FastFlow depends on the specific requirements of the parallel algorithm being implemented. Collective communication is useful when the parallel algorithm requires communication and synchronization among all processes in a group.

Collective communication can be more efficient than point-to-point communication for certain types of parallel algorithms, such as those that require global reduction or data movement. However, collective communication can also introduce additional overhead and may not be necessary for all parallel algorithms.

2) Define Service Time, Latency, and Efficiency. In which context each metric is more meaningful?

- **Service time (T_S)**: the inverse of the throughput: the time needed to output the results of a task belonging to a task sequence after having delivered the results of the previous task
  - high-performance computing systems wants the service time to be as low as possible
- **Latency:** the time between the reception of an input and the deliver of the output
  - real-time systems where responsiveness is critical requires a low latency
- **Efficiency:** the ration between the ideal parallel execution time and the actual execution time (basically it is the speedup divided by the number of worker). It is useful to identify the optimal number of processors to use for a given task

3) How is the Speedup Calculated?

The speedup is a derived performance indicator and it is defined as the ration between the best known sequential execution time (on the target architecture at hand) and the parallel execution time.
The speedup is a function of n, where n is the parallelism degree of the parallel program:

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)}$$

4) What should I add to a Farm that takes into account both time and energy consumption, considering that both depend on the number of workers?

I have to take into account the degree of parallelism and the frequency of the processor.
We compute the service time T_S and then we see how the energy consumptions changes as we increase/decrease the workers, and how it impact on the service time.

More workers means less CPU usage time (the service time decreases) with more consumption (the CPU is used more).
We can stop when it becomes inconvenient to increase the number of workers for overhead.

We could use an Autonomic Manager that decides to spawn more workers or to kill active workers based on the target energy consumption and service time.

5) Types of Parallel Patterns

We divide patterns by the kind of parallelism.

Stream Patterns

1. **pipeline:** a stream of input that "flow" through a series of functions (stages)
    1. the time needed to compute a single task (the time needed for a single input to enter and exit the pipeline), which is the pipeline latency, is close to the sum of the times required to compute all the stages
    2. the time needed to output a new result (service time) is close to the time spent to compute the longest stage of the pipeline
2. **farm:** a stream of input, to each element we apply a function
    1. the time needed to compute a single task, which is the farm latency, is close to the time needed to compute the function sequentially
    2. the time needed to delivery to different task results (service time) is close to the sequential time divided the parallelism degree of the farm

Data Patterns

1. **map:** apply a function to every element of the array
2. **reduce:** use a function to reduce in parallel using a reversed-tree fashion
3. **prefix:** similar to reduce, but to compute the prefix, as partial sums
4. **stencil:** compute the new item in a data structure as a function of the neighbors (kinda dynamic programming)
5. **divide & conquer:** divide in subproblems, resolve them in parallel and recombine the results

Huffman Project Related Questions

1) How could you use the Actor Model for the Counting part of the Huffman Project?

Same as the following question using MPI, same reasoning.
To each actor a chunk, count, send messages.

2) What kind of Refactoring Rule could you use?

Pipe Introduction, Parallel Degree Changing, ...

3) How would implement the COUNT phase using MPI?

Besides the initialization (using `MPI_init`, ...) I would use the `MPI_scatter` to distribute among threads the chunks of the file and then I would use a `MPI_reduce` using the addition as function between the resultant vectors.
Using this solution would increase the overhead due to the communication between workers, which we do not have in our implementation with native threads.