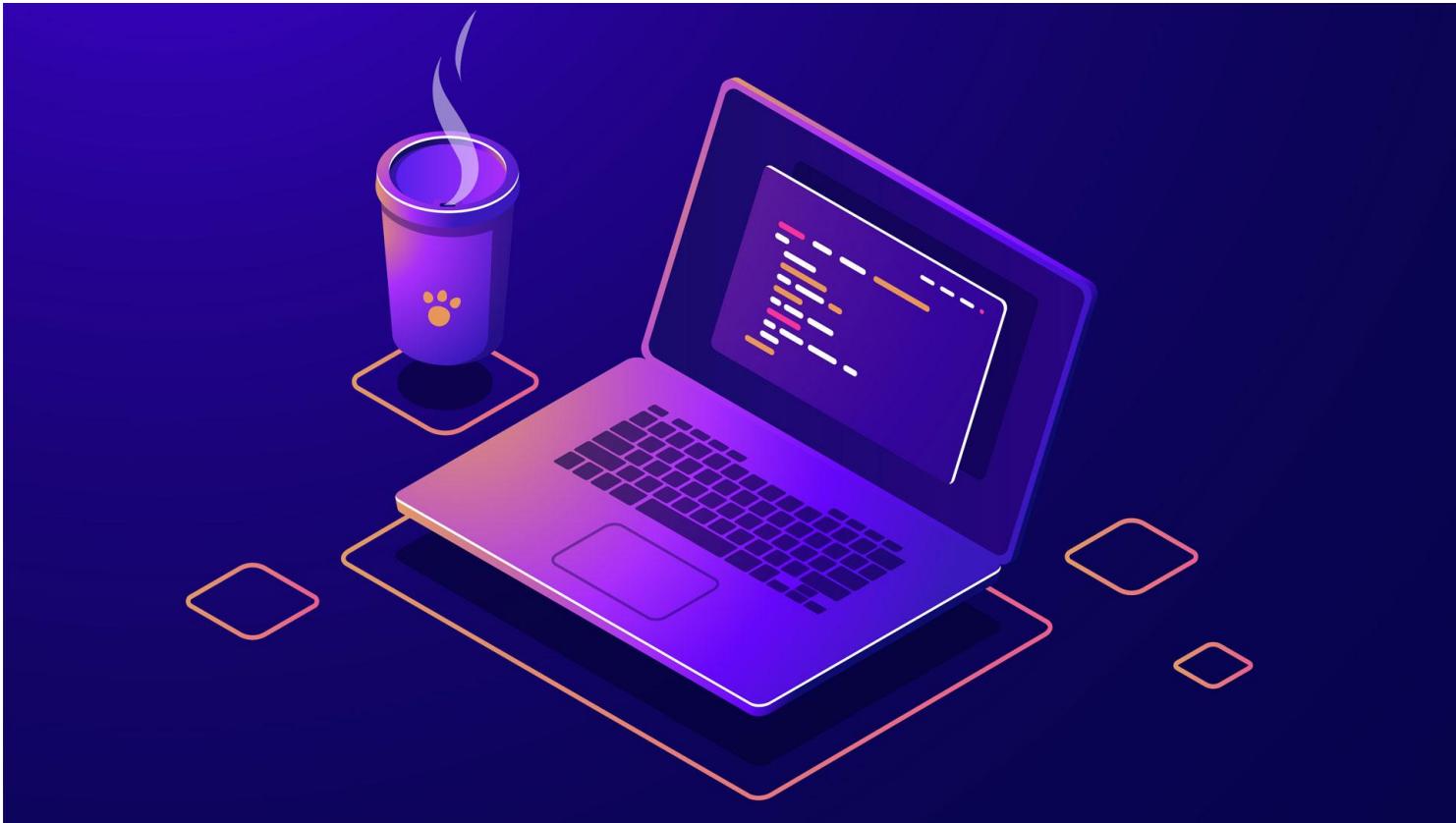


Introduction to Parallelism



Introduction to MPI



Introduction to MPI



- MPI, which stands for Message Passing Interface, is a message-passing system designed to facilitate communication and coordination between parallel processes.
- It is widely used for **parallel programming** and high-performance computing (HPC) applications.



Introduction to MPI



- MPI enables efficient **data exchange** and **synchronization** among processes running on different nodes of a parallel computing system.



Features of MPI



- Message Passing Model
- Parallelism
- Portability
- Scalability
- Process Topologies
- Many More...

Applications of MPI



- Scientific Simulations
- Molecular Dynamics
- Finite Element Analysis
- Data Analysis and Machine Learning

Introduction to MPI



MPI plays a crucial role in addressing the computational challenges posed by modern applications, enabling researchers and developers to harness the power of parallel processing for solving complex problems.



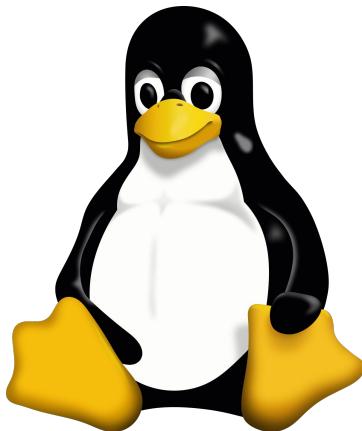
How to Install MPI



How to Install MPI on Linux



First of all, let us see how we can install MPI on Linux systems.



How to Install MPI on Linux



1 - Open the terminal of our system.

2 - Update our package list information:

```
sudo apt update
```

3 - Install MPI:

```
sudo apt install mpich
```

How to Install MPI on Linux



If we get the dependencies error we only need to run:

```
sudo apt --fix-broken install
```

And then proceed with the installation:

```
sudo apt install mpich
```

How to Install MPI on MacOS



Let us now see how we can install MPI on MacOS.



How to Install MPI on MacOS



We can use Homebrew package manager

```
brew install mpich
```

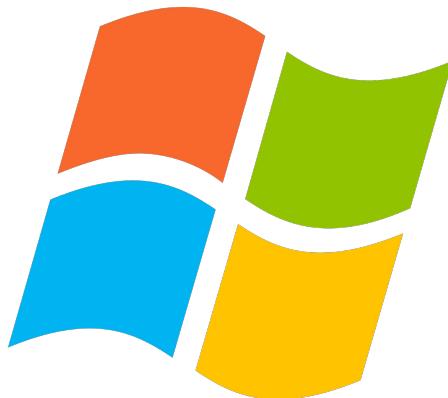
If we do not have Homebrew, we can install it:

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Alternatives for Windows



MPI is not officially supported on Windows to the same extent as it is on Linux and MacOS.

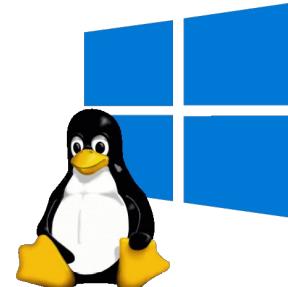


Alternatives for Windows



Two alternatives are proposed:

- Windows Subsystem for Linux.
- Use a virtual machine like VMware or Virtualbox.



First MPI Program



First MPI Program



On this lecture we are going to construct our first program in MPI.



Basic MPI Functions

- `MPI_Init`: initializes MPI execution environment.
- `MPI_Finalize`: finalizes MPI execution environment.
- `MPI_Comm_rank`: retrieves the process identifier.
- `MPI_Comm_size`: retrieves the number of processes.

Simple Example of argc and argv



```
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Argument count (argc): %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

Simple Example of argc and argv



If we run the program with this command:

```
./program word1 word2 word3
```



Argument count (argc): 4

Argument 0: ./program

Argument 1: word1

Argument 2: word2

Argument 3: word3

Point-to-Point Communication



Sending and Receiving Messages



Sending and Receiving Messages



- The basic communication mechanism of MPI is the transmittal of data between a pair of processes.
- This is called point-to-point communication.
- MPI provides a set of **send** and **receive** functions that allow the communication of **typed** data with an associated **tag**.

Sending and Receiving Messages



- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`

Blocking Operations



Blocking Operations



- The function `MPI_Recv` can be started whether or not a matching send has been posted.
- It is a **blocking** operation.
- The function `MPI_Send` can be started whether or not a matching receive has been posted.
- Thus, it is also a **blocking** operation.

Blocking Operations



- In ill-constructed programs, **blocking** operations may lead to **deadlock**, where all processes are blocked and no progress occurs.

Example of Deadlock



```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank == 0) {
MPI_Recv(msg,count,MPI_REAL,1,tag,MPI_COMM_WORLD,&status);
MPI_Send(msg,count,MPI_REAL,1,tag,MPI_COMM_WORLD);
}

else if (rank == 1) {
MPI_Recv(msg,count,MPI_REAL,0,tag,MPI_COMM_WORLD,&status);
MPI_Send(msg,count,MPI_REAL,0,tag,MPI_COMM_WORLD);
}
```



Messages Order

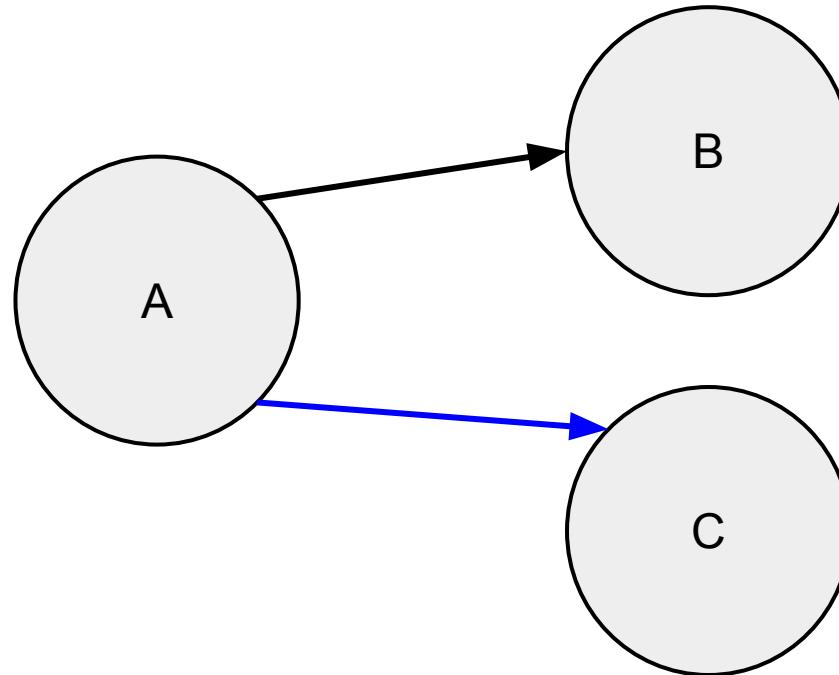


- Messages are non-overtaking.

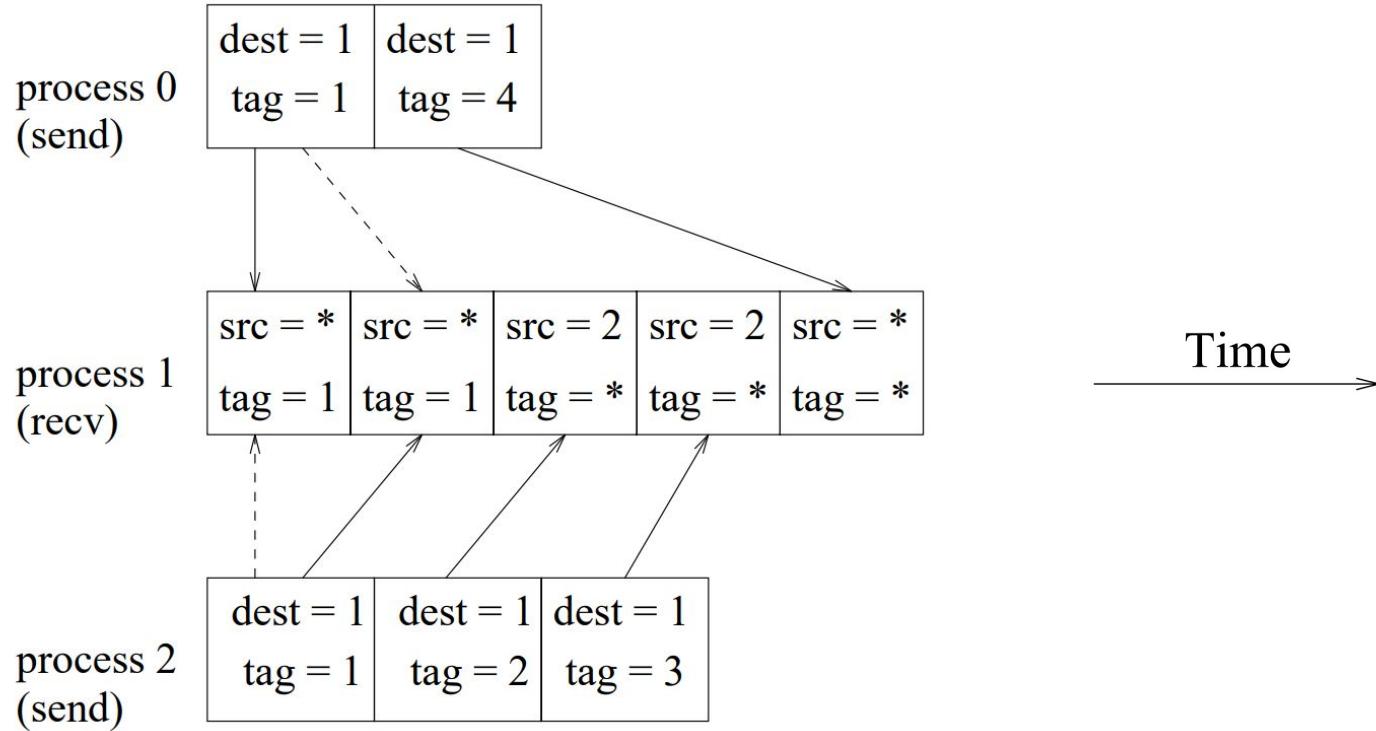
Messages Order



- Messages are non-overtaking.



Messages Order



Messages Order



- If a sender sends two messages in succession to the same destination and both match the same receive then the receive cannot get the second message if the first message is still pending.
- If a receiver posts two receives in succession and both match the same message then the second receive operation cannot be satisfied by this message if the first receive is still pending.

Collective Operations



Collective Operations



- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
- `int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,MPI_Op op, int root, MPI_Comm comm)`

Broadcast



- **MPI_Bcast** distributes data from one process (the root) to all others in a communicator.

```
int array[100];
int root = 0;
...
MPI_Bcast(array,100,MPI_INT,root,MPI_COMM_WORLD);
```

Reduce



- **MPI_Reduce** combines data from all processes in communicator and returns it to one process.



Computation of π

- The value of π is equal to this definite integral:

$$\int_0^1 \frac{4}{(1+x^2)} dx$$



Computation of π

- We can approximate this value using the midpoint rule, a numerical integration method.

$$h \sum_{i=1}^n \frac{4}{1 + \left(\frac{i-0.5}{n}\right)^2}$$

where $h = \frac{1}{n}$

Parallel Region



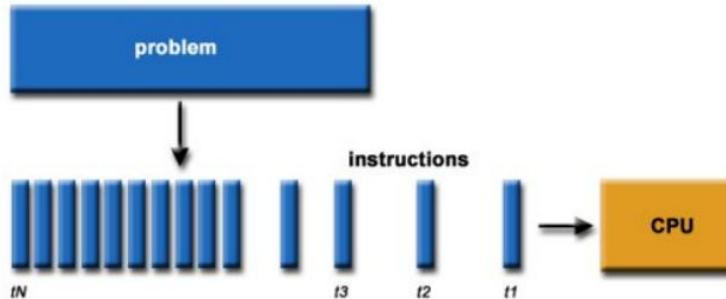
1. Process 0 reads the value of n .
2. The value of n is broadcast to all MPI processes.
3. Each MPI process calculates its partial sum using a portion of the intervals assigned to it.
4. The individual results from each process are reduced using MPI_Reduce to obtain the total sum on the root process (process with *rank* 0).



Serial Execution

Usually programs are written with a serial execution model in mind

- Program is composed of a sequence of instructions (arithmetic, memory read and write, control, ...) ...
- ... to be run on a computer with a single processor (CPU)



- Instructions are executed one after another, only one at any moment in time

Serial Execution



The execution time of a program with N instructions on a processor that is able to execute F instructions per second is

$$T = N \div F$$

One could execute the program faster (i.e. reduce T) by augmenting the value of F . And this has been the trend during more than 30 years of technology and computer architecture evolution.

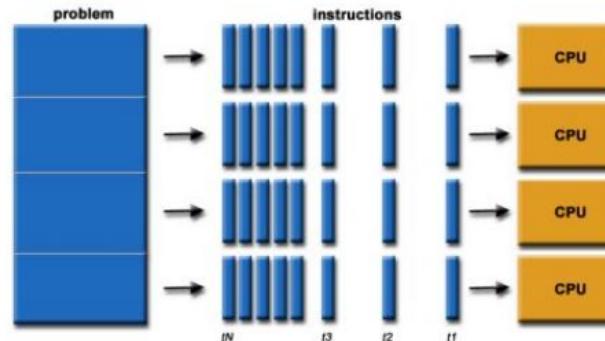
Parallel Execution



So another way to reduce the execution time of a program T

$$T = N \div F$$

would be to split the program into discrete parts, to be called tasks, and use multiple processors (CPUs) to execute them at the same time

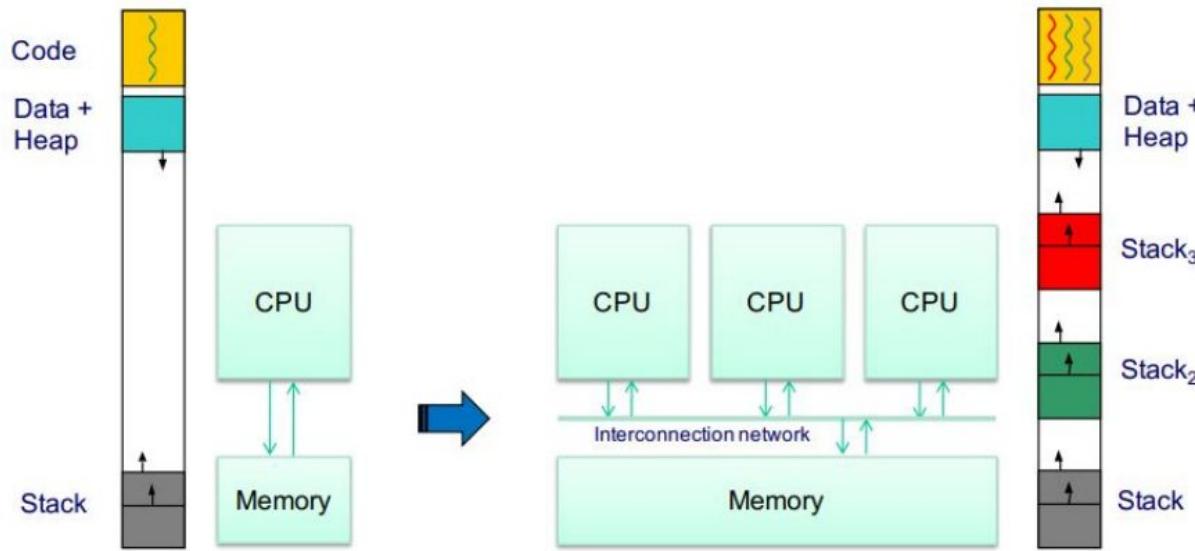


And data?

Parallel Execution



Shared-memory architecture and memory address space

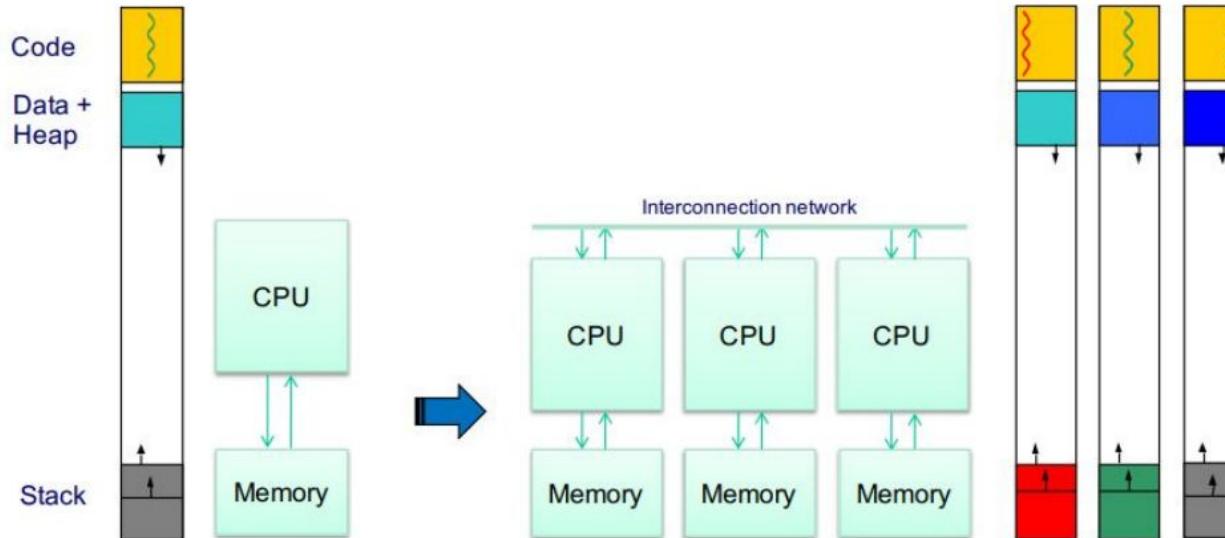


Hardware support for coherent data sharing and tight synchronization

Parallel Execution



Distributed-memory architecture and memory address space



Hardware support for remote data accesses and communication

Parallel Execution

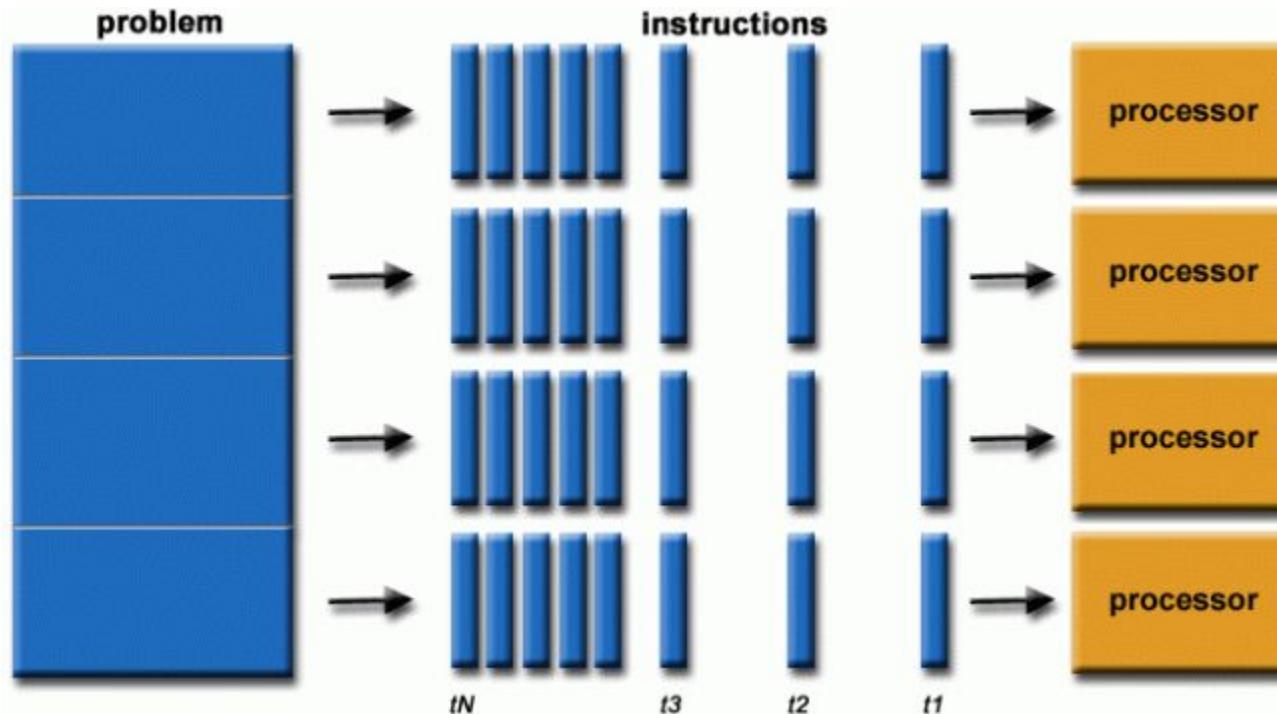


Ideally, each processor could receive $\frac{1}{P}$ of the program, reducing its execution time by P

$$T = (N \div P) \div F$$

Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

So How Can We Achieve This?



Instructor Social Media

Youtube: Lucas Science



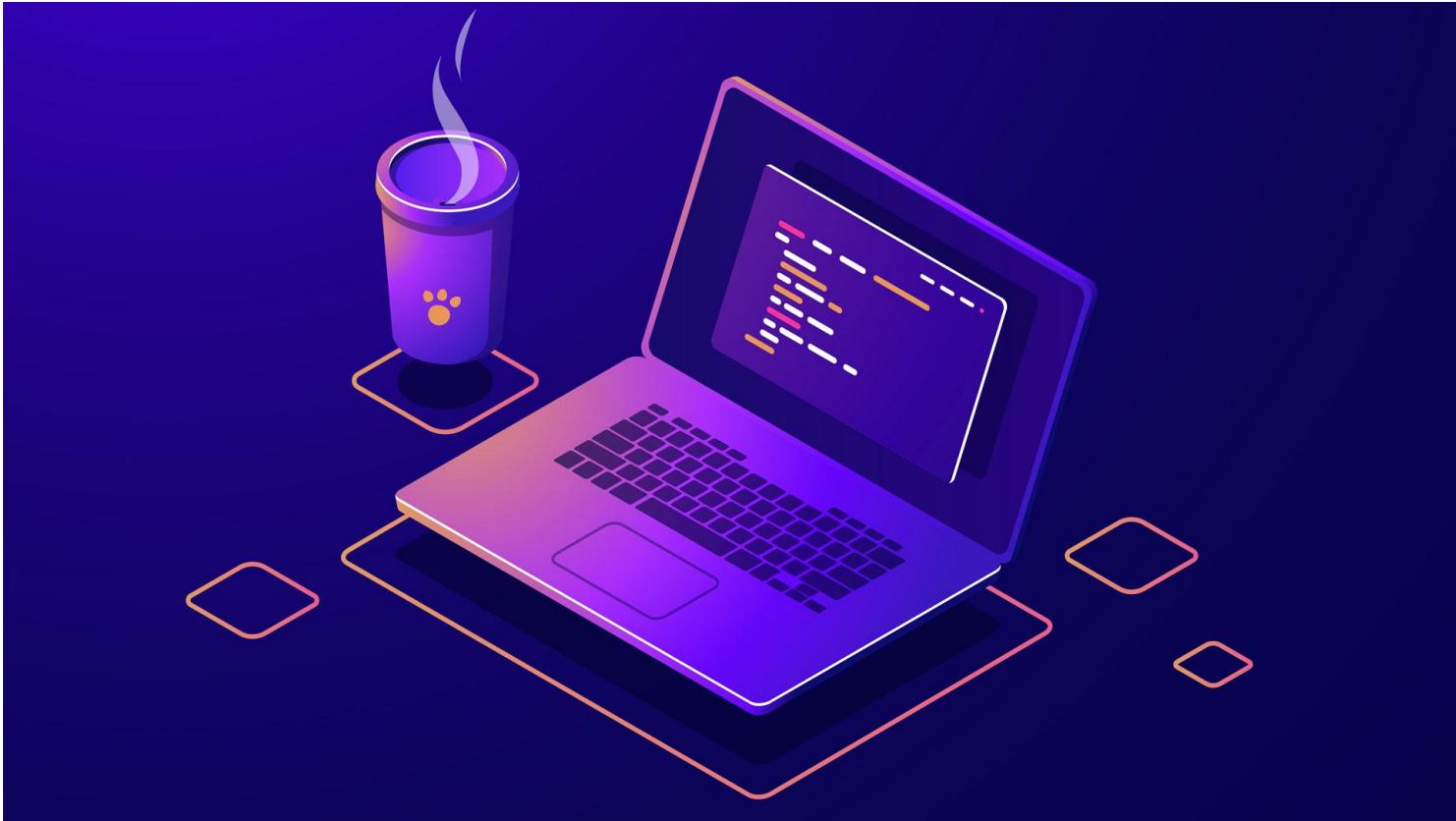
Instagram: lucaasbazilio



Twitter: lucasebazilio



Expressing Tasks



Motivation



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

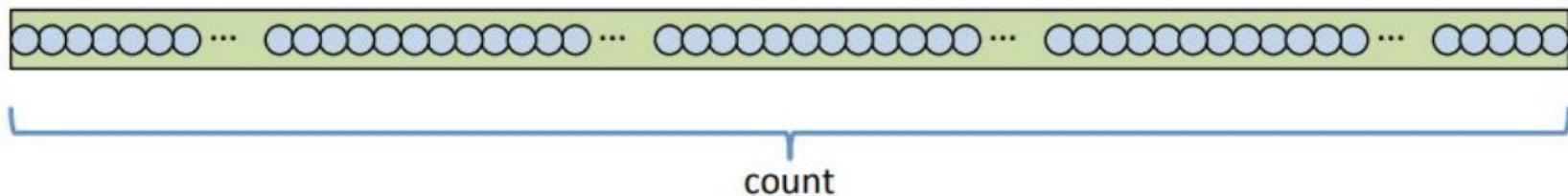
And assume that we want to count how many Green cars are available to sell.

Motivation



- One could traverse all the records $X[0] \dots X[n-1]$ in the database X and check if the Color field matches the required value Green, storing the number of matches in variable count

database X





Motivation

- A possible sequential program could be:

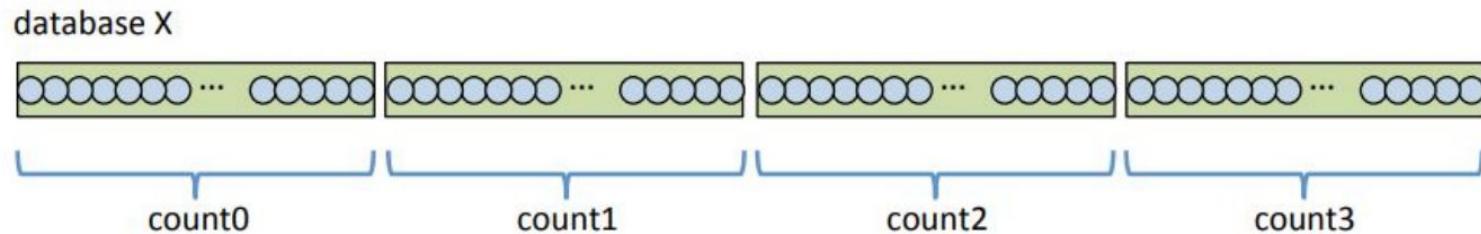
```
count = 0;  
for ( i = 0 ; i < n ; i++ )  
    if (X[i].Color == "Green") count++;
```

whose computation time on a single processor would be proportional to the number of records in the database $T_1 \propto n$

Tasks



- One could divide the traversal in P groups (tasks), for example for $P = 4$:

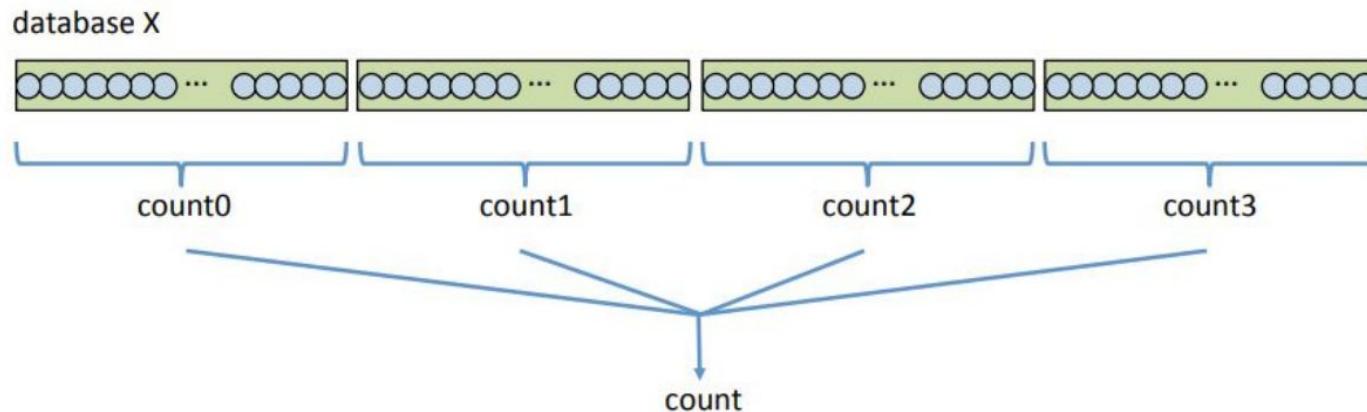


checking the Color field for a subset of $n \div P$ consecutive records, and counting on a per-task "private" copy of variable count

Tasks



- However, we still need to "globally" count the number of records found that match the condition by combining the individual "private" counts into the original count variable



Tasks



- Up to this point you could anticipate that the computation time would be divided by the number of tasks P if P workers are used to do the computation

$$T_P = T_1 \div P$$

with an additional "overhead" to perform this global reduction

$$T_p = T_1 \div P + T_{ovh}(P)$$

probably proportional to the number of workers P

Instructor Social Media

Youtube: Lucas Science



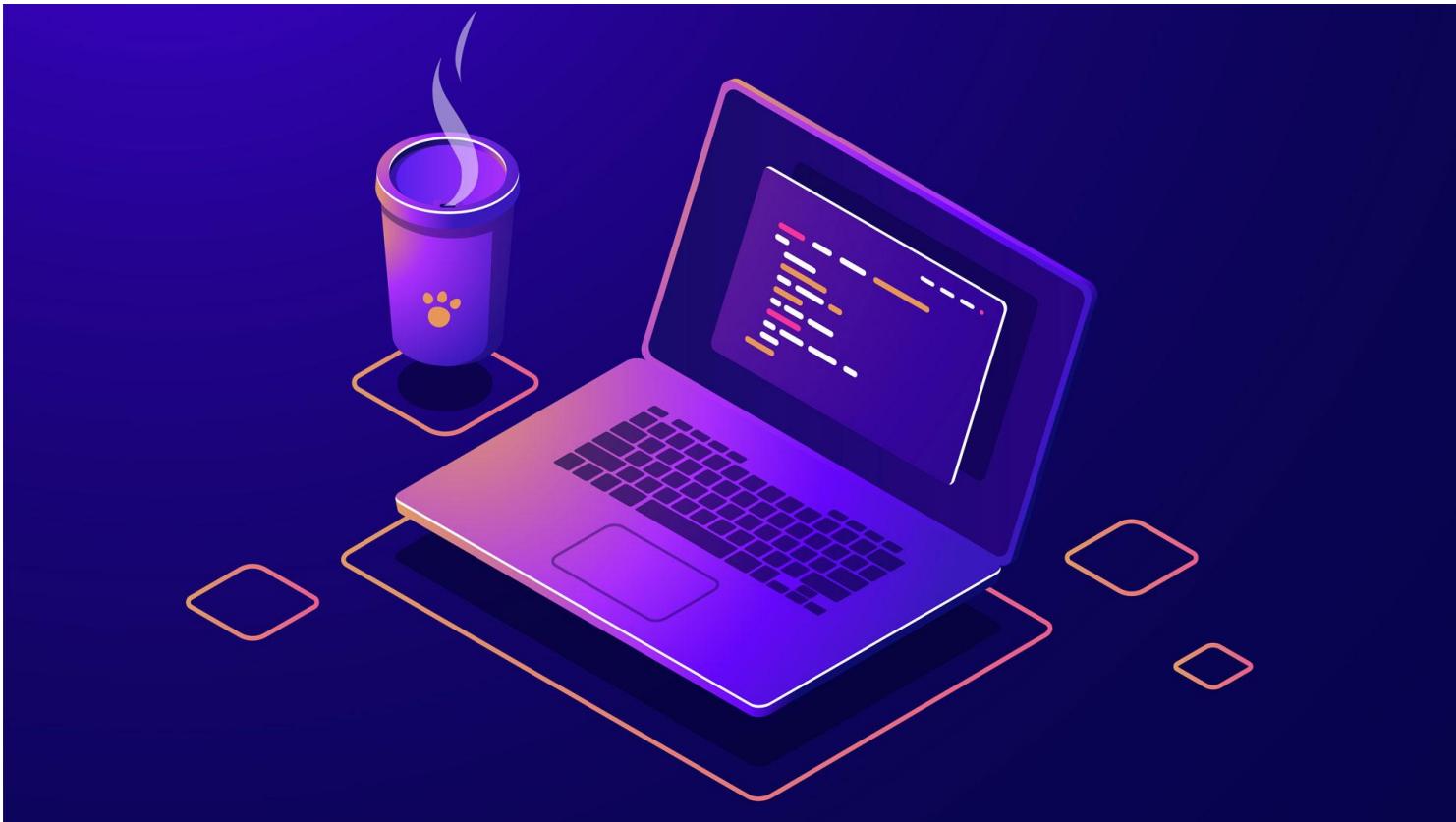
Instagram: lucaasbazilio



Twitter: lucasebazilio



Tasks and Dependencies





Motivation

Consider now that we want to execute the following query:

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

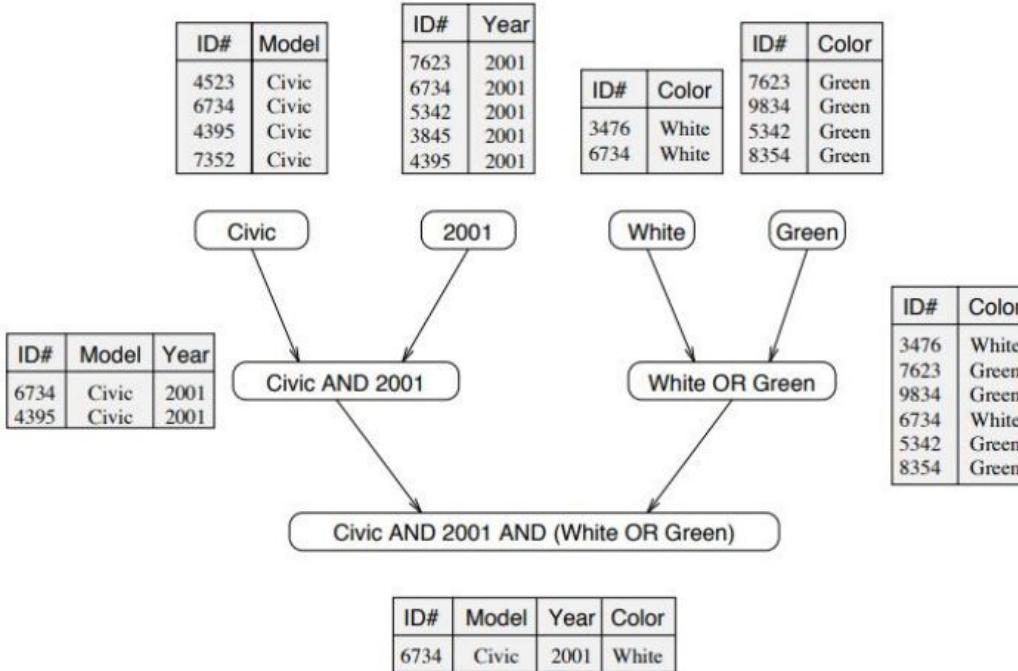
on our car dealer database:

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



Tasks

- A possible query plan could be:



Tasks and Dependencies

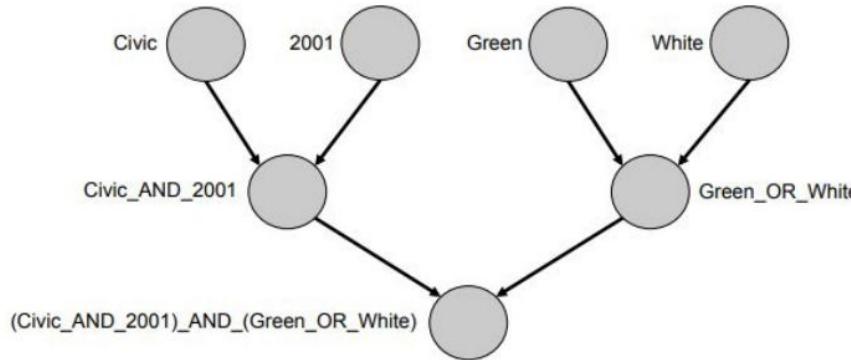


- Each of these operations in the query plan could be a task, each computing an intermediate table of entries that satisfy particular conditions
- Are they independent?
 - Some of them are, for example tasks "*Civic*", "*2001*", "*Green*" and "*White*"
 - Others are not independent. For example, task "*Civic_AND_2001*" can not start its execution until both tasks "*Civic*" and "*2001*" complete

Tasks and Dependencies



- Dependences impose task execution ordering constraints that need to be fulfilled in order to guarantee correct results

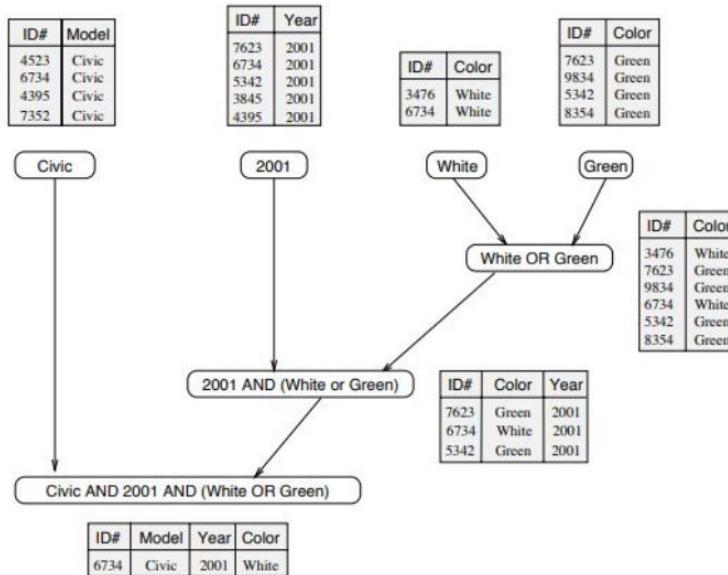


- Task dependence graph:** graphical representation of the task decomposition

Tasks and Dependencies



- Other query plans are possible, for example



... with different task dependence graphs and potential to execute tasks in parallel

Instructor Social Media

Youtube: Lucas Science



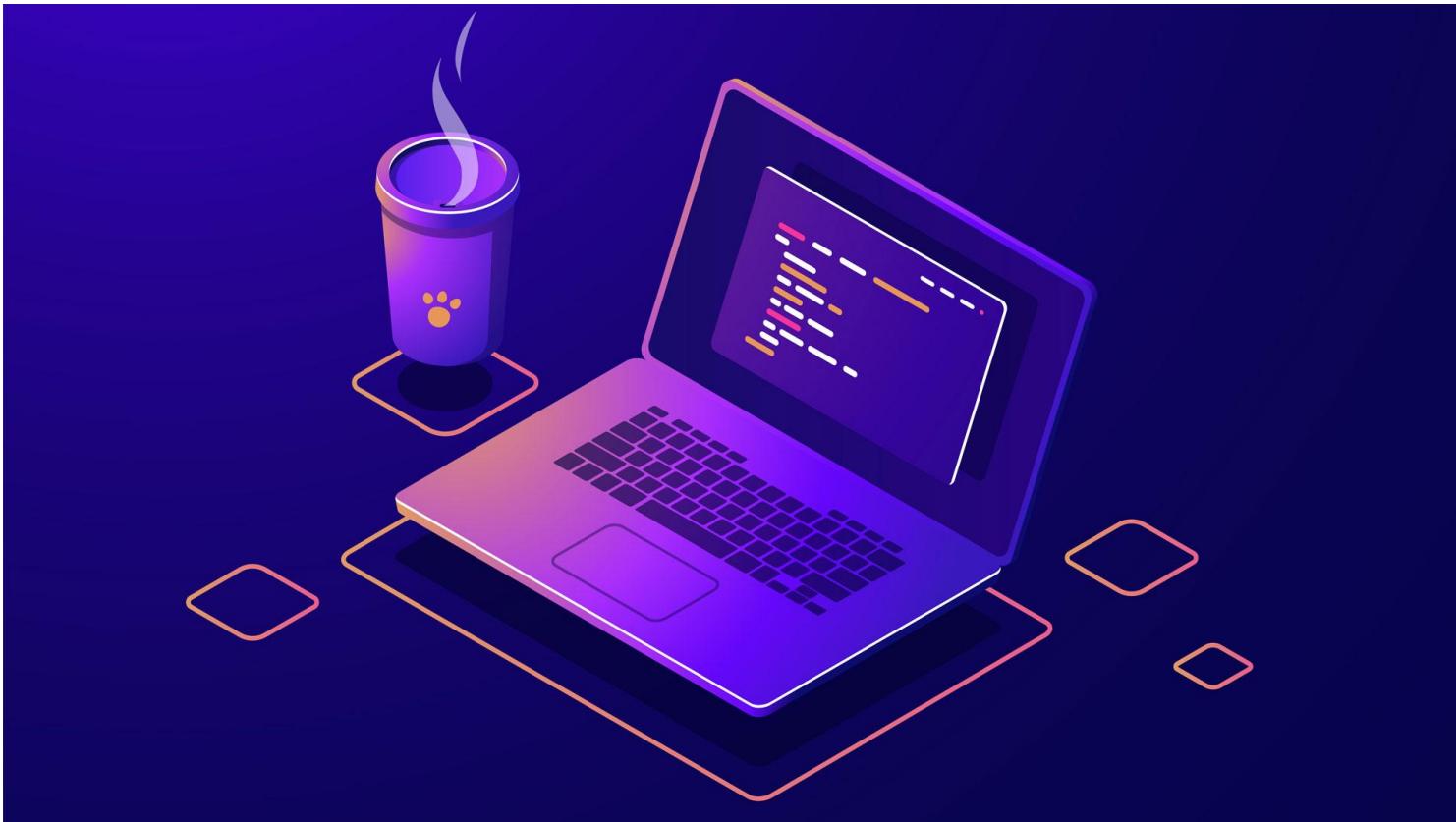
Instagram: lucaasbazilio



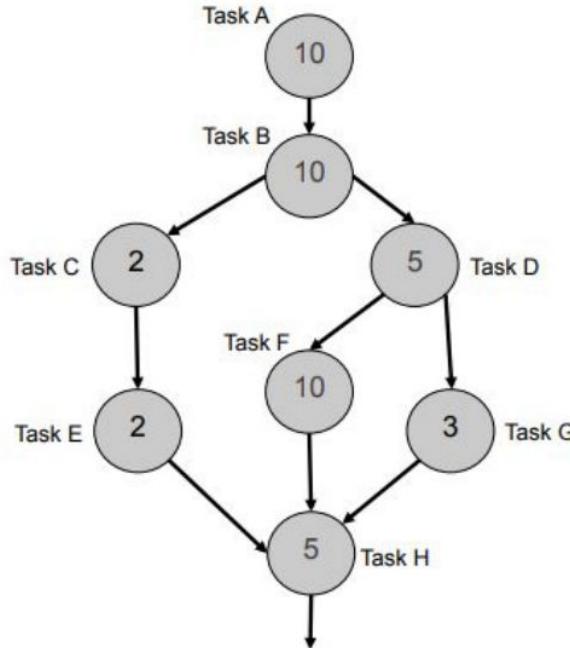
Twitter: lucasebazilio



Task Dependency Graph



Task Dependency Graph

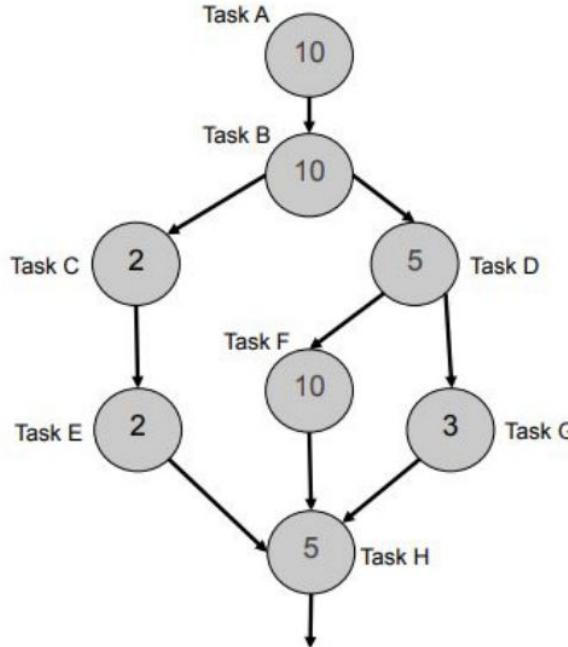


- **Task dependence graph** abstraction

- Directed Acyclic Graph
- Node = task, its weight represents the amount of work to be done
- Edge = dependence, i.e. successor node can only execute after predecessor node has completed

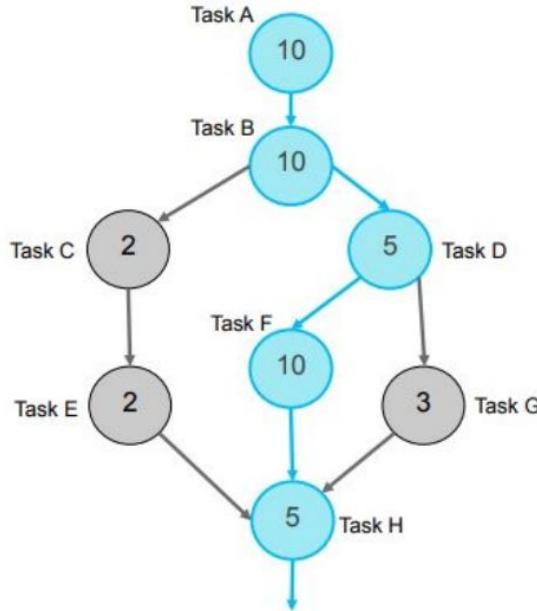


Total Work T_1



- Parallel machine abstraction
 - P identical processors
 - Each processor executes a node at a time
- $T_1 = \sum_{i=1}^{nodes} (work_node_i)$

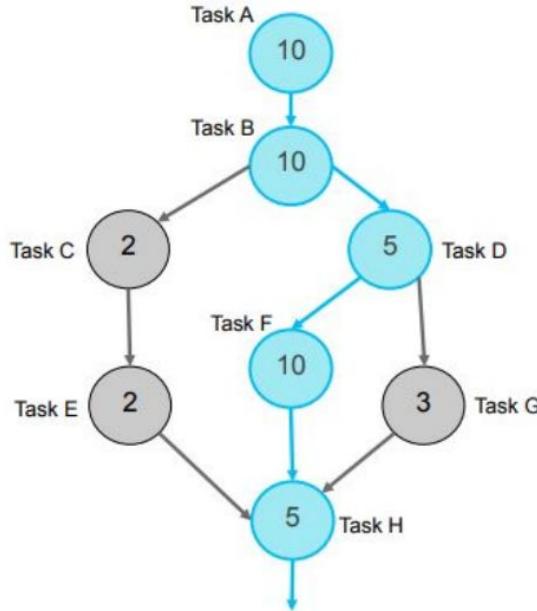
Critical Path Work T_∞



- Critical path: path in the task graph with the highest accumulated work
- $T_\infty = \sum_{i \in \text{criticalpath}} (\text{work_node}_i)$, assuming sufficient processors

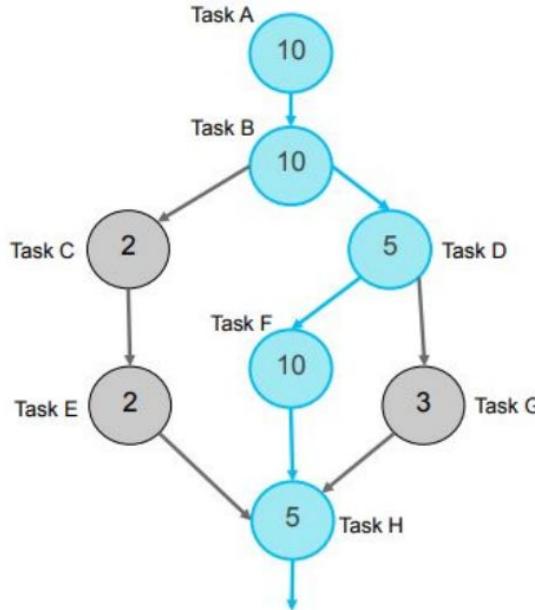


Parallelism and P_{min}



- Parallelism = T_1/T_∞ , if sufficient processors were available
- P_{min} is the minimum number of processors necessary to achieve Parallelism

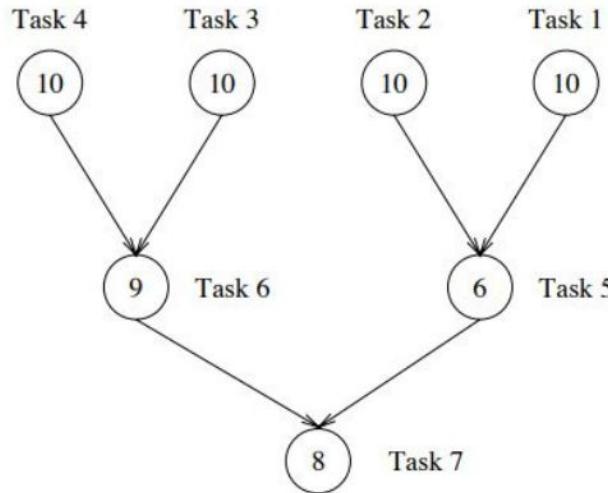
Wrap Up



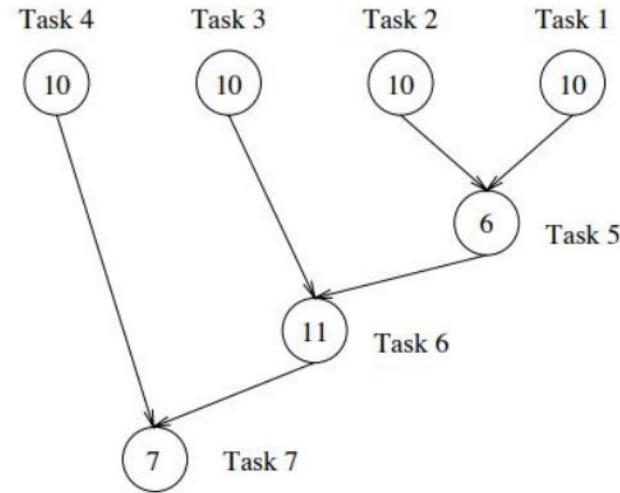
- $T_1 = 47$ including tasks $\{ABCDEF\}$
- Possible paths:
 - {ABCEH} with total cost 29
 - {ABDFH} with total cost 40
 - {ABDHG} with total cost 33
- $T_\infty = 40$ for critical path {ABDFH}
- Parallelism = $47/40 = 1.175$
- $P_{min} = 2$



Consider the task dependency graphs for the two database queries, assuming *work_node* is proportional to the number of inputs to be processed



(a)



(b)

Which are T_1 , T_∞ and *Parallelism* in each case?

Instructor Social Media

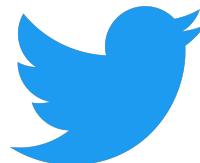
Youtube: Lucas Science



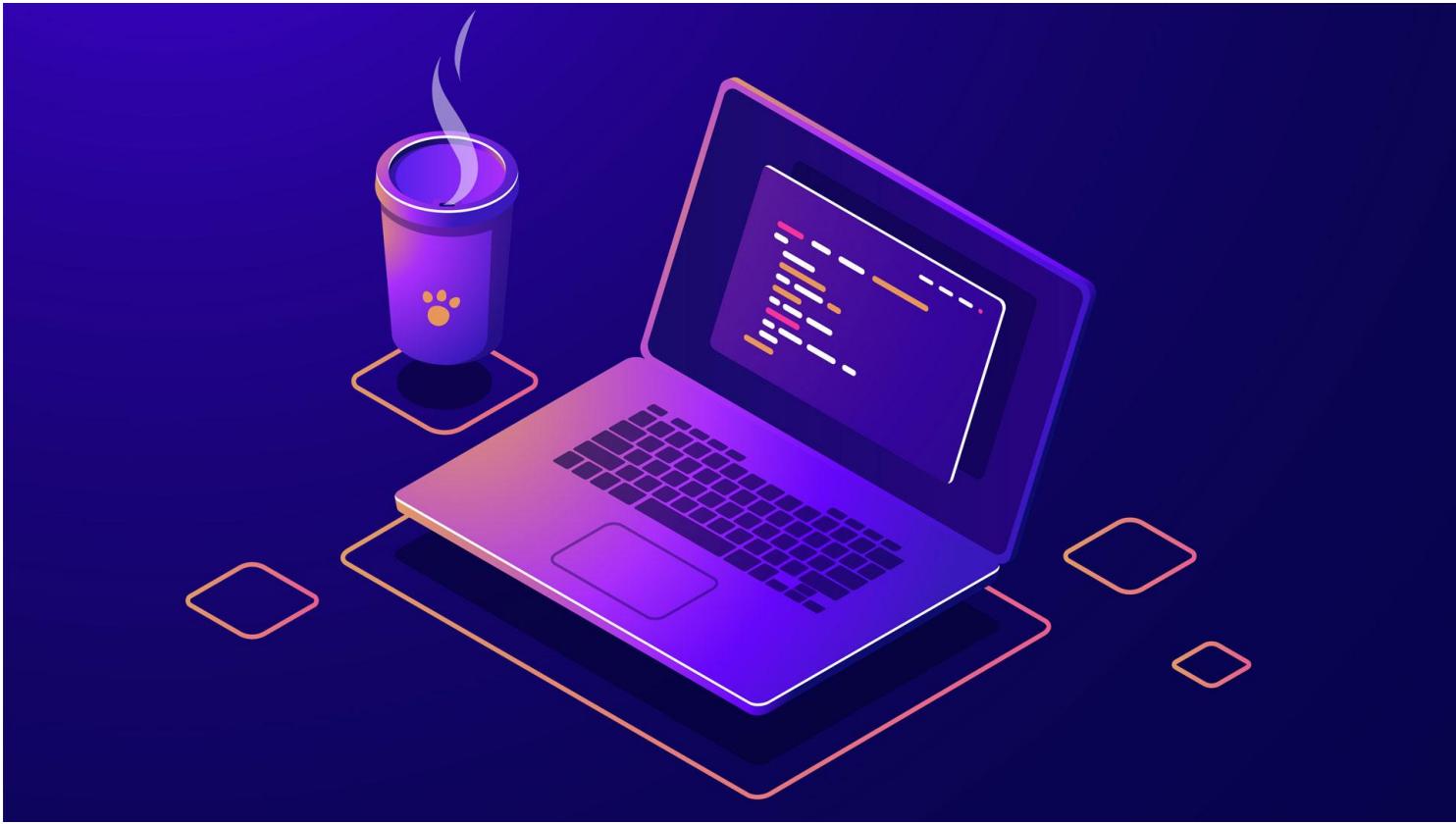
Instagram: lucaasbazilio



Twitter: lucasebazilio



Granularity and Parallelism



Granularity and Parallelism



- The granularity of the task decomposition is determined by the computational size of the nodes (tasks) in the task graph
- Example: counting matches in our car dealer database

```
count = 0;  
  
for ( i=0 ; i< n ; i++ )  
    if (X[i].Color == "Green") count++;
```

Coarse-grain decomposition:
The whole loop is a task
Parallelism = 1

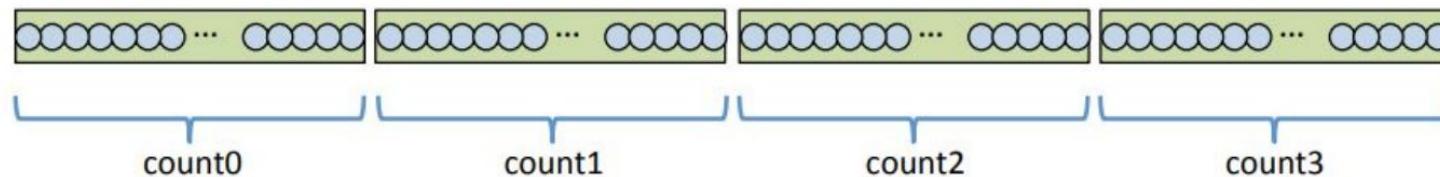
Fine-grain decomposition:
Each iteration of the loop is a task
Parallelism = n

Granularity and Parallelism



- Example: A task could be in charge of checking a number of consecutive elements m of the database:

database X



with a potential *parallelism* = $n \div m$

$$1 < (n \div m) < n$$

Granularity and Parallelism



- It would appear that the parallelism is higher when going to fine-grain task decompositions

	Coarse grain	Fine grain	Medium grain
Number of tasks	1	n	n / m
Iterations per task	n	1	m
Parallelism	1	n	n / m

- However, tradeoff between potential parallelism and overheads related with its exploitation (e.g. creation of tasks, synchronization, exchange of data, ...)

Instructor Social Media

Youtube: Lucas Science



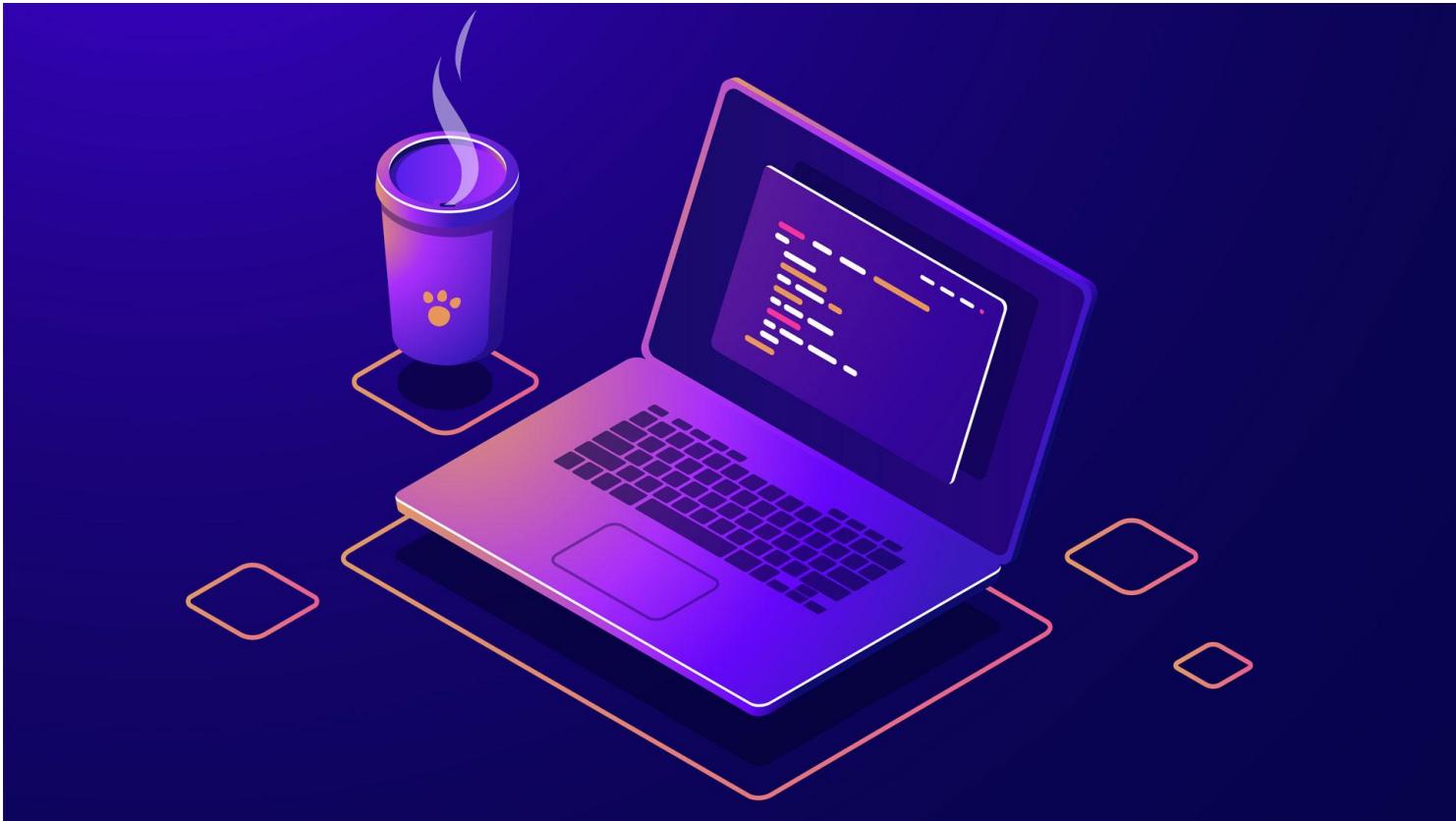
Instagram: lucaasbazilio



Twitter: lucasebazilio



Task Definition



Task Definition



- ▶ Can the computation be divided in parts?¹
 - ▶ Task decomposition: based on the processing to do (e.g. functions, loop iterations)
 - ▶ Data decomposition: based on the data to be processed (e.g. elements of a vector, rows of a matrix) (implies task decomposition)
 - ▶ There may be (data or control) dependencies between tasks
- ▶ Metrics to understand how our task/data decomposition can potentially behave
- ▶ Factors: granularity and overheads

Task Definition



- ▶ TDG: directed acyclic graph to represent tasks and dependencies between them
- ▶ Metrics:
 - ▶ $T_1 = \sum_{i=1}^{nodes} (work_node_i)$
 - ▶ $T_\infty = \sum_{i \in criticalpath} (work_node_i)$, assuming sufficient (infinite) resources
 - ▶ $Parallelism = T_1/T_\infty$
 - ▶ P_{min} is the minimum number of processors necessary to achieve $Parallelism$
- ▶ Task granularity vs. number of tasks



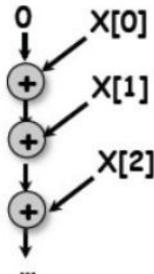
Example 1: Vector Sum

Compute the sum of elements $X[0] \dots X[n-1]$ of a vector X

```
sum = 0; for ( i=0 ; i< n ; i++ ) sum += X[i];
```

Task definition: each iteration of the i loop is a task.

- ▶ **TDG** (with input data):



- ▶ **Metrics:**

$$T_1 \propto n$$

$$T_\infty \propto n$$

$$\text{Parallelism} = 1$$

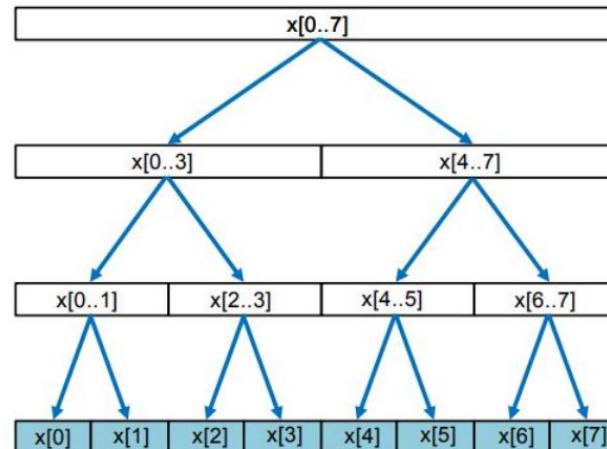
How can we design an algorithm which leads to a TDG with more parallelism?



Example 1: Vector Sum

Writing a **recursive version** of the sequential program to compute the sum of elements $X[0] \dots X[n-1]$ of a vector X , following a *divide-and-conquer* strategy:

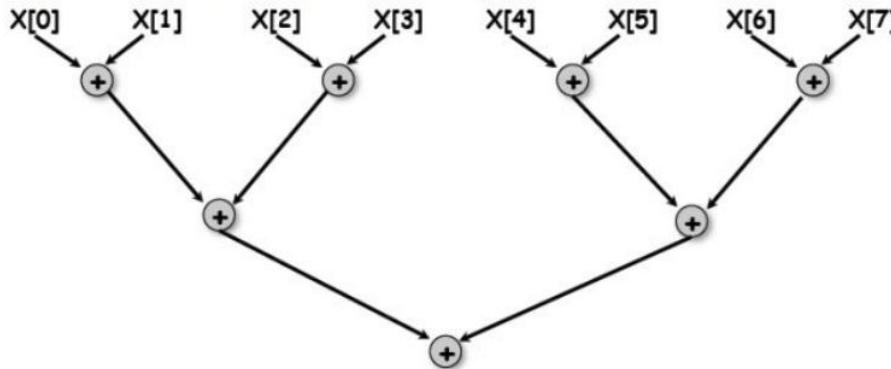
```
int recursive_sum(int *X, int n) {  
    int ndiv2 = n/2;  
    int sum=0;  
  
    if (n==1) return X[0];  
  
    sum1 = recursive_sum(X, ndiv2);  
    sum2 = recursive_sum(X+ndiv2, n-ndiv2);  
    return sum1+sum2;  
}  
  
void main() {  
    int sum, X[N];  
    ...  
    sum = recursive_sum(X,N);  
    ...  
}
```





Example 1: Vector Sum

- ▶ **Task definition:** each invocation to recursive_sum
- ▶ **TDG (with input data):**



- ▶ **Metrics:**
 $T_1 \propto n$; $T_\infty \propto \log_2(n)$; *Parallelism* $\propto (n \div \log_2(n))$
- ▶ Same problem can be expressed with different algorithms/implementations leading to different metrics

Instructor Social Media

Youtube: Lucas Science



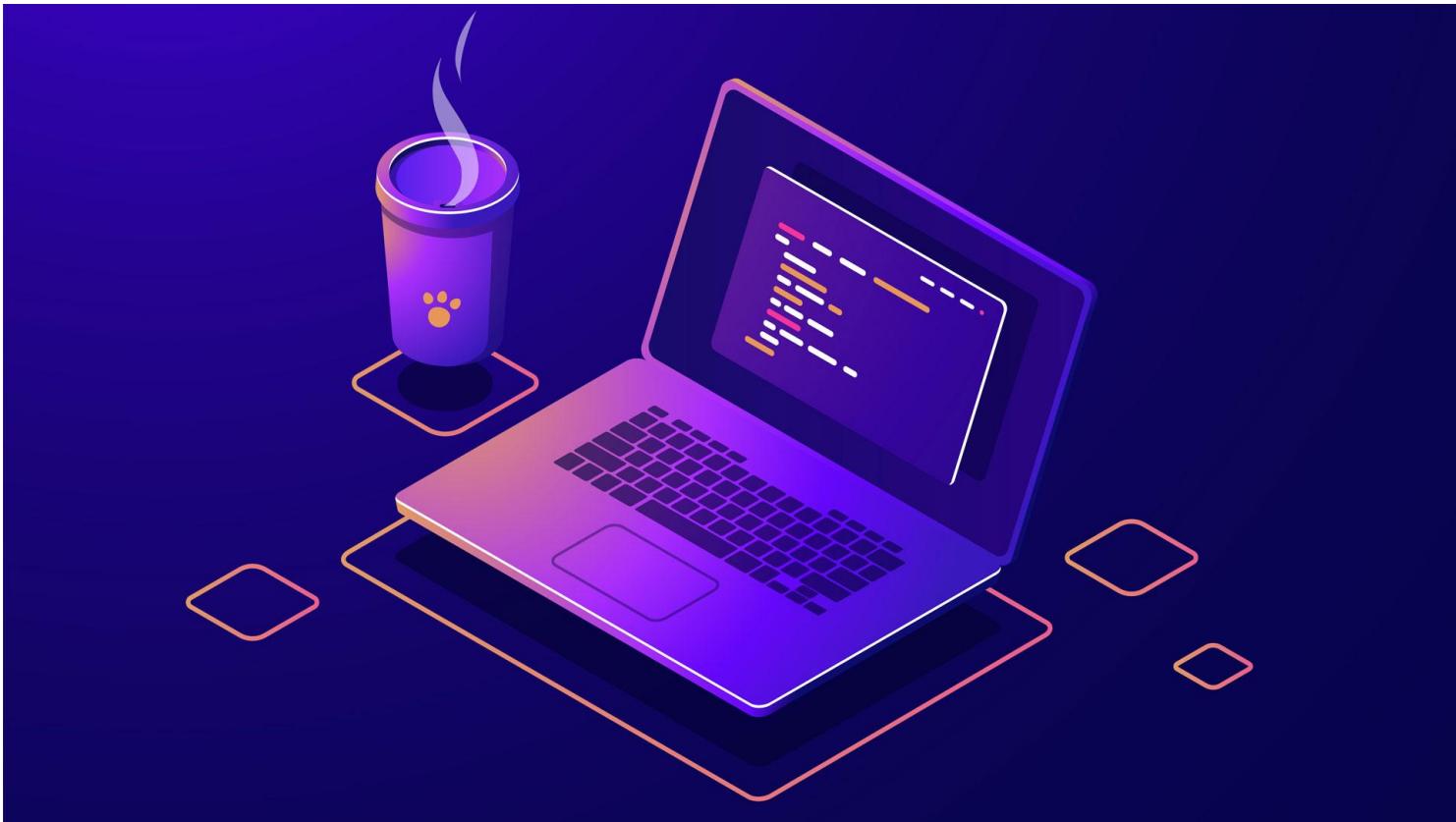
Instagram: lucaasbazilio



Twitter: lucasebazilio



Advanced Granularity



Advanced Granularity



Given a sequential program, the number of tasks that one can generate and the size of the tasks (what is called **granularity**) are related one to the other.

- ▶ Fine-grained tasks vs. coarse-grained tasks
- ▶ The parallelism increases as the decomposition becomes finer in granularity (small tasks) and vice versa

Example 1: matrix-vector product

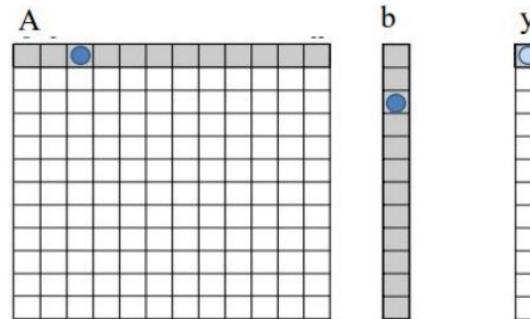


Fine-grained Decomposition



Example: matrix-vector product (n by n matrix):

- ▶ A task could be each individual \times and $+$ in the dot product that contributes to the computation of an element of y
 $(y[i] = y[i] + A[i][j] * b[j])$

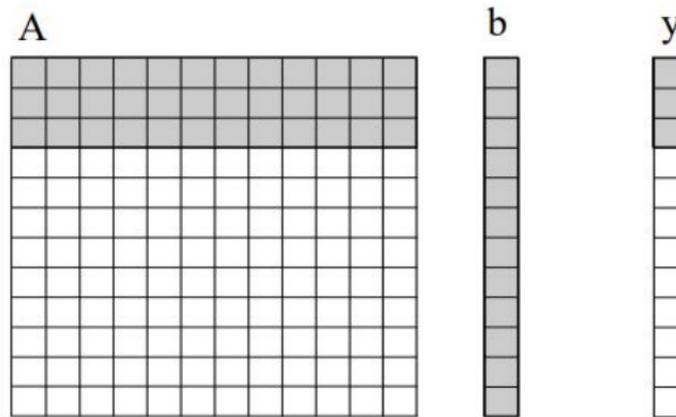


- ▶ A task could also be each complete dot product to compute an element of y ($y[i] = y[i] + \sum_{j=1}^{j=n} (A[i][j] * b[j])$)

Coarse-grained Decomposition



- ▶ A task could be in charge of computing a number of consecutive elements of y (e.g. three elements)



- ▶ A task could be in charge of computing the whole vector y

So...



- ▶ It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity but...
 - ▶ Inherent bound on how fine the granularity of a computation can be
 - ▶ e.g. *matrix-vector multiply*: (n^2) concurrent tasks.
 - ▶ Tradeoff between the granularity of a decomposition and associated overheads (sources of overhead: creation of tasks, task synchronization, exchange of data between tasks, ...)
 - ▶ The granularity may determine performance bounds

Example 2: stencil computation using Jacobi solver

Stencil algorithm that computes each element of matrix `utmp` using 4 neighbor elements of matrix `u`, both matrices with $n \times n$ elements

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                      // element u[i][j]
            utmp[n*i + j] = tmp / 4;                // element utmp[i][j]
        }
    }
}
```

Example 2: stencil computation using Jacobi solver

What tasks can be? Assume: 1) the innermost loop body takes t_{body} time units; and 2) n is very large, so that $n - 2 \simeq n$

Task is ... (granularity)	Num. tasks	Task cost	T_1	T_∞	Parallelism
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1
Each iteration of i loop	n	$n \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	n
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{body}$	t_{body}	n^2
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$
A block of $r \times c$ iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$

Finer grain task decomposition → higher parallelism, but ...

Example 2: stencil computation using Jacobi solver

... what if each task creation takes t_{create} ?

Task is ... (granularity)	Num. tasks	Task cost	Task creation ovh
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	t_{create}
Each iteration of i loop	n	$n \cdot t_{body}$	$n \cdot t_{create}$
Each iteration of j loop	n^2	t_{body}	$n^2 \cdot t_{create}$
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$(n \div r) \cdot t_{create}$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$(n^2 \div c) \cdot t_{create}$
A block of $r \times c$ iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$(n^2 \div (r \cdot c)) \cdot t_{create}$

Trade-off between task granularity and task creation overhead

Instructor Social Media

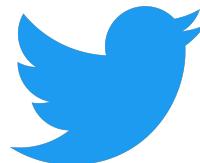
Youtube: Lucas Science



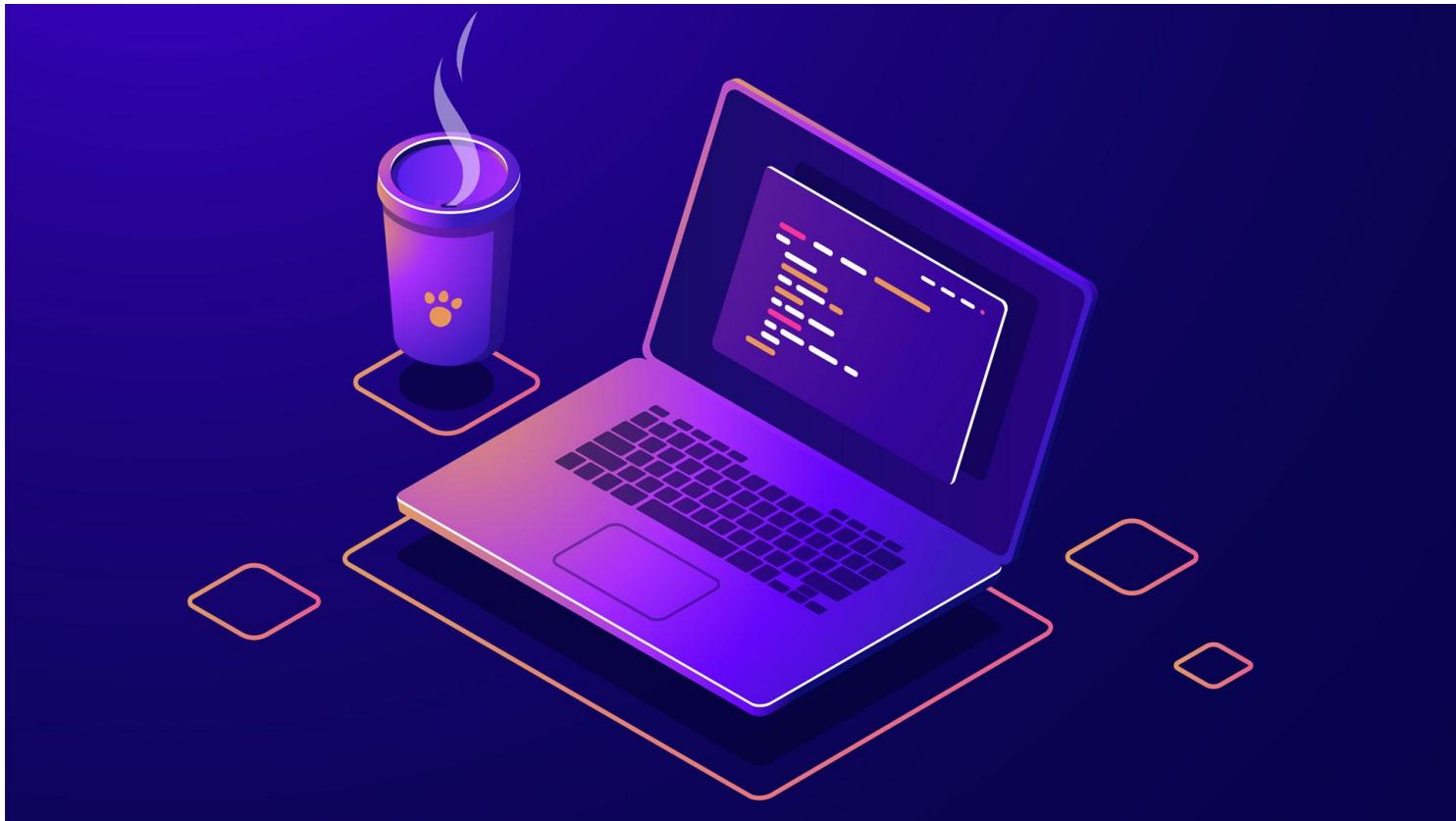
Instagram: lucaasbazilio



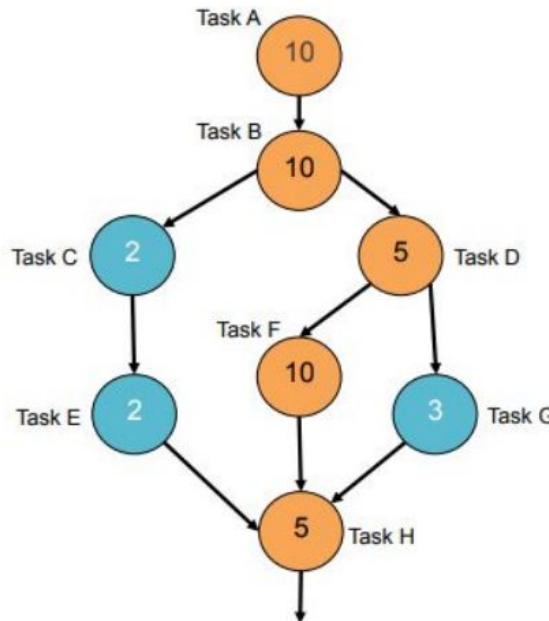
Twitter: lucasebazilio



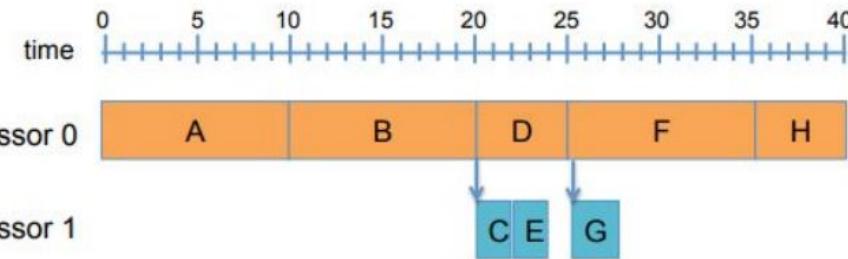
Speedup and Efficiency



Execution Time in P processors



- $T_p =$ execution time on P processors
- **Task scheduling:** how are tasks assigned to processors? For example:

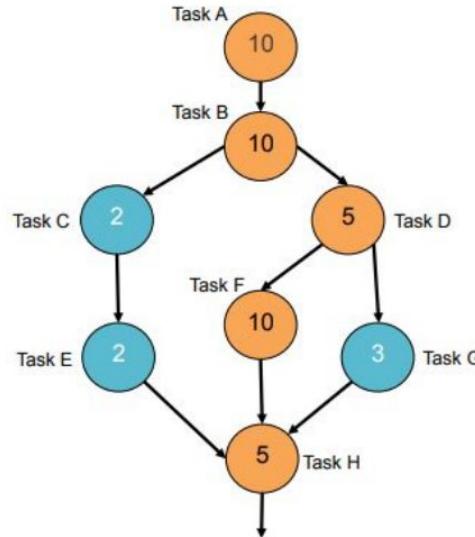


- Lower bounds
 - $T_p \geq T_1/P$
 - $T_p \geq T_\infty$



Speedup

Speedup S_p : relative reduction of the sequential execution time when using P processors

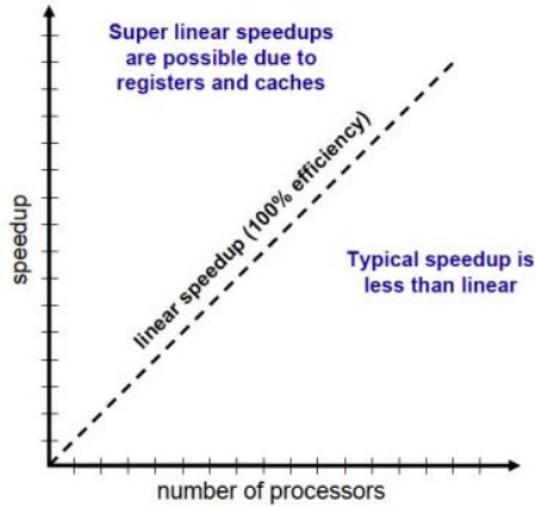


- $S_p = T_1 \div T_p$
- In this example:
 $T_2 = 40, S_2 = 47/40 = 1.175$

Scalability and Efficiency



- Scalability: how the speed-up evolves when the number of processors is increased
- Efficiency: $E_p = S_p \div P$

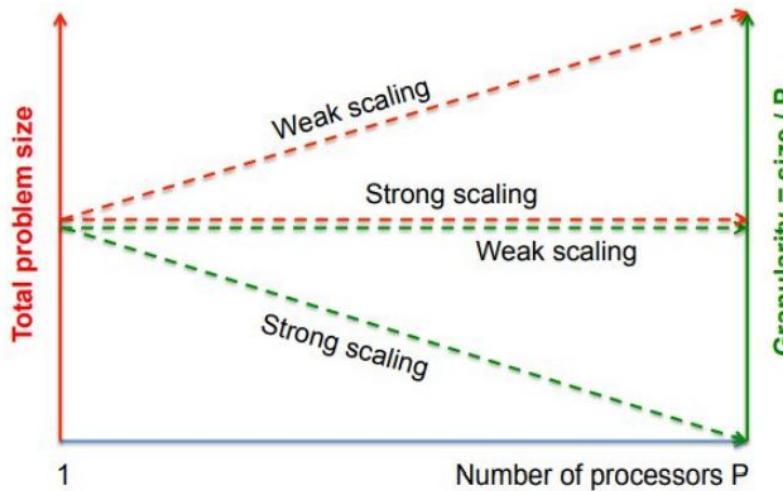


Strong vs Weak Scalability



Two usual scenarios to evaluate the scalability of one application:

- Increase the number of processors P with constant problem size (strong scaling → reduce the execution time)
- Increase the number of processors P with problem size proportional to P (weak scaling → solve larger problem)



Instructor Social Media

Youtube: Lucas Science



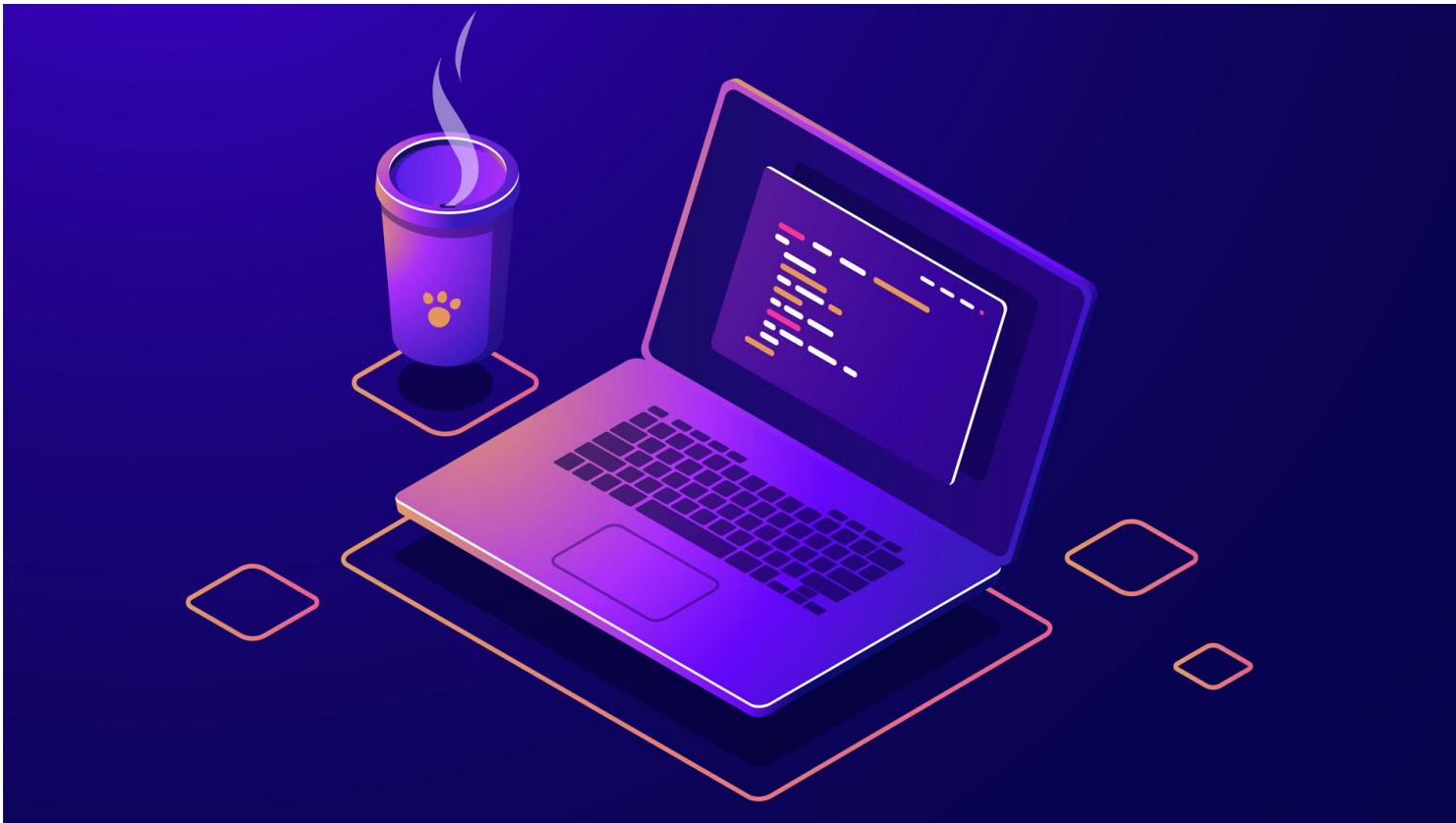
Instagram: lucaasbazilio



Twitter: lucasebazilio



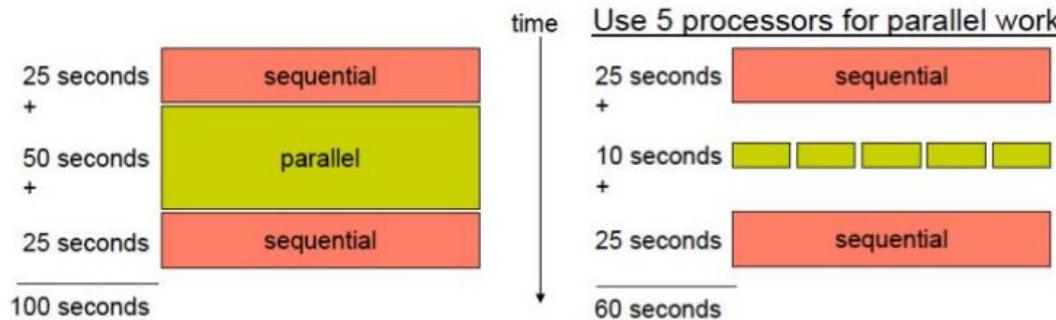
Amdahl's Law





Amdahl's Law

Performance improvement is limited by the fraction of time the program does not run in fully parallel mode

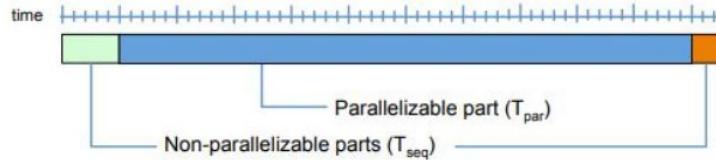


- Parallel part is 5 times faster: $Speedup_{parallel_part} = 50/10 = 5$
- Parallel version is just 1.67 times faster: $S_p = 100/60 = 1.67$, $E_p = 1.67/5 = 0.33$

Amdahl's Law



Assume the following simplified case, where the parallel fraction φ is the fraction, of total execution time, the program can be parallelized

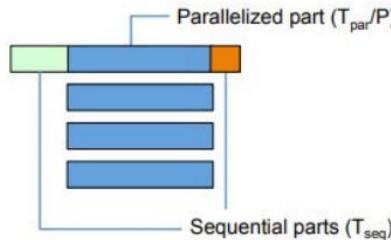


$$T_1 = T_{seq} + T_{par}$$

$$\varphi = T_{par}/T_1$$

$$T_{seq} = (1 - \varphi) \times T_1$$

$$T_{par} = \varphi \times T_1$$



$$T_1 = (1 - \varphi) \times T_1 + \varphi \times T_1$$

$$T_P = T_{seq} + T_{par}/P$$

$$T_P = (1 - \varphi) \times T_1 + (\varphi \times T_1/P)$$



Amdahl's Law

From where we can compute the speed-up S_P that can be achieved as

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1 - \varphi) \times T_1 + (\varphi \times T_1/P)}$$

$$S_p = \frac{1}{((1 - \varphi) + \varphi/P)}$$

Two particular cases:

$$\varphi = 0 \rightarrow S_p = 1$$

$$\varphi = 1 \rightarrow S_p = P$$

Amdahl's Law



When $P \rightarrow \infty$ the expression of the speed-up becomes

$$S_{p \rightarrow \infty} = \frac{1}{(1 - \varphi)}$$

Instructor Social Media

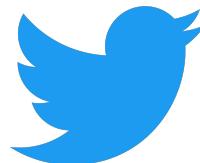
Youtube: Lucas Science



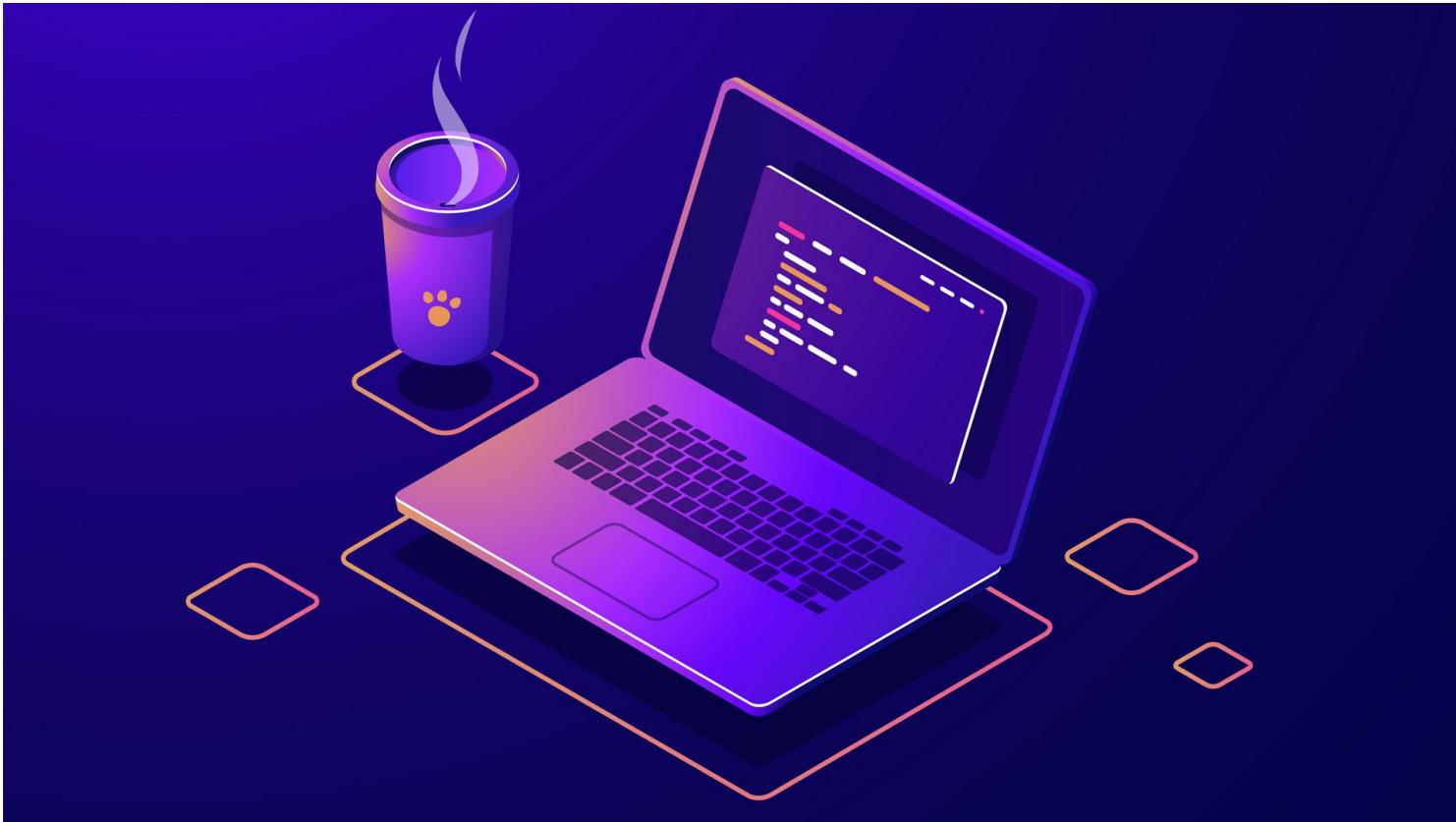
Instagram: lucaasbazilio



Twitter: lucasebazilio



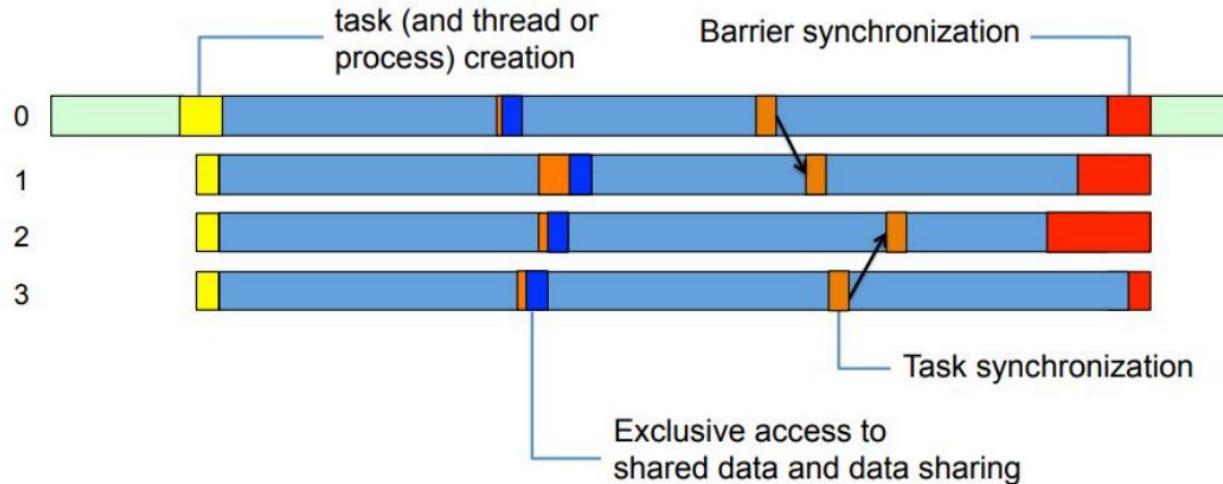
Overhead Sources



Overhead Sources



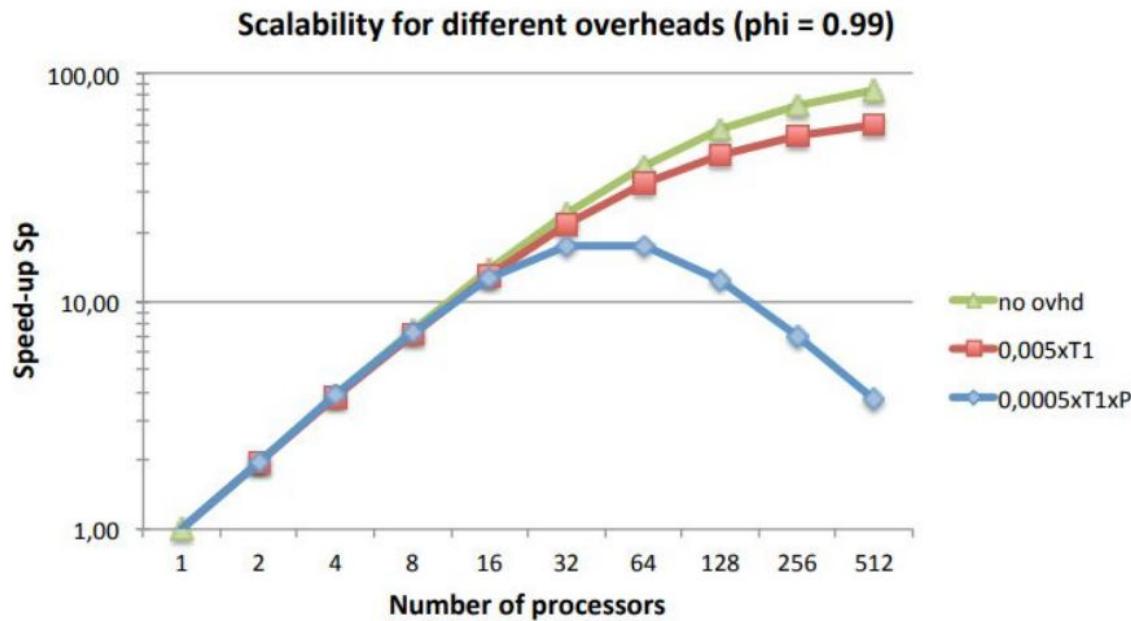
Parallel computing is not for free, we should account overheads
(i.e. any cost that gets added to a sequential computation so as to enable it to run in parallel)



Overhead Sources



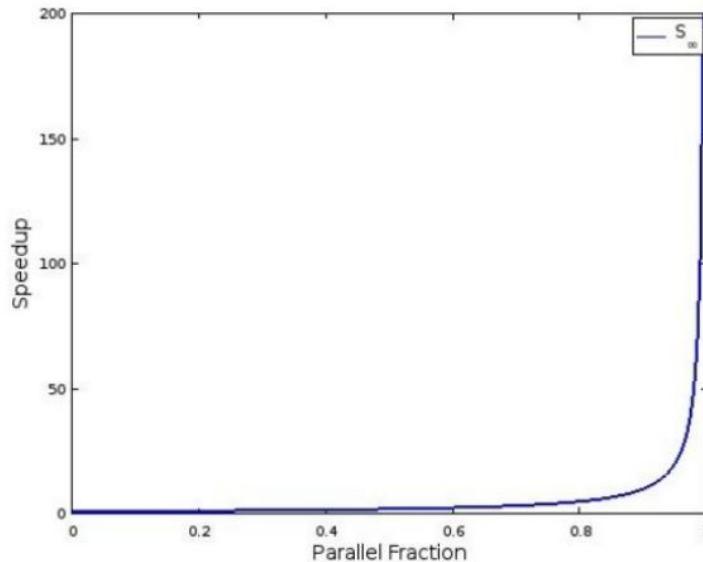
$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + \text{overhead}(p)$$



Amdahl's Law Conclusion



Amdahl's Law can be overly pessimistic:

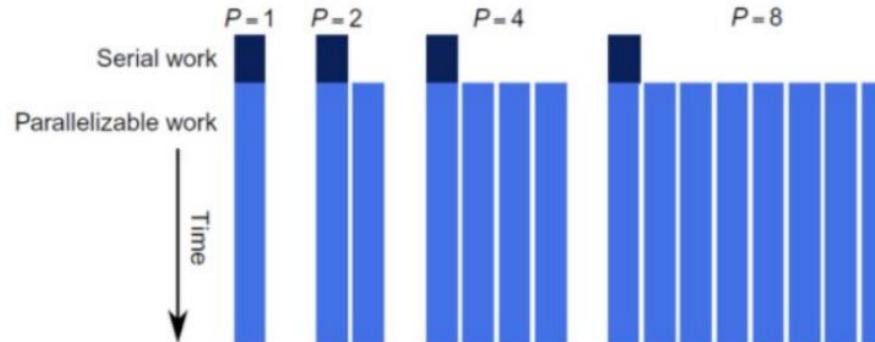


- Parallel processing might not be worthwhile if there is a large amount of inherently sequential code.



In practise

- The goal of applying parallelism is to increase the accuracy of the solution that can be computed in a fixed amount of time.
→ Treat time as constant and let problem size increase with P .
- The serial part grows slowly or remains fixed
→ Its proportion gets reduced as the problem size increases.



- Speedup grows as workers are added and the problem size is increased. (Weak Scaling).

Instructor Social Media

Youtube: Lucas Science



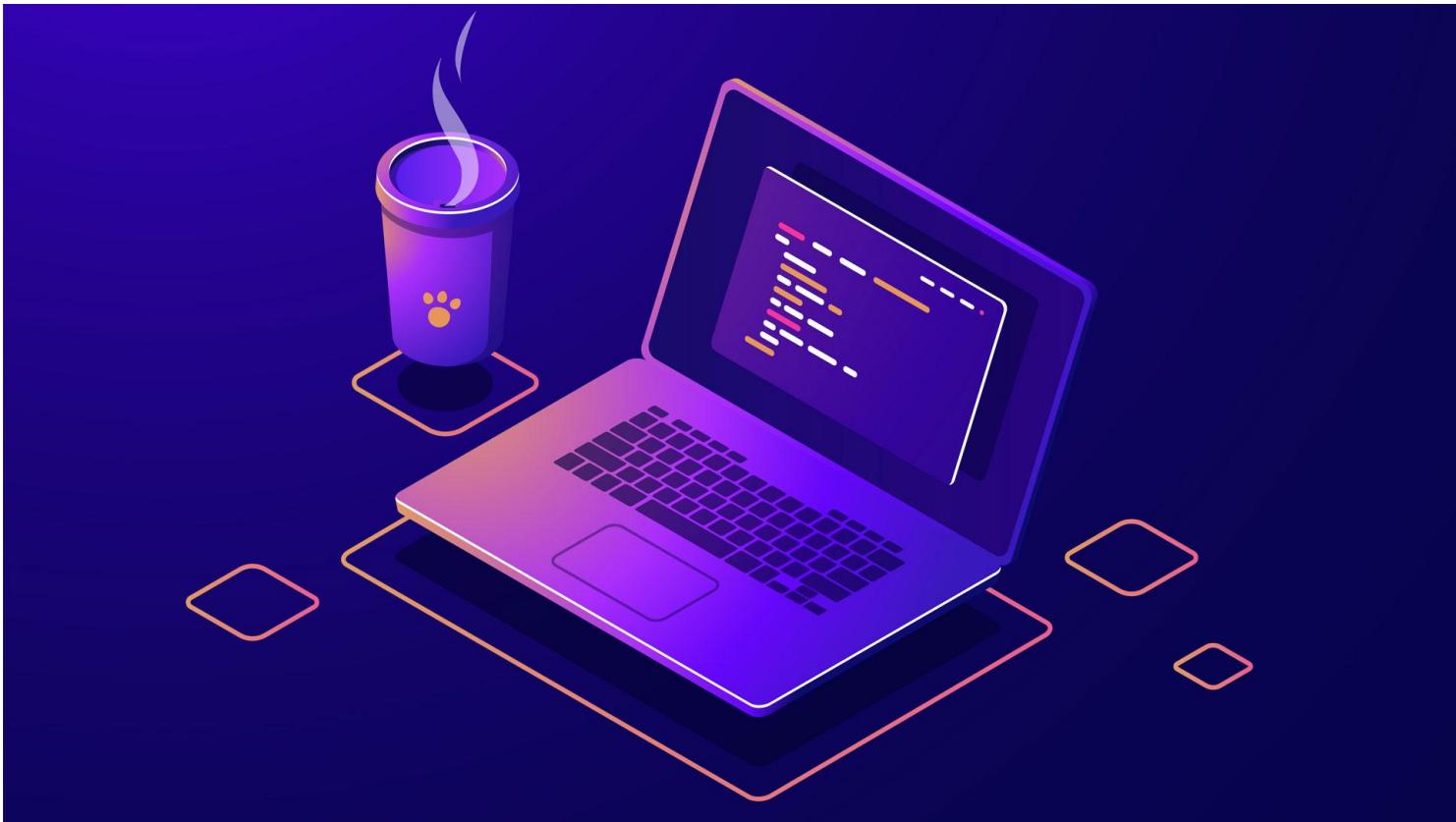
Instagram: lucaasbazilio



Twitter: lucasebazilio



Common Overheads



Common Overheads

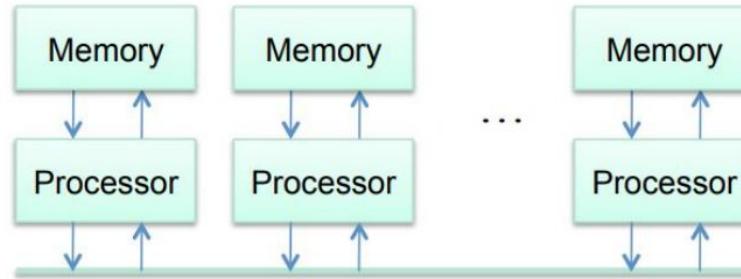


- ▶ **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)
- ▶ **Idleness:** thread cannot find any useful work to execute (e.g. dependences, load imbalance, poor communication and computation overlap or hiding of memory latencies, ...)
- ▶ **Computation:** extra work added to obtain a parallel algorithm (e.g. replication)
- ▶ **Memory:** extra memory used to obtain a parallel algorithm (e.g. impact on memory hierarchy, ...)
- ▶ **Contention:** competition for the access to shared resources (e.g. memory, network)

How to model data sharing overhead?



We start with a simple architectural model in which each processor P_i has its own memory, interconnected with the other processors through an interconnection network.



- ▶ Processors access to local data (in its own memory) using regular load/store instructions
- ▶ We will assume that local accesses take zero overhead.

How to model data sharing overhead?



- ▶ Processors can access remote data (in other processors) using a message-passing model (remote load instruction²)
- ▶ To model the time needed to access remote data we will use two components:
 - ▶ Start up: time spent in preparing the remote access (t_s)
 - ▶ Transfer: time spent in transferring the message (number of bytes m , time per byte t_w) from/to the remote location

$$t_{access} = t_s + m \times t_w$$

- ▶ Synchronization between the two processors involved may be necessary to guarantee that the data is available

How to model data sharing overhead?



Assumptions (to make simpler the model)

- ▶ At a given moment, a processor P_i can only perform one remote memory access to another processor P_j
- ▶ At a given moment, a processor P_i can only serve one remote memory access from another processor P_k
- ▶ Both actions can be performed simultaneously: $P_i \rightarrow P_j$ and $P_i \leftarrow P_k$

Instructor Social Media

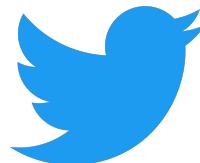
Youtube: Lucas Science



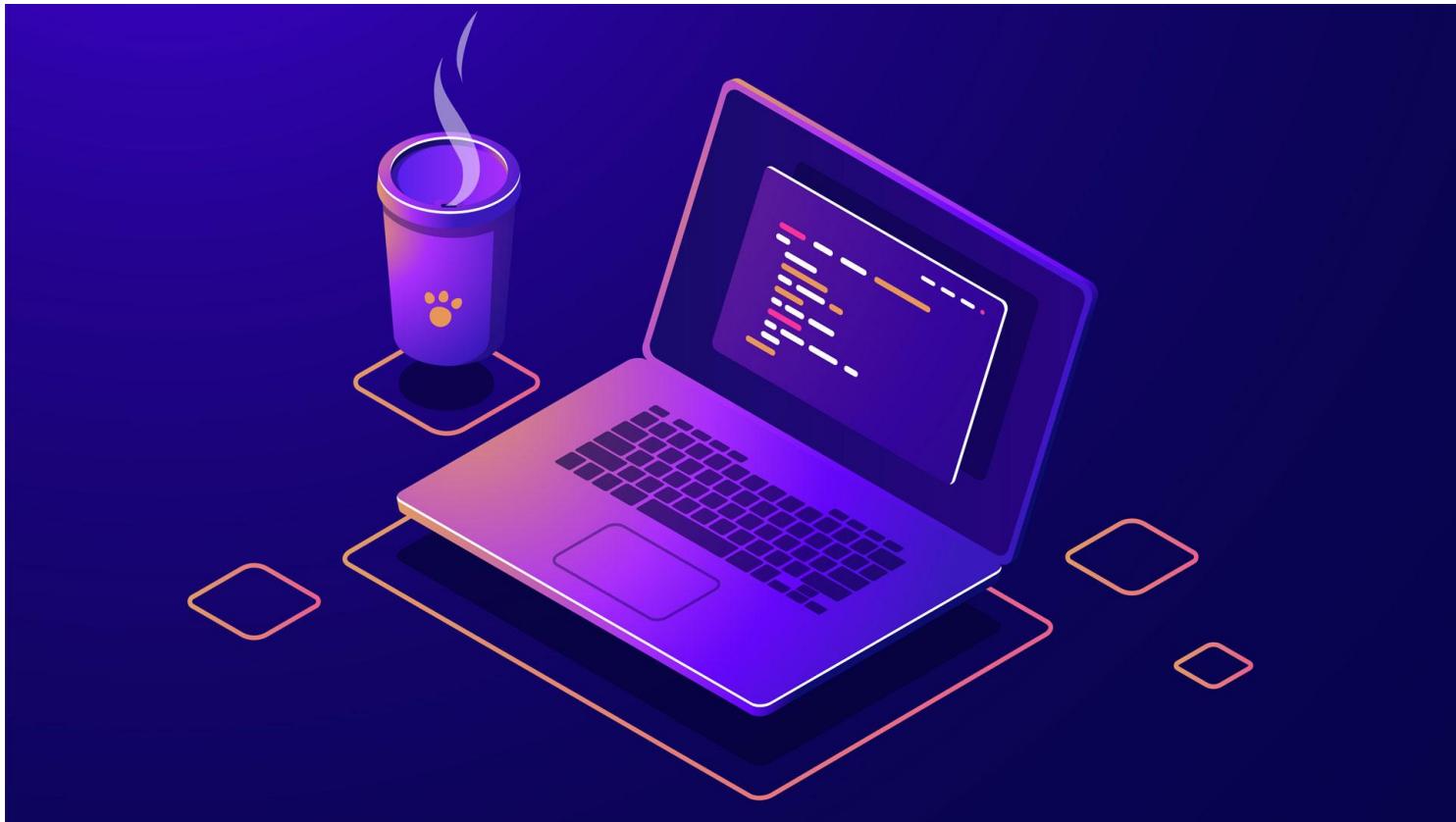
Instagram: lucaasbazilio



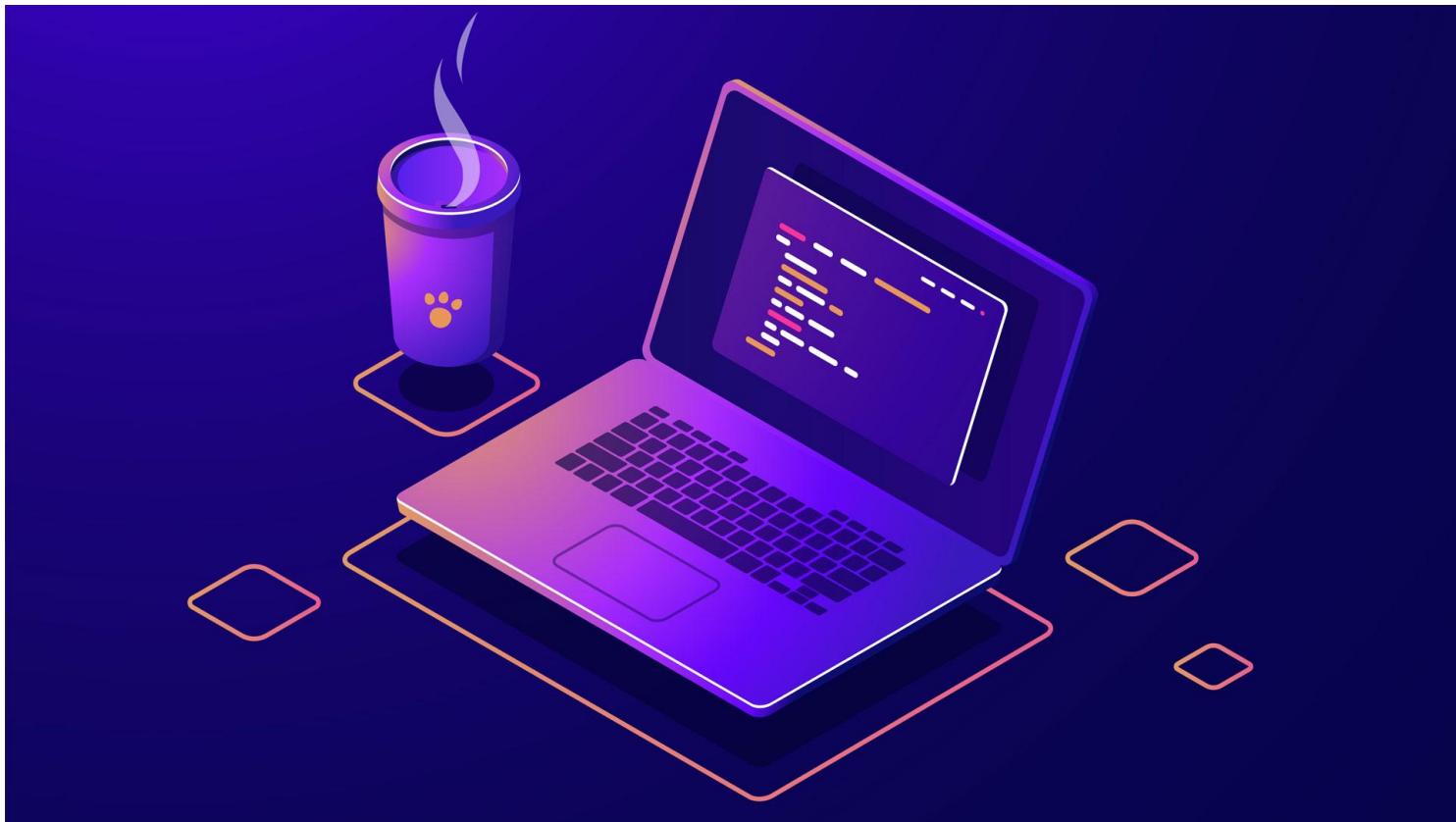
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 1



Problem 1

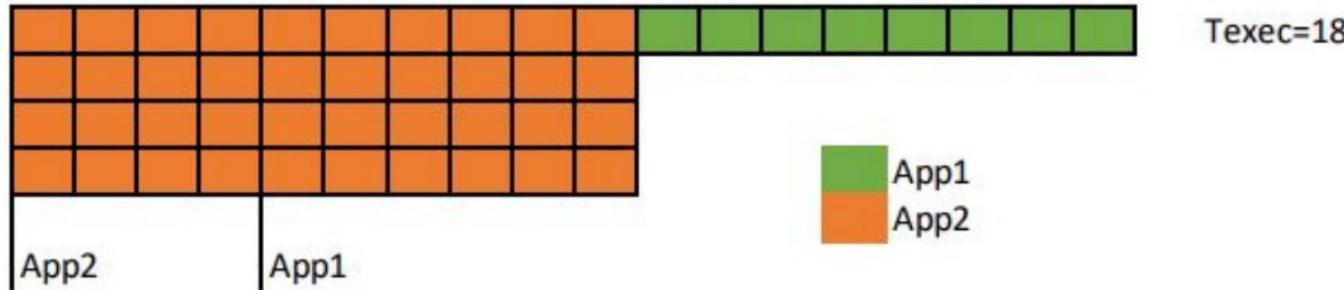


1. Assume we want to execute two different applications in our parallel machine with 4 processors: application $App1$ is sequential; $App2$ is parallelised defining 4 tasks, each task executing one fourth of the total application. The sequential time for the applications is 8 and 40 time units, respectively. Assuming: that $App1$ starts its execution at time 4 and $App2$ starts at time 0, draw a time line showing how they will be executed if the operating system:
 - (a) Does not allow multiprogramming, i.e. only one application can be executed at the same time in the system.
 - (b) Allows multiprogramming so that the system tries to have both applications running concurrently, each application making use of the number of processors is able to use.
 - (c) The same as in the second case, but now $App2$ is parallelised defining 3 tasks, each task executing one third of the total application.



Point a)

a)



Instructor Social Media

Youtube: Lucas Science



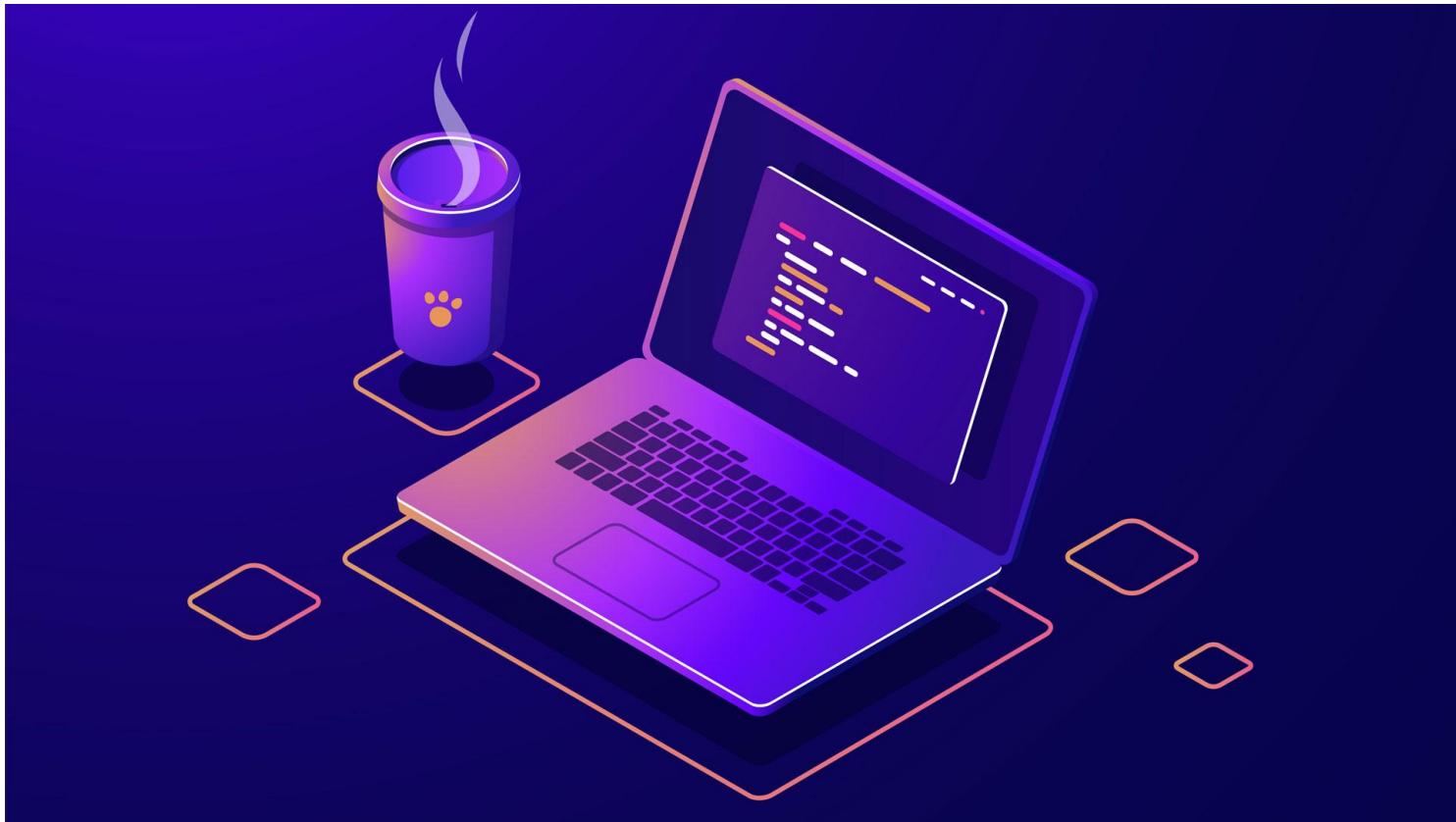
Instagram: lucaasbazilio



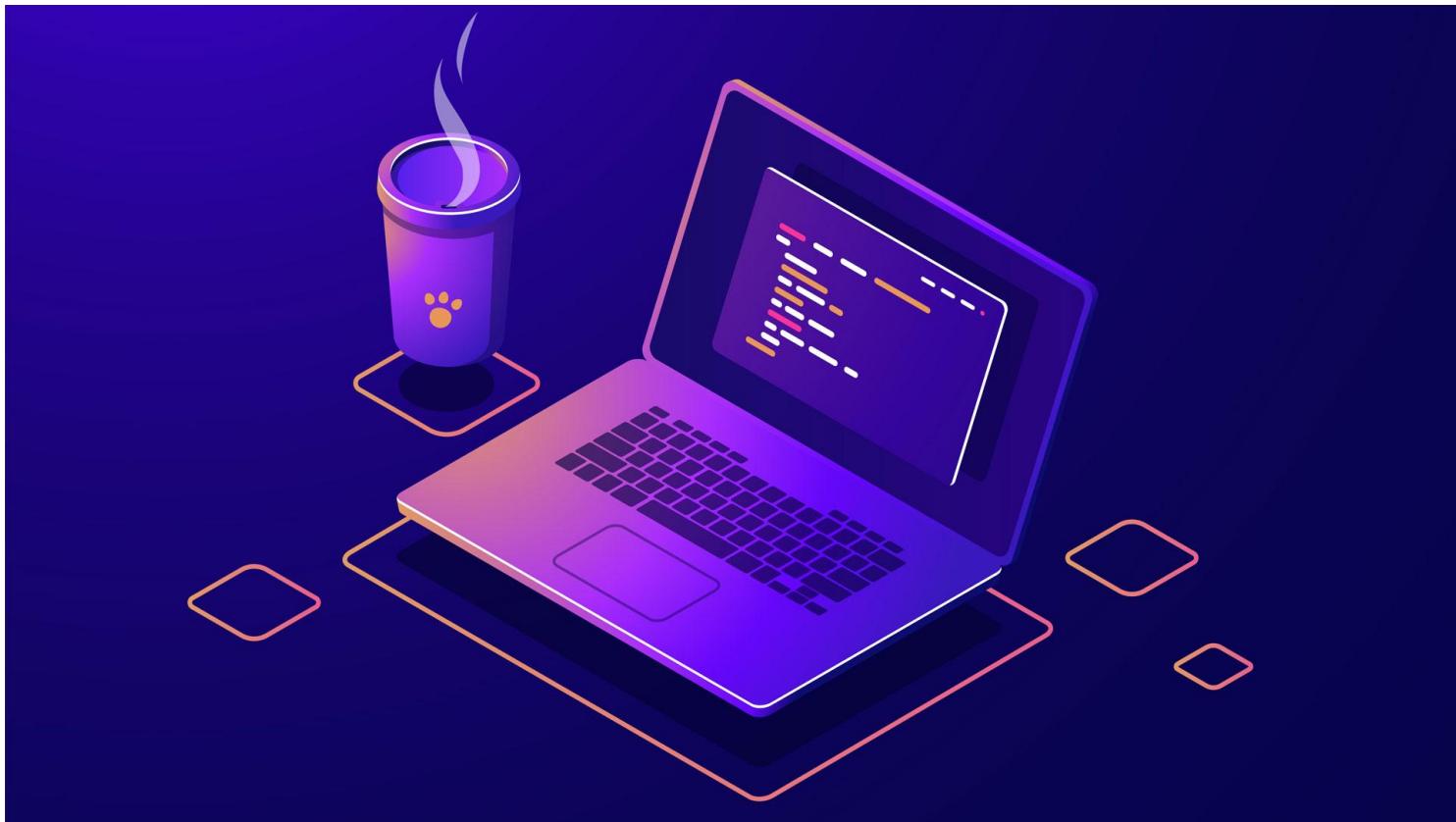
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 1



Problem 1

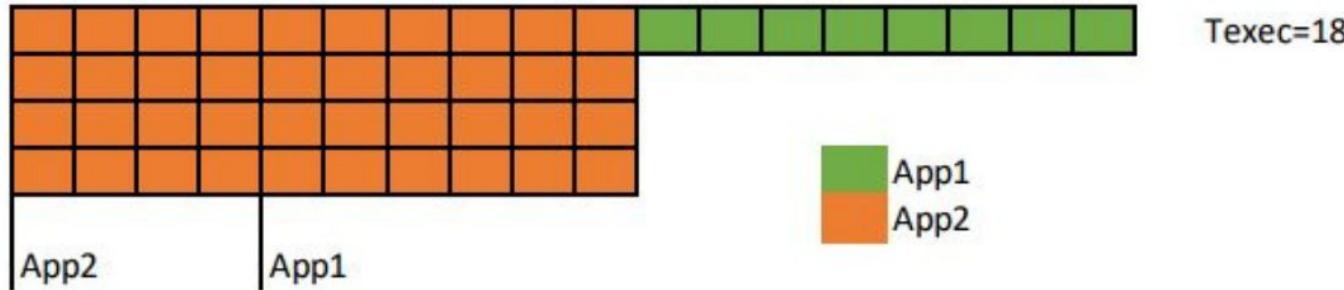


1. Assume we want to execute two different applications in our parallel machine with 4 processors: application $App1$ is sequential; $App2$ is parallelised defining 4 tasks, each task executing one fourth of the total application. The sequential time for the applications is 8 and 40 time units, respectively. Assuming: that $App1$ starts its execution at time 4 and $App2$ starts at time 0, draw a time line showing how they will be executed if the operating system:
 - (a) Does not allow multiprogramming, i.e. only one application can be executed at the same time in the system.
 - (b) Allows multiprogramming so that the system tries to have both applications running concurrently, each application making use of the number of processors is able to use.
 - (c) The same as in the second case, but now $App2$ is parallelised defining 3 tasks, each task executing one third of the total application.



Point a)

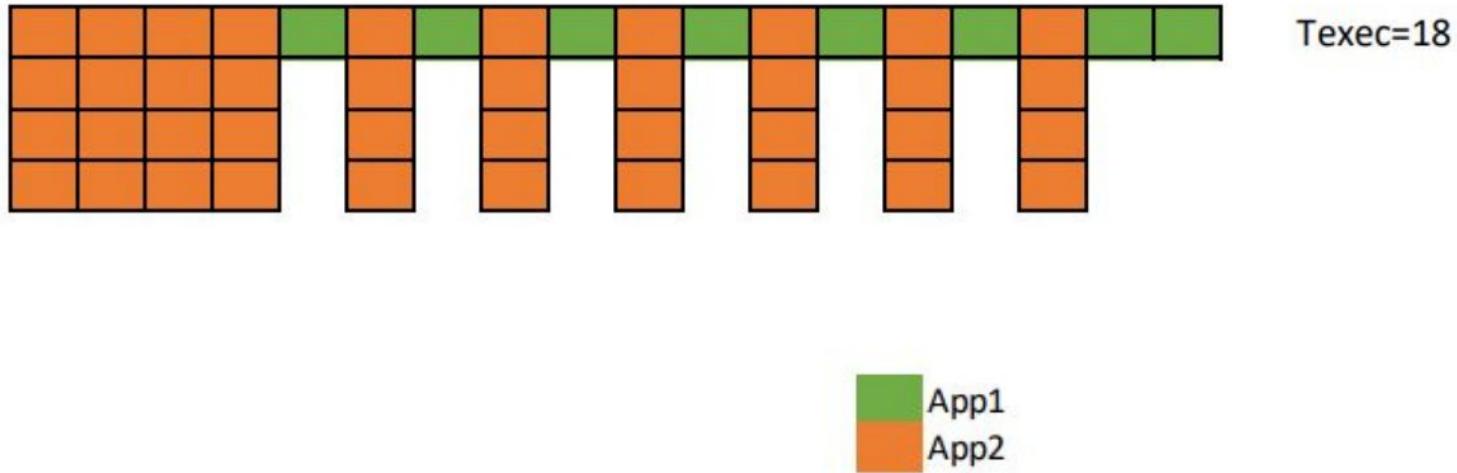
a)





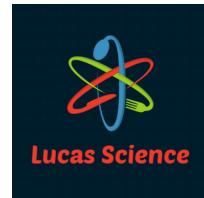
Point b)

b)



Instructor Social Media

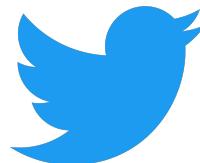
Youtube: Lucas Science



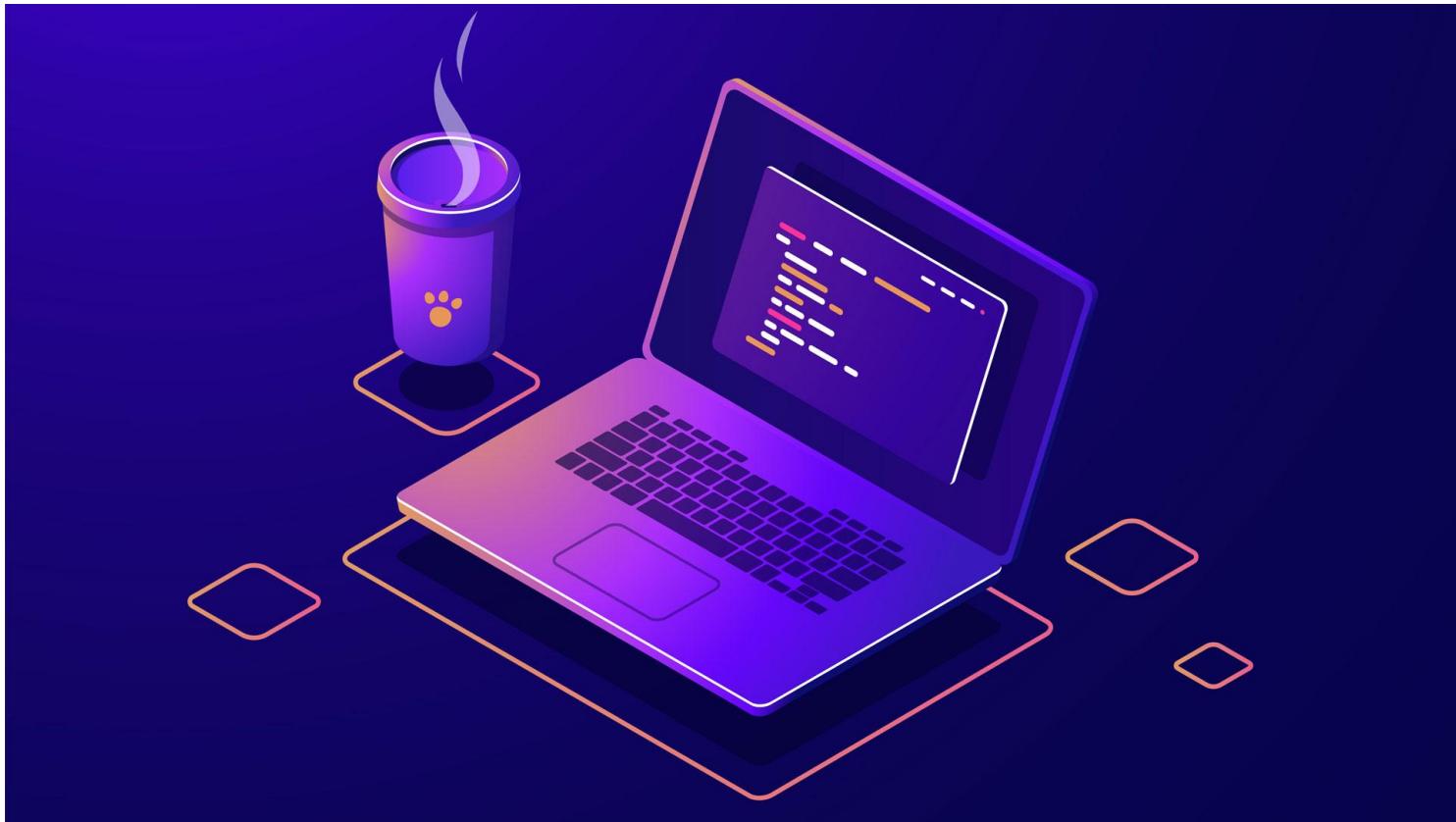
Instagram: lucaasbazilio



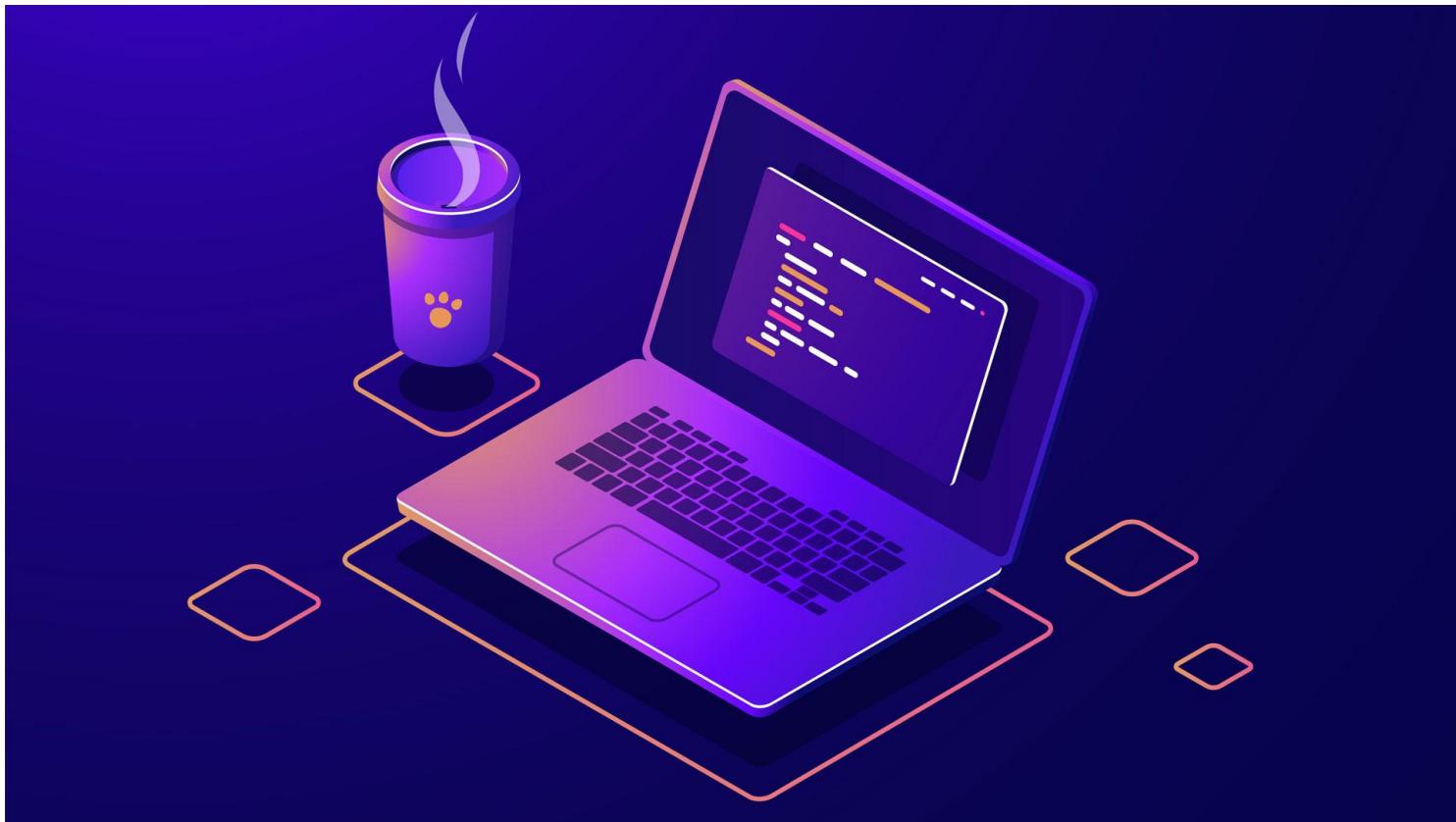
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 1



Problem 1

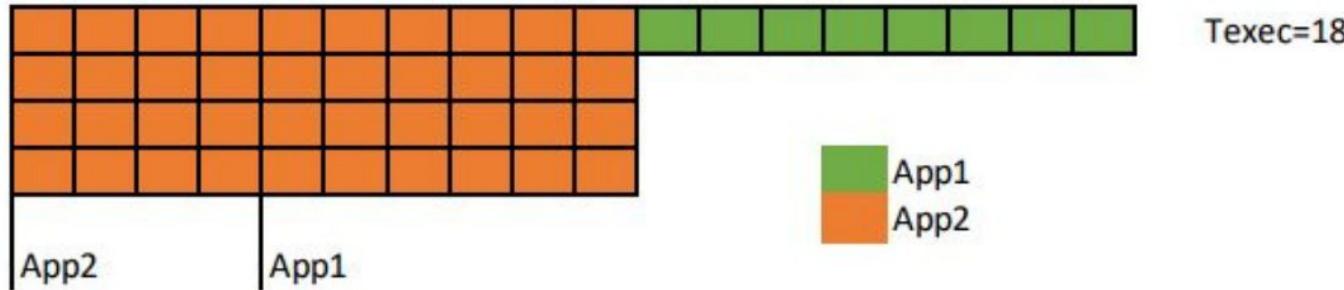


1. Assume we want to execute two different applications in our parallel machine with 4 processors: application $App1$ is sequential; $App2$ is parallelised defining 4 tasks, each task executing one fourth of the total application. The sequential time for the applications is 8 and 40 time units, respectively. Assuming: that $App1$ starts its execution at time 4 and $App2$ starts at time 0, draw a time line showing how they will be executed if the operating system:
 - (a) Does not allow multiprogramming, i.e. only one application can be executed at the same time in the system.
 - (b) Allows multiprogramming so that the system tries to have both applications running concurrently, each application making use of the number of processors is able to use.
 - (c) The same as in the second case, but now $App2$ is parallelised defining 3 tasks, each task executing one third of the total application.



Point a)

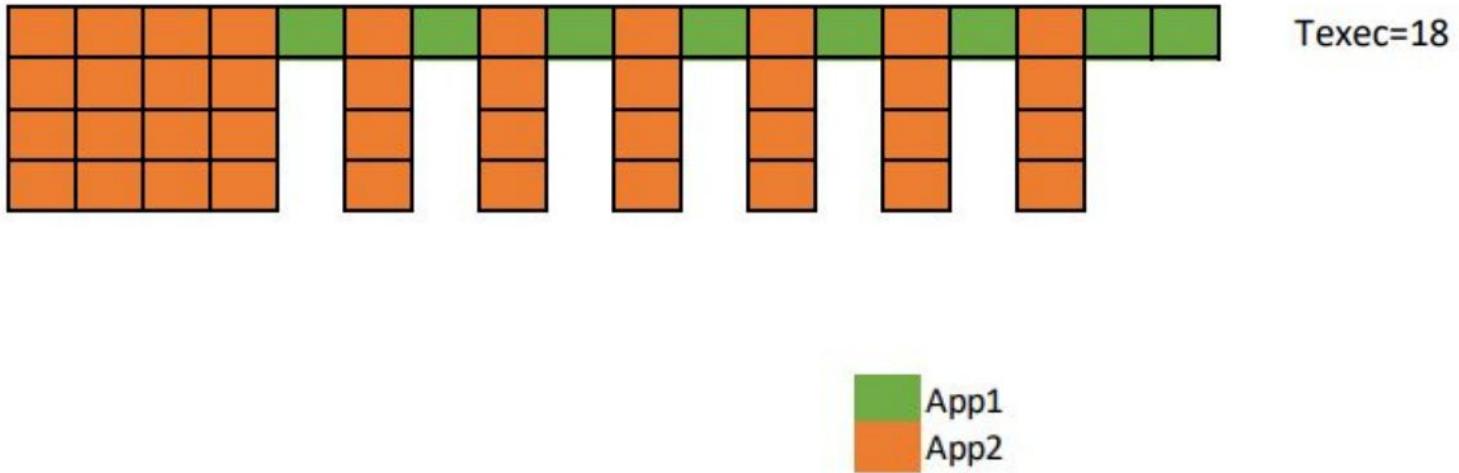
a)





Point b)

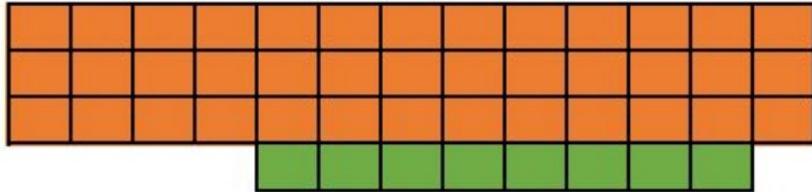
b)



Point c)



c)



$$T_{exec} = 40/3 = 13.33$$



Instructor Social Media

Youtube: Lucas Science



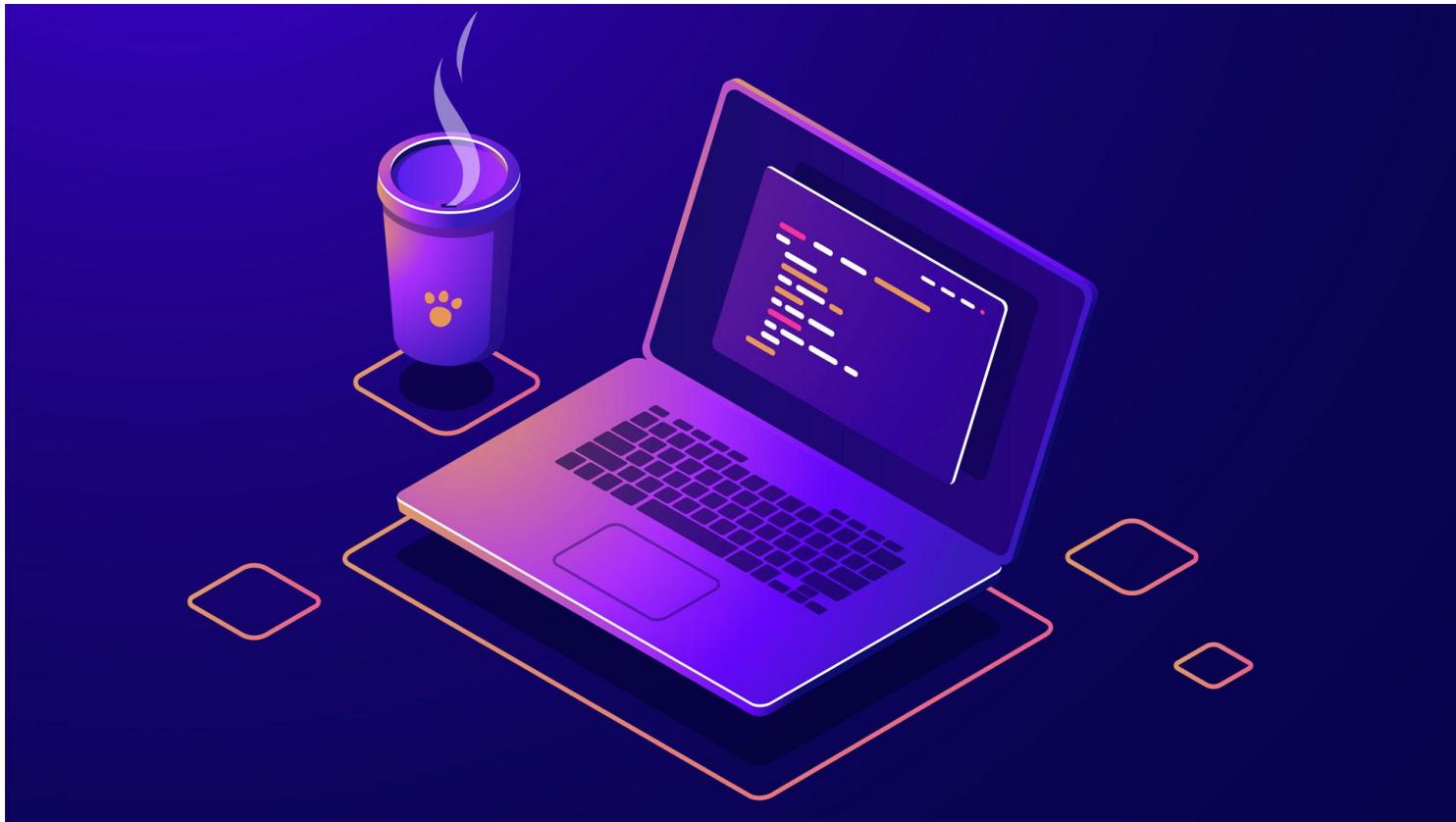
Instagram: lucaasbazilio



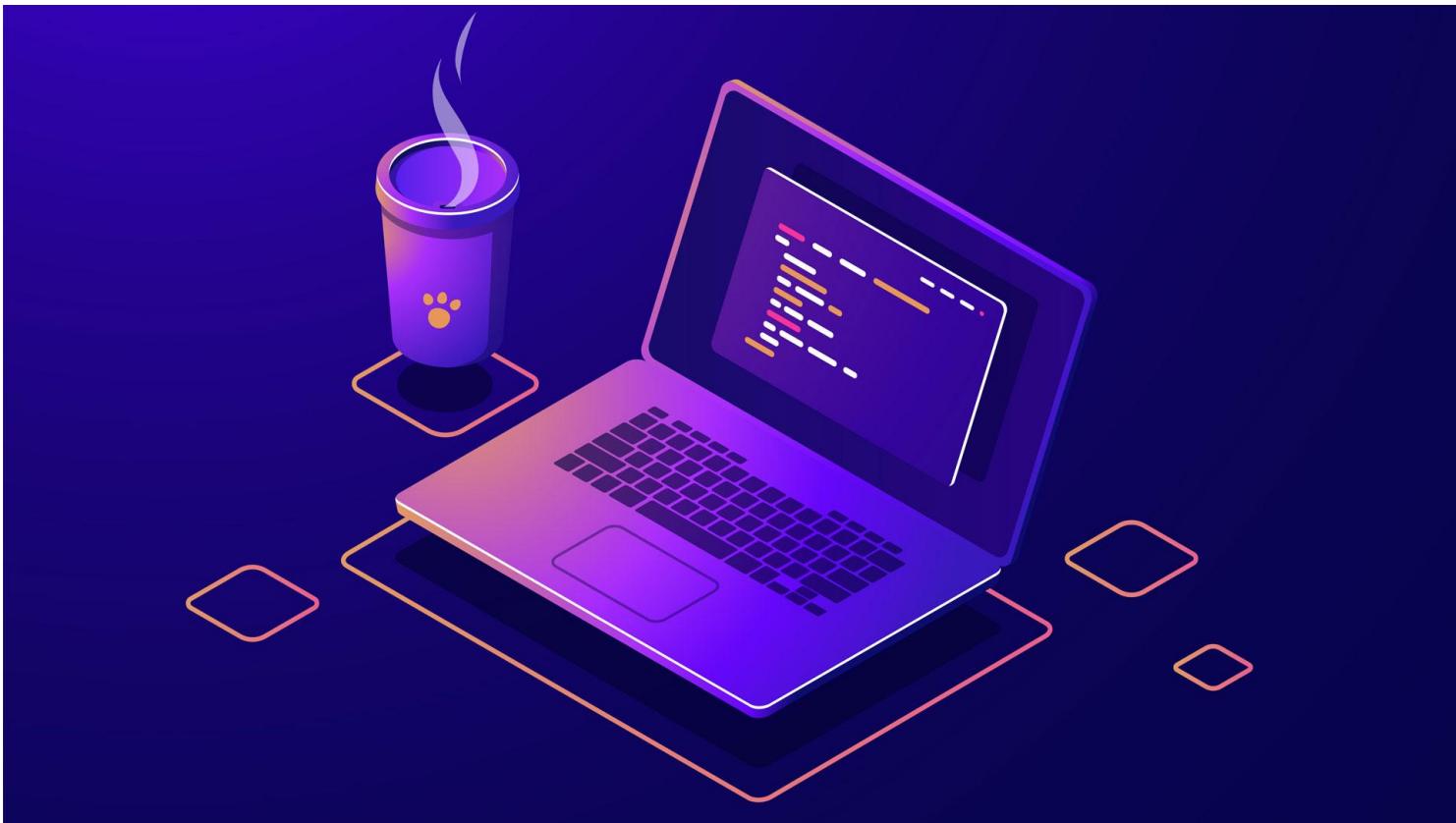
Twitter: lucasebazilio



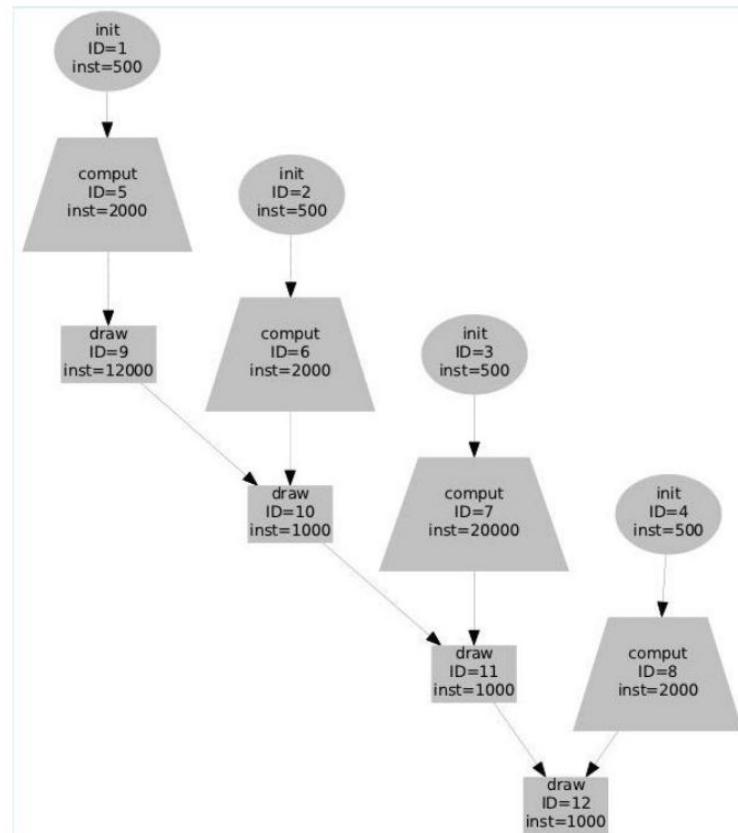
Parallelism Fundamentals Problems



Problem 2



Problem 2



Problem 2



a) Calculate T_1

b) Critical Path

c) Calculate T_∞

d) Parallelism and minimum number of processors (P_{\min})

Solutions



$$a) T_1 = 43000$$

Solutions



a) $T_1 = 43000$

b) Critical Path = {3, 7, 11, 12}

Solutions



a) $T_1 = 43000$

b) Critical Path = {3, 7, 11, 12}

c) $T_{\infty} = 22500$

Solutions



a) $T_1 = 43000$

b) Critical Path = {3, 7, 11, 12}

c) $T_{\infty} = 22500$

d) Parallelism = $43000/22500 = 1.91$

Hence, minimum number of processors $P_{\min} = 2$

Instructor Social Media

Youtube: Lucas Science



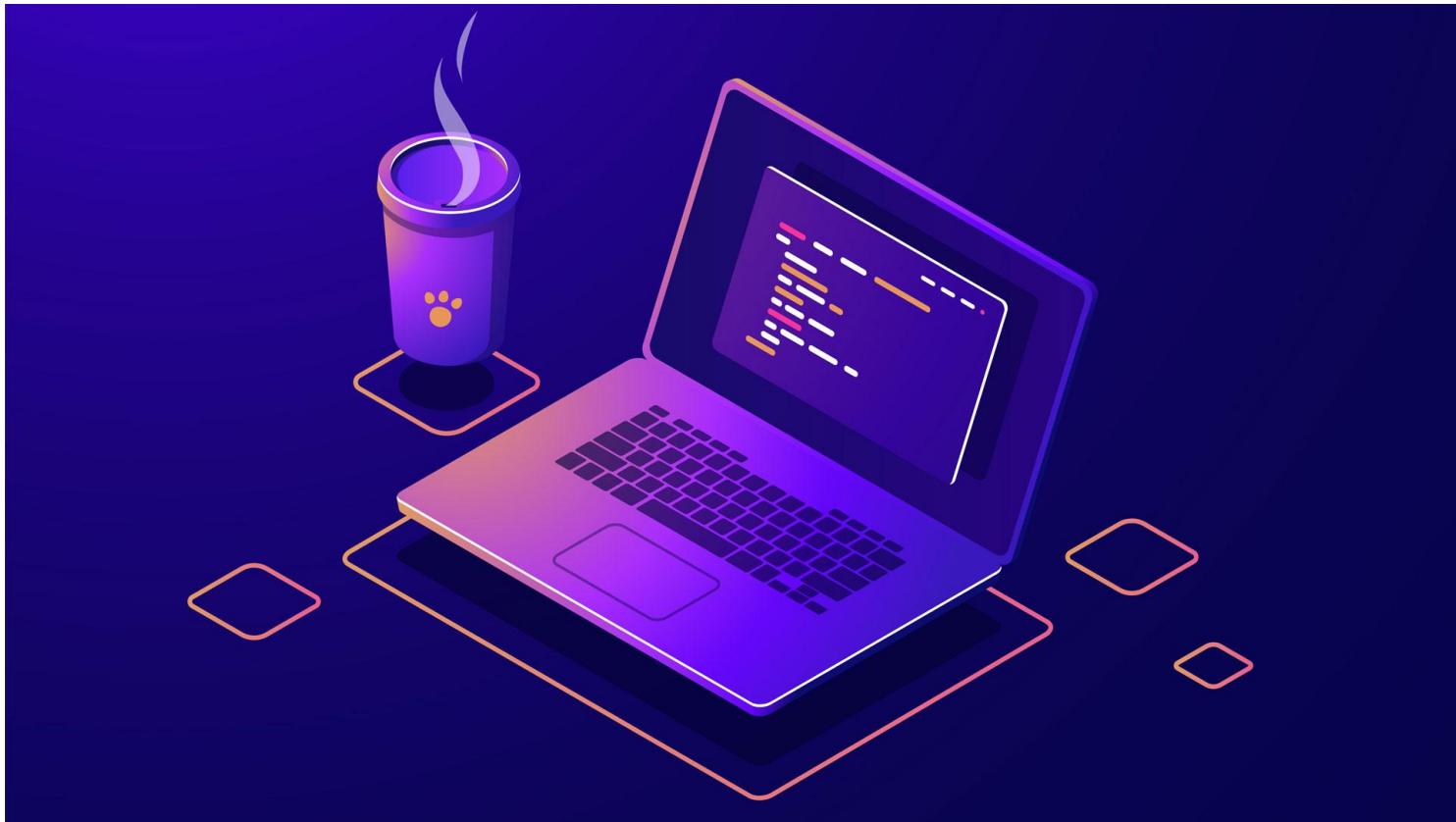
Instagram: lucaasbazilio



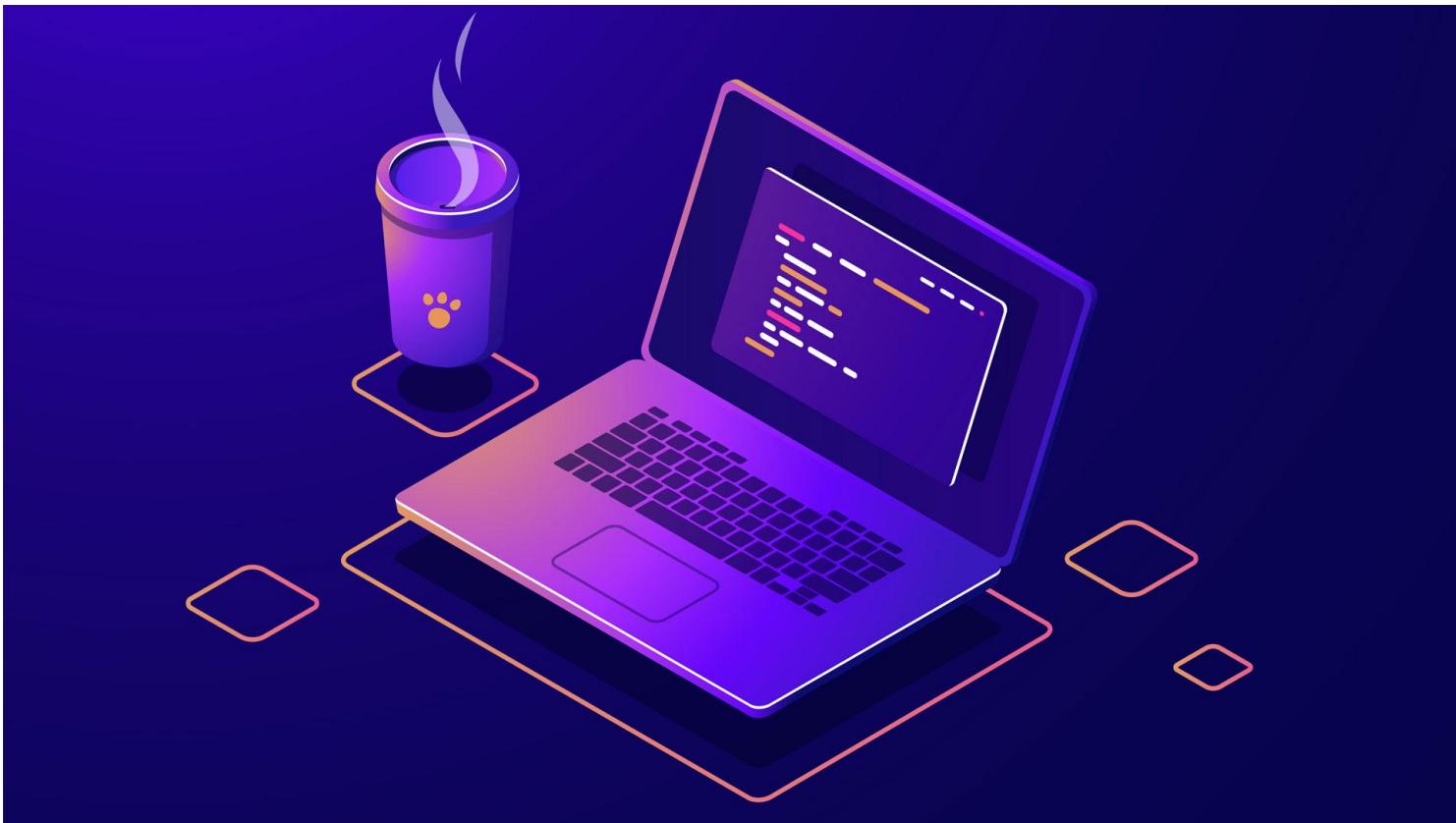
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 3





```
# define MAX 8
// initialization
for (outer = 0; outer < MAX; outer++) {
    start_task("for-initialize");
    for (inner = 0; inner < MAX; inner++)
        matrix[outer][inner] = inner;
    end_task("for-initialize");
}
// computation
for (outer = 0; outer < MAX; outer++) {
    start_task("for-compute");
    for (inner = 0; inner <= outer; inner++)
        matrix[outer][inner] = matrix[outer][inner] + foo(outer,inner);
    end_task("for-compute");
}
```

Problem 3



Assuming that: 1) in the initialization loop the execution of each iteration of the internal loop lasts 10 cycles; 2) in the computation loop the execution of each iteration of the internal loop lasts 100 cycles; and 3) the execution of the **foo** function does not cause any kind of dependence. **We ask:**

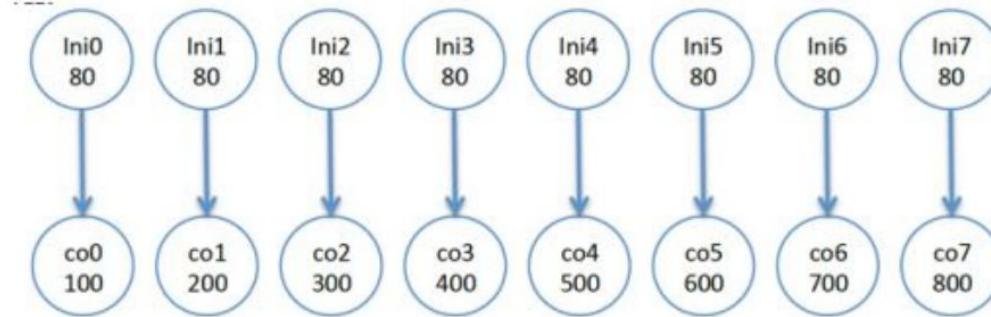
- a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time.
- b) Calculate the values for T_1 and T_∞ as well as the potential *Parallelism*.
- c) Calculate which is the best value for the "speed-up" on 4 processors (S_4), indicating which would be the proper task mapping (assignment) to processors to achieve it.

Solutions



a)

Task Dependence Graph:



Instructor Social Media

Youtube: Lucas Science



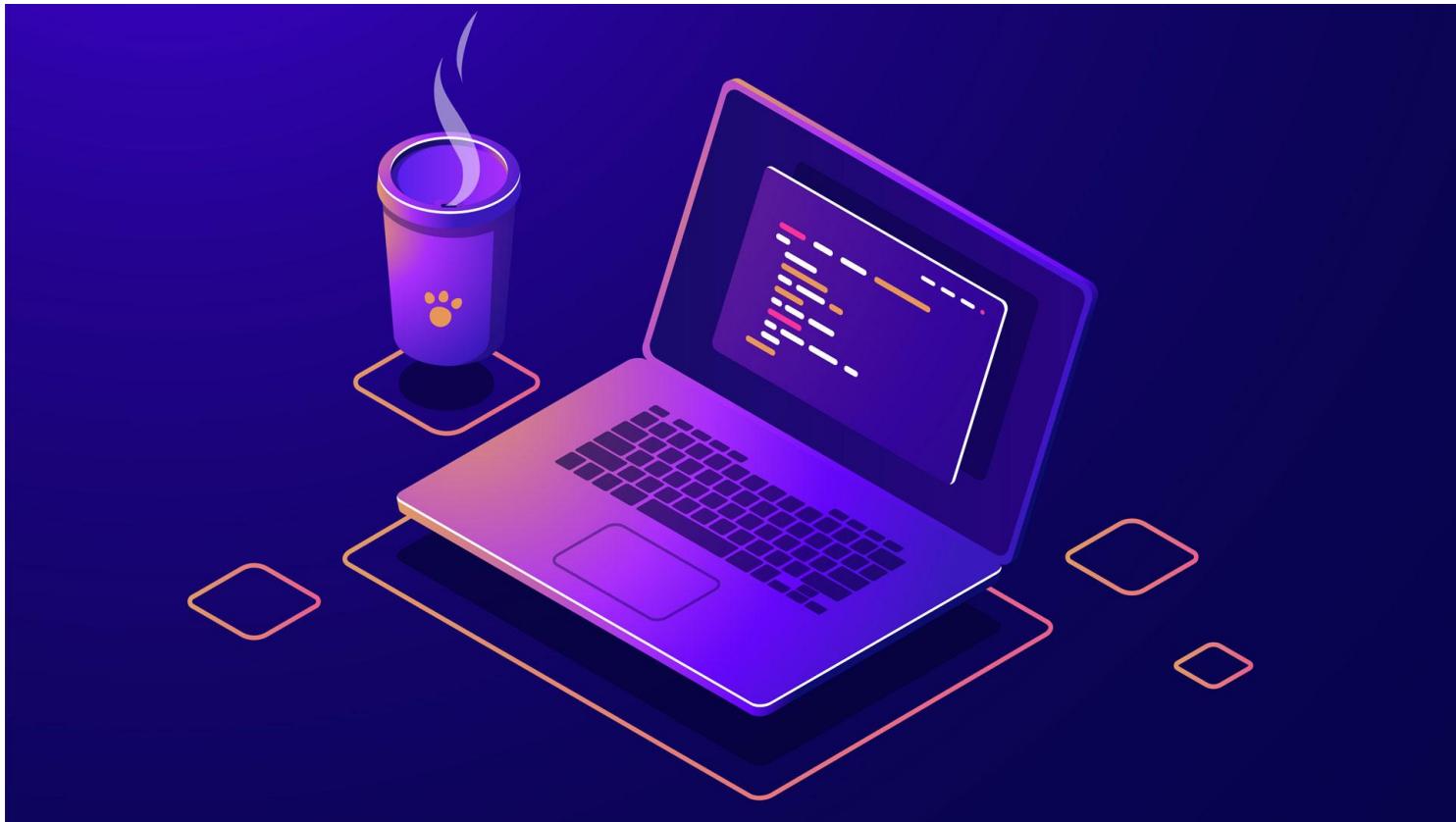
Instagram: lucaasbazilio



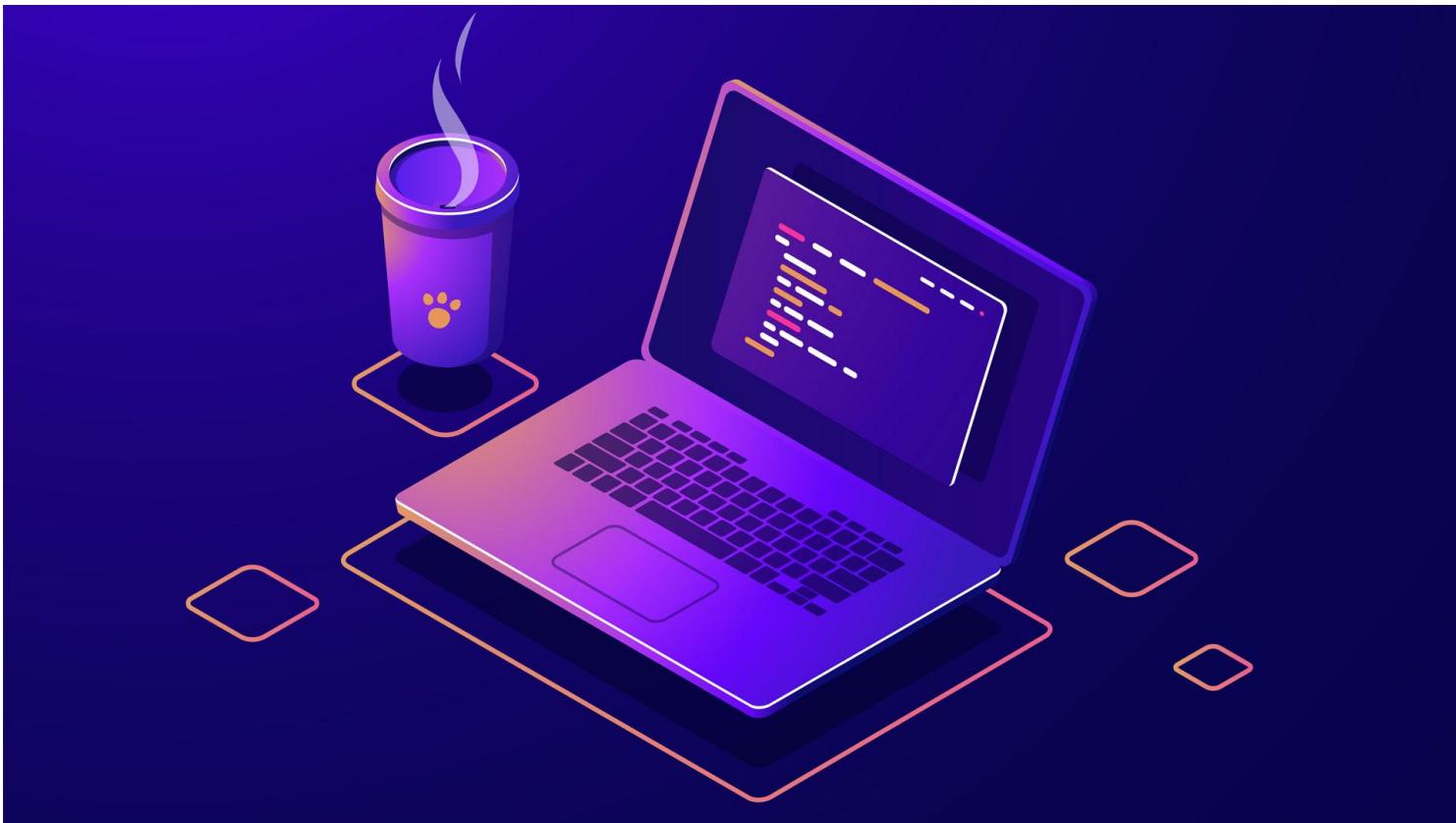
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 3





```
# define MAX 8
// initialization
for (outer = 0; outer < MAX; outer++) {
    start_task("for-initialize");
    for (inner = 0; inner < MAX; inner++)
        matrix[outer][inner] = inner;
    end_task("for-initialize");
}
// computation
for (outer = 0; outer < MAX; outer++) {
    start_task("for-compute");
    for (inner = 0; inner <= outer; inner++)
        matrix[outer][inner] = matrix[outer][inner] + foo(outer,inner);
    end_task("for-compute");
}
```

Problem 3



Assuming that: 1) in the initialization loop the execution of each iteration of the internal loop lasts 10 cycles; 2) in the computation loop the execution of each iteration of the internal loop lasts 100 cycles; and 3) the execution of the **foo** function does not cause any kind of dependence. **We ask:**

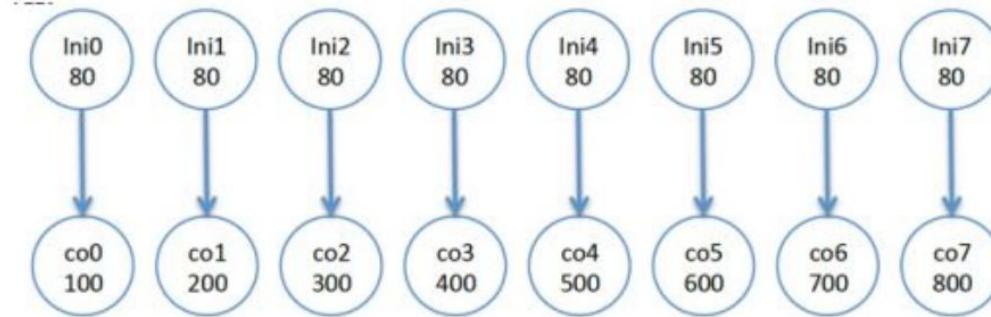
- a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time.
- b) Calculate the values for T_1 and T_∞ as well as the potential *Parallelism*.
- c) Calculate which is the best value for the "speed-up" on 4 processors (S_4), indicating which would be the proper task mapping (assignment) to processors to achieve it.

Solutions



a)

Task Dependence Graph:



Solutions



b)

$$T_1 = 4240$$

$$T_{\infty} = 880$$

$$\text{Parallelism} = T_1 / T_{\infty} = 4.81$$

$$\text{So } P_{\min} = 5$$

Instructor Social Media

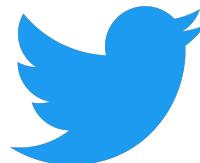
Youtube: Lucas Science



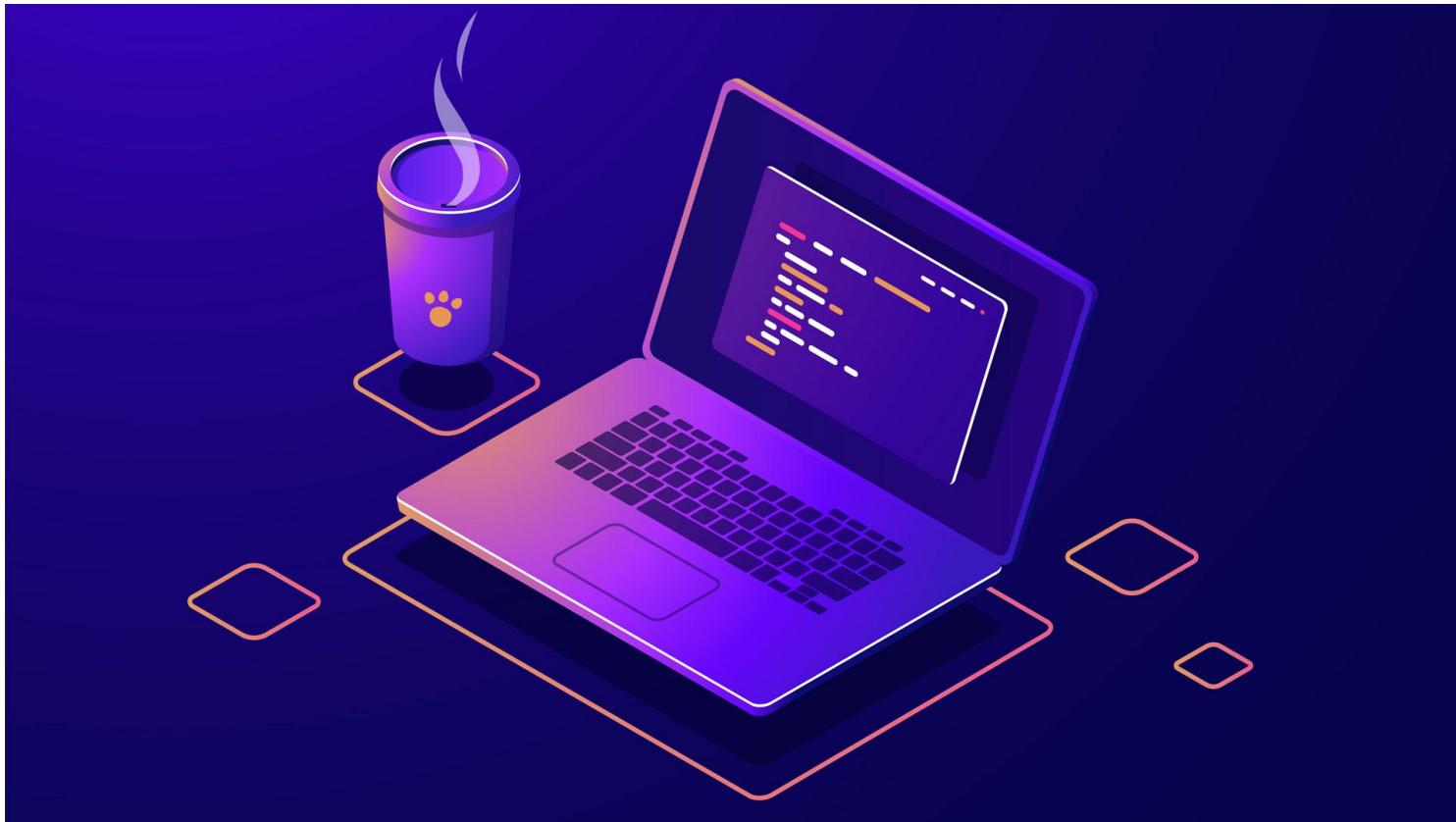
Instagram: lucaasbazilio



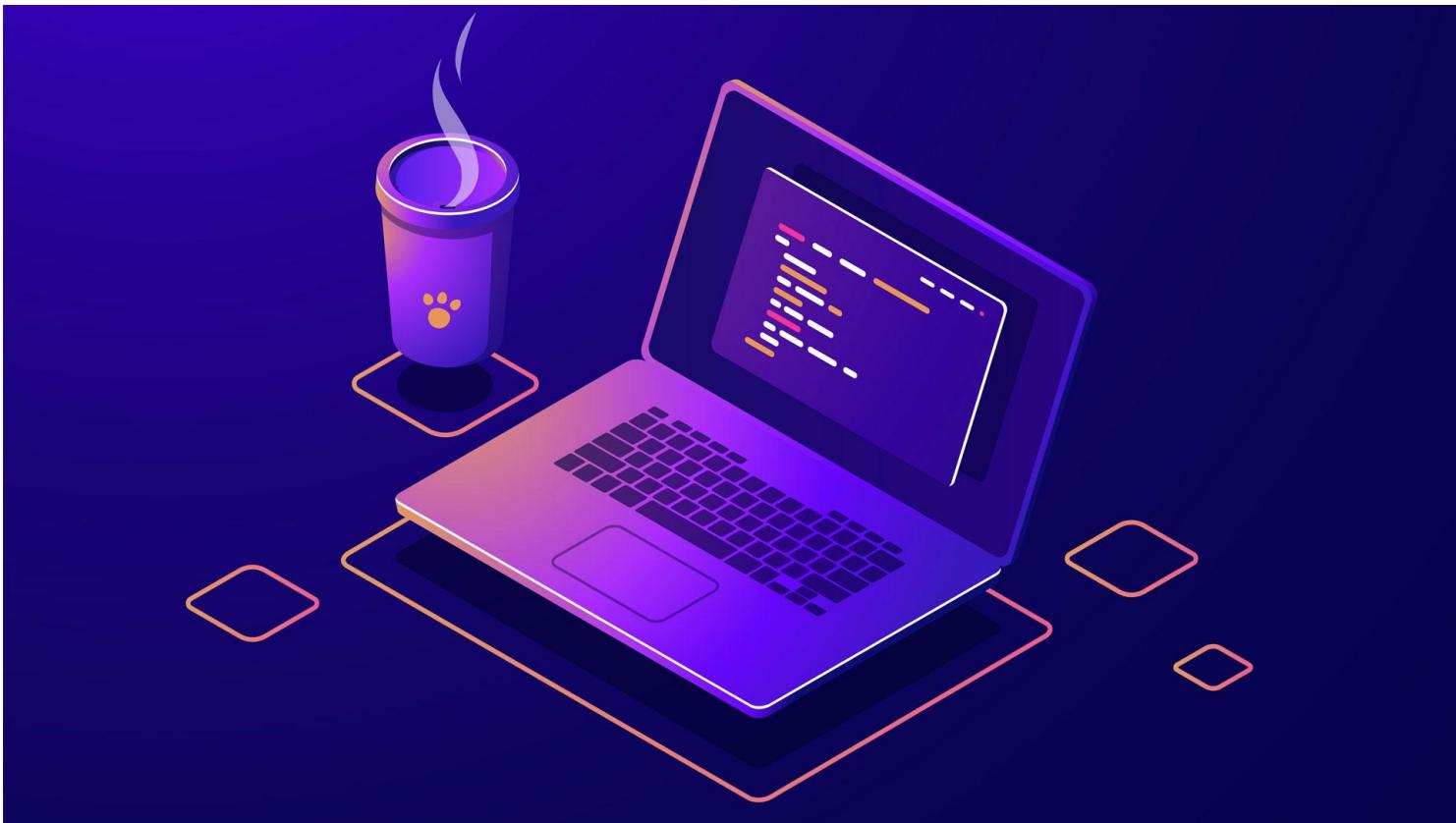
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 3





```
# define MAX 8
// initialization
for (outer = 0; outer < MAX; outer++) {
    start_task("for-initialize");
    for (inner = 0; inner < MAX; inner++)
        matrix[outer][inner] = inner;
    end_task("for-initialize");
}
// computation
for (outer = 0; outer < MAX; outer++) {
    start_task("for-compute");
    for (inner = 0; inner <= outer; inner++)
        matrix[outer][inner] = matrix[outer][inner] + foo(outer,inner);
    end_task("for-compute");
}
```

Problem 3



Assuming that: 1) in the initialization loop the execution of each iteration of the internal loop lasts 10 cycles; 2) in the computation loop the execution of each iteration of the internal loop lasts 100 cycles; and 3) the execution of the **foo** function does not cause any kind of dependence. **We ask:**

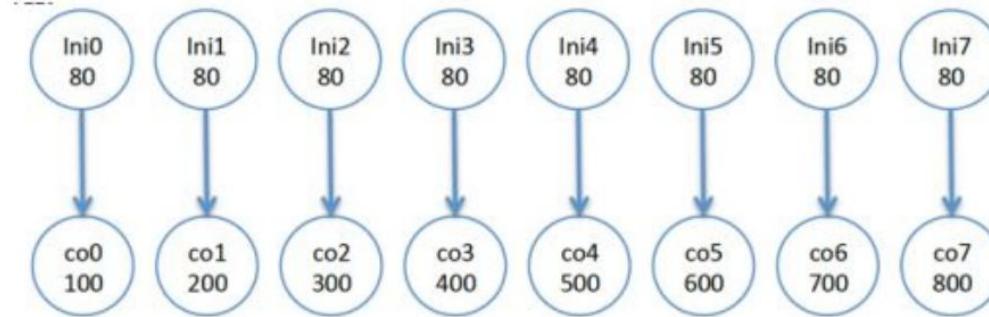
- a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time.
- b) Calculate the values for T_1 and T_∞ as well as the potential *Parallelism*.
- c) Calculate which is the best value for the "speed-up" on 4 processors (S_4), indicating which would be the proper task mapping (assignment) to processors to achieve it.

Solutions



a)

Task Dependence Graph:



Solutions



b)

$$T_1 = 4240$$

$$T_{\infty} = 880$$

$$\text{Parallelism} = T_1 / T_{\infty} = 4.81$$

$$\text{So } P_{\min} = 5$$



Solutions

c)



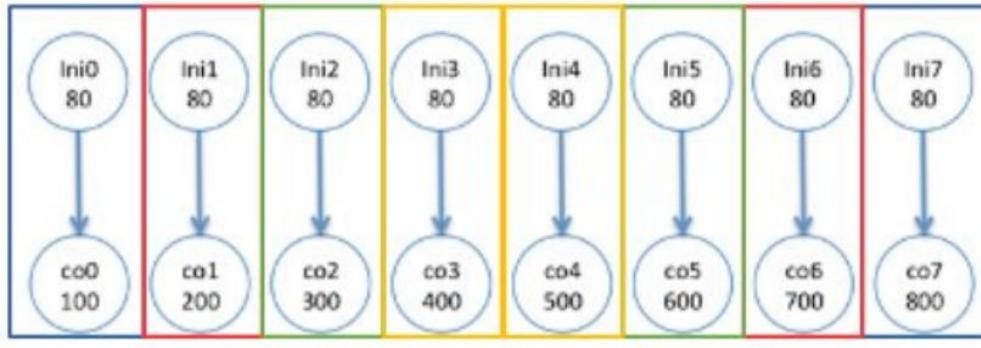
$$S_4 = T_1 / T_4 = 4240 / 1060 = 4$$



Solutions

c)

Best assignment of tasks to processors (scheduling) ...



Instructor Social Media

Youtube: Lucas Science



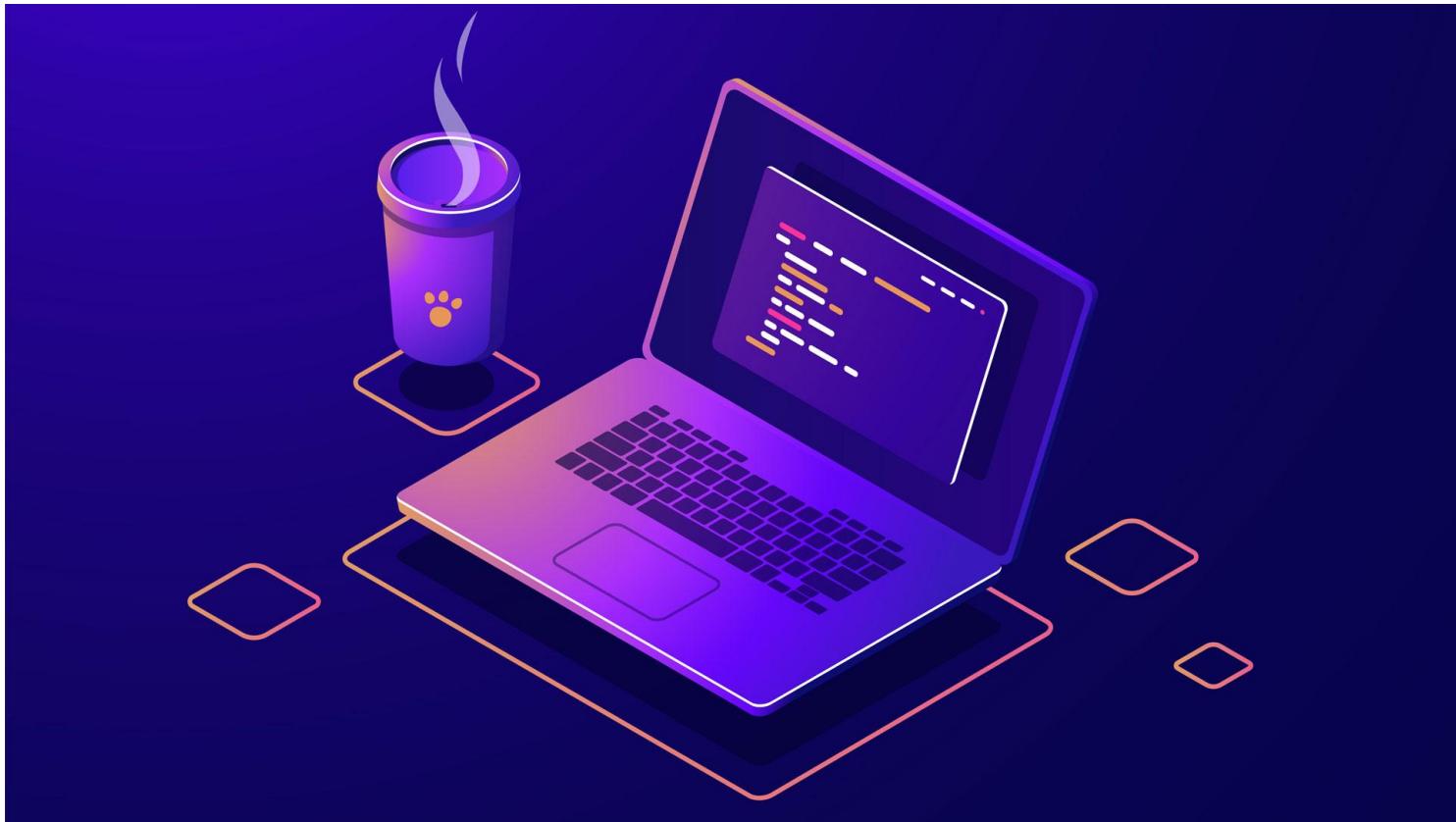
Instagram: lucaasbazilio



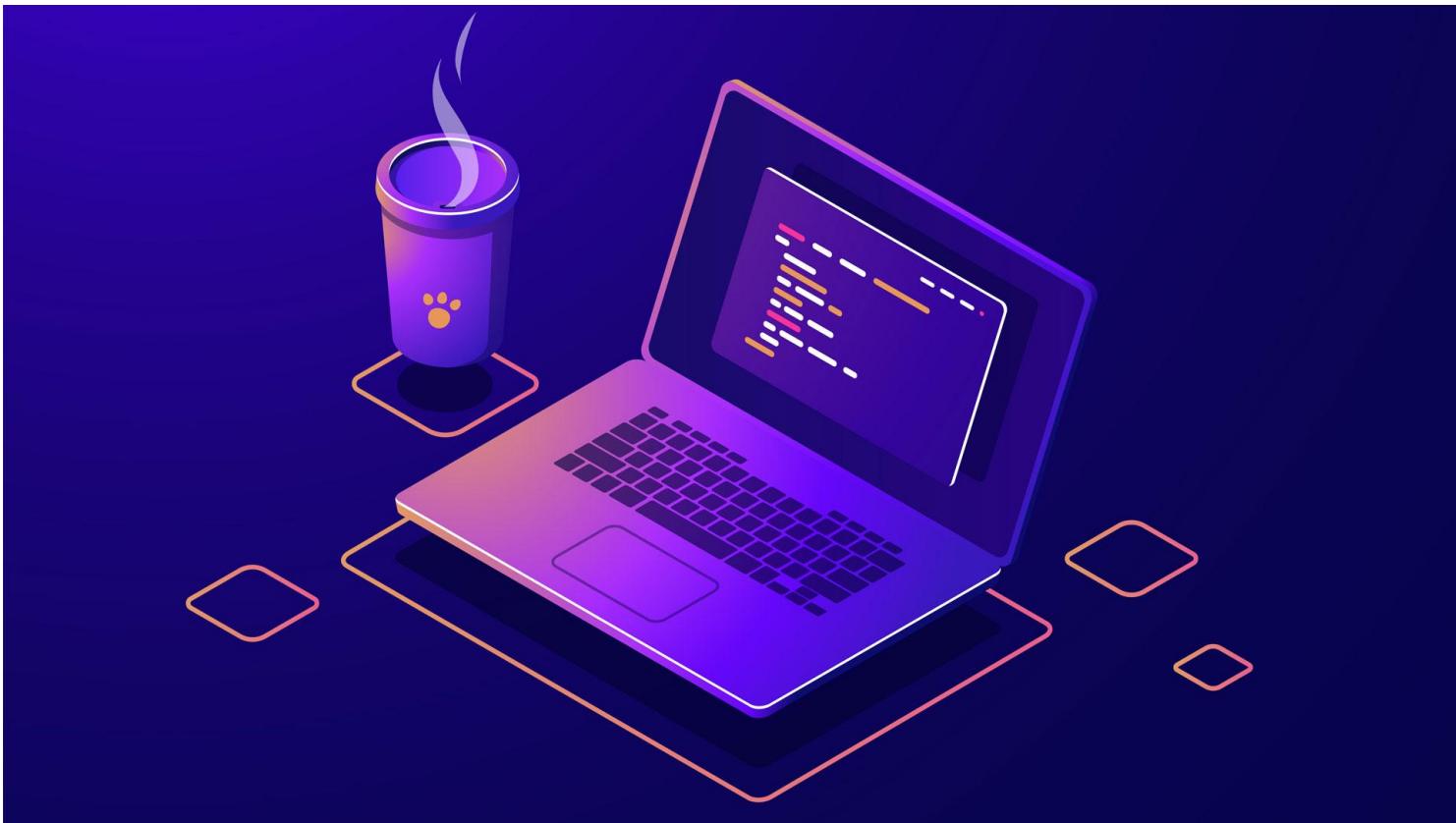
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 4



Problem 4

Given the following code:



Problem 4

```
# define N 4
int m[N][N]

// initialization
for (int i = 0; i < N; i++) {
    start_task("for_initialize");
    for (k = i; k < N; k++)
        if(k==i) modify_d(&m[i][i],i,i);
        else {
            modify_nd(&m[i][k],i,k);
            modify_nd(&m[k][i],k,i);
        }
    }
end_task("for_initialize");
}
```

Problem 4



```
// computation

for (int i = 0; i < N; i++) {
    start_task("for_compute");
    for (k = i+1; k < N; k++)
        int temp = m[i][k];
        m[i][k] = m[k][i];
        m[k][i] = temp;
    }
    end_task("for_compute");
}
```



Problem 4

```
// print results  
  
start_task("output");  
print_results(m);  
end_task("output");
```

Problem 4



Assuming that: 1) the execution of the `modify_d` routine takes 10 time units and the execution of the `modify_nd` routines takes 5 time units; 2) each internal iteration of the computation loop (i.e. each internal iteration of the *for-compute* task) takes 5 time units; and 3) the execution of the *output* task takes 100 time units, **we ask:**

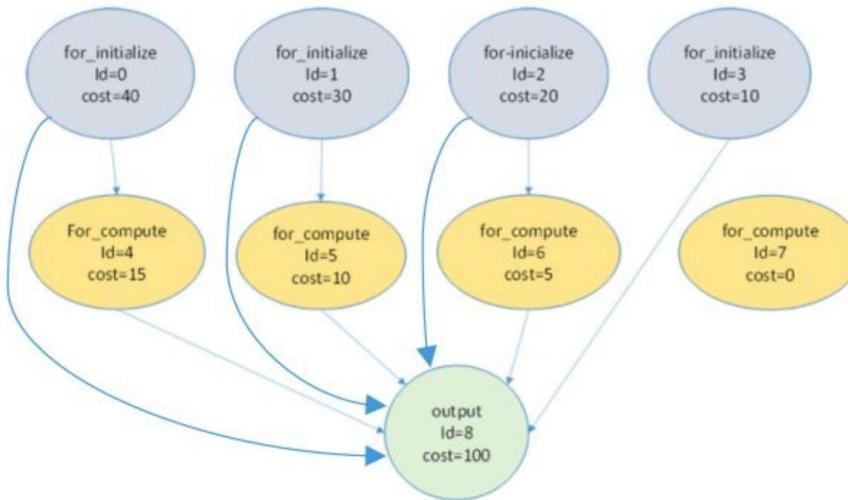
- (a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time (in time units).
- (b) Compute the values for T_1 , T_∞ and the potential *Parallelism*, as well as the parallel fraction (*phi*).
- (c) Indicate which would be the most appropriate task assignment on two processors in order to obtain the best possible "speed up". For that assignment, calculate T_2 and S_2 .

Solutions



a)

Task Dependence Graph:



Solutions



a)

m	0	1	N-1
0	10	5	5
1	5		
N-1	5		

task for_initialize (i=1, cost=30)

m	0	1	N-1
0			
1		5	5
N-1			

task for_compute (i=1, cost=10)

m	0	1	N-1
0			
1			
N-1			

task print_results (cost=100)

Instructor Social Media

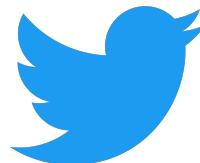
Youtube: Lucas Science



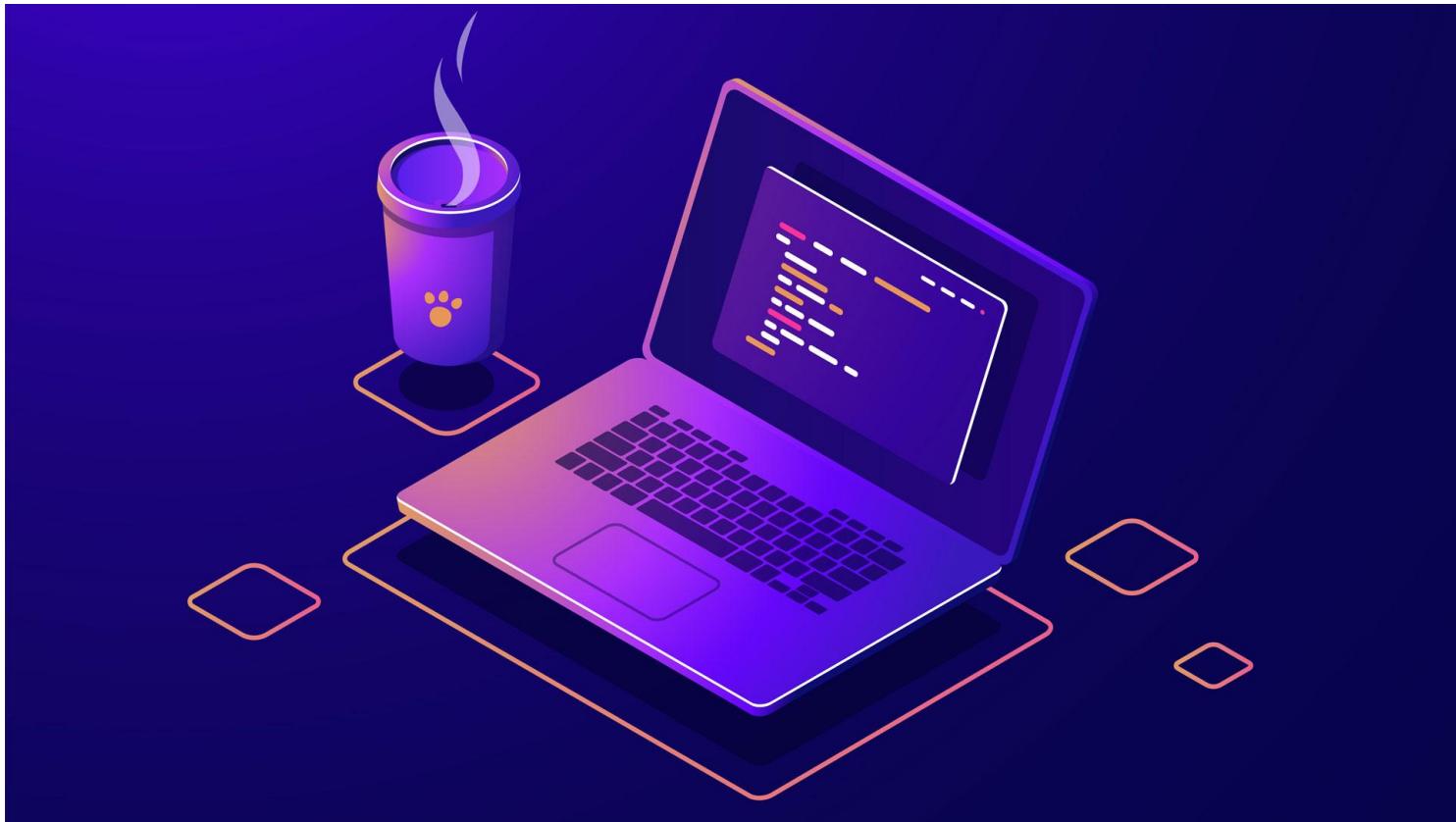
Instagram: lucaasbazilio



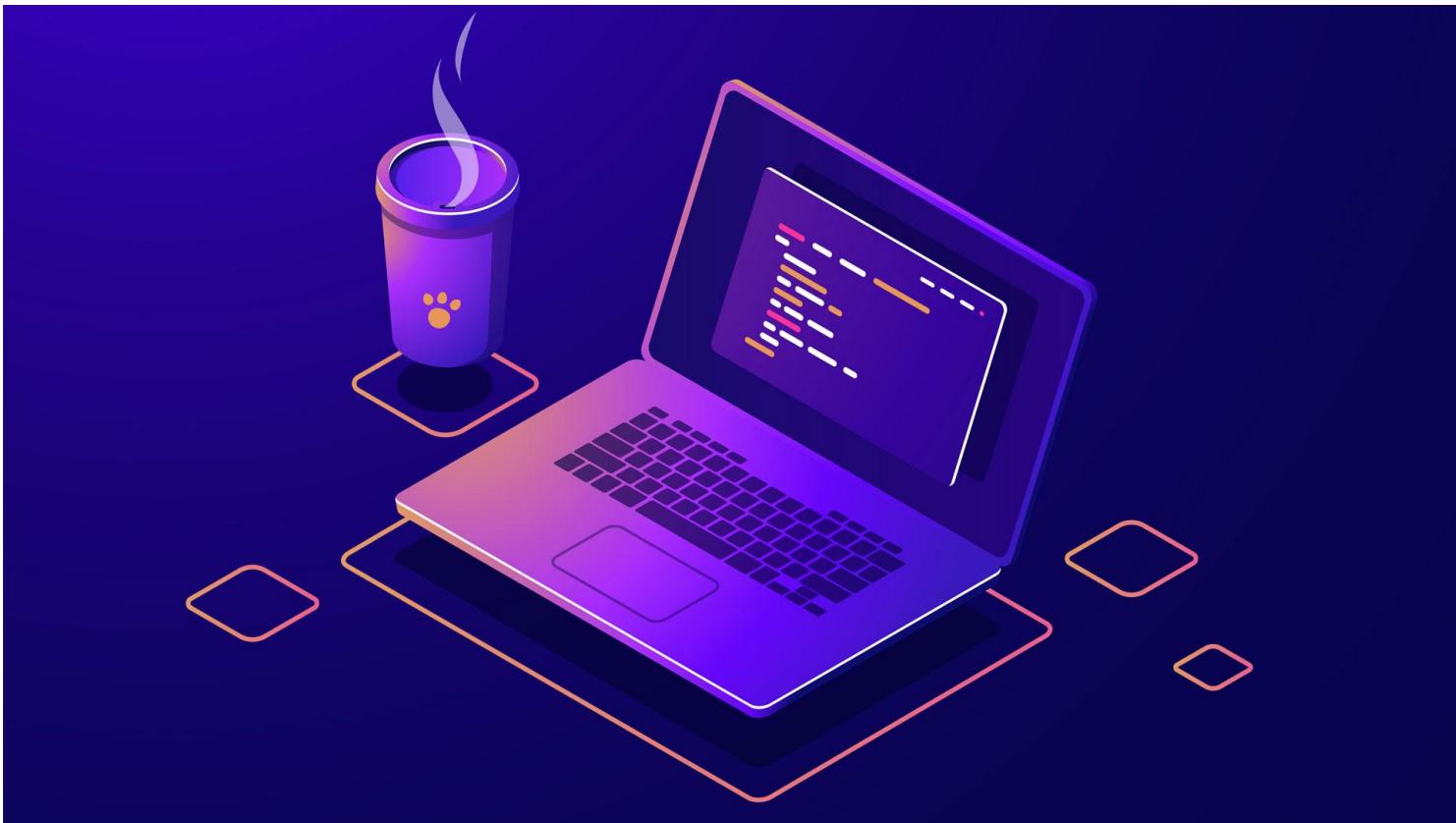
Twitter: lucasebazilio



Parallelism Fundamentals Problems



Problem 4



Problem 4

Given the following code:



Problem 4

```
# define N 4
int m[N][N]

// initialization
for (int i = 0; i < N; i++) {
    start_task("for_initialize");
    for (k = i; k < N; k++)
        if(k==i) modify_d(&m[i][i],i,i);
        else {
            modify_nd(&m[i][k],i,k);
            modify_nd(&m[k][i],k,i);
        }
    }
end_task("for_initialize");
}
```

Problem 4



```
// computation

for (int i = 0; i < N; i++) {
    start_task("for_compute");
    for (k = i+1; k < N; k++)
        int temp = m[i][k];
        m[i][k] = m[k][i];
        m[k][i] = temp;
    }
    end_task("for_compute");
}
```



Problem 4

```
// print results  
  
start_task("output");  
print_results(m);  
end_task("output");
```

Problem 4



Assuming that: 1) the execution of the `modify_d` routine takes 10 time units and the execution of the `modify_nd` routines takes 5 time units; 2) each internal iteration of the computation loop (i.e. each internal iteration of the *for-compute* task) takes 5 time units; and 3) the execution of the *output* task takes 100 time units, **we ask:**

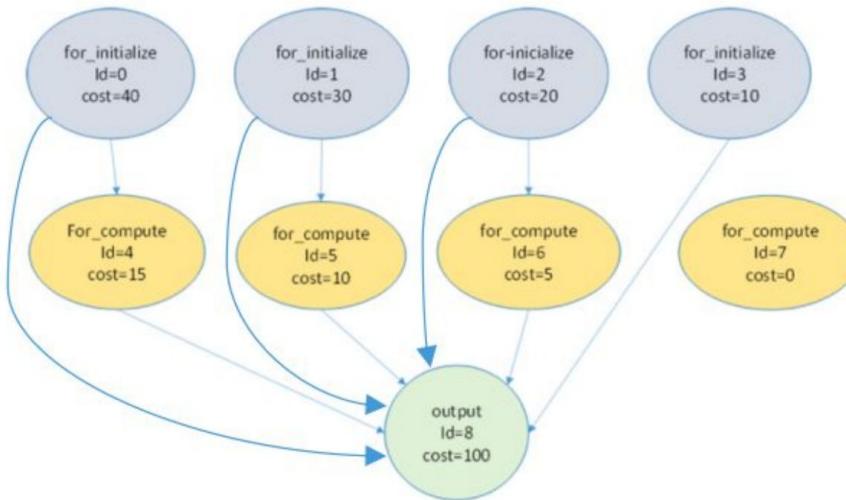
- (a) Draw the Task Dependence Graph (TDG), indicating for each node its cost in terms of execution time (in time units).
- (b) Compute the values for T_1 , T_∞ and the potential *Parallelism*, as well as the parallel fraction (*phi*).
- (c) Indicate which would be the most appropriate task assignment on two processors in order to obtain the best possible "speed up". For that assignment, calculate T_2 and S_2 .

Solutions



a)

Task Dependence Graph:



Solutions



a)

m	0	1	N-1
0	10	5	5
1	5		
N-1	5		

task for_initialize (i=1, cost=30)

m	0	1	N-1
0			
1		5	5
N-1			

task for_compute (i=1, cost=10)

m	0	1	N-1
0			
1			
N-1			

task print_results (cost=100)

Solutions



b)

$$T_1 = 230$$

$$T_{\infty} = 155$$

$$\text{Parallelism} = T_1 / T_{\infty} = 230 / 155 = 1.48$$

$$P_{\min} = 2$$

$$\varphi = T_{\infty} / T_1 = 155 / 230 = 0.673$$

Instructor Social Media

Youtube: Lucas Science



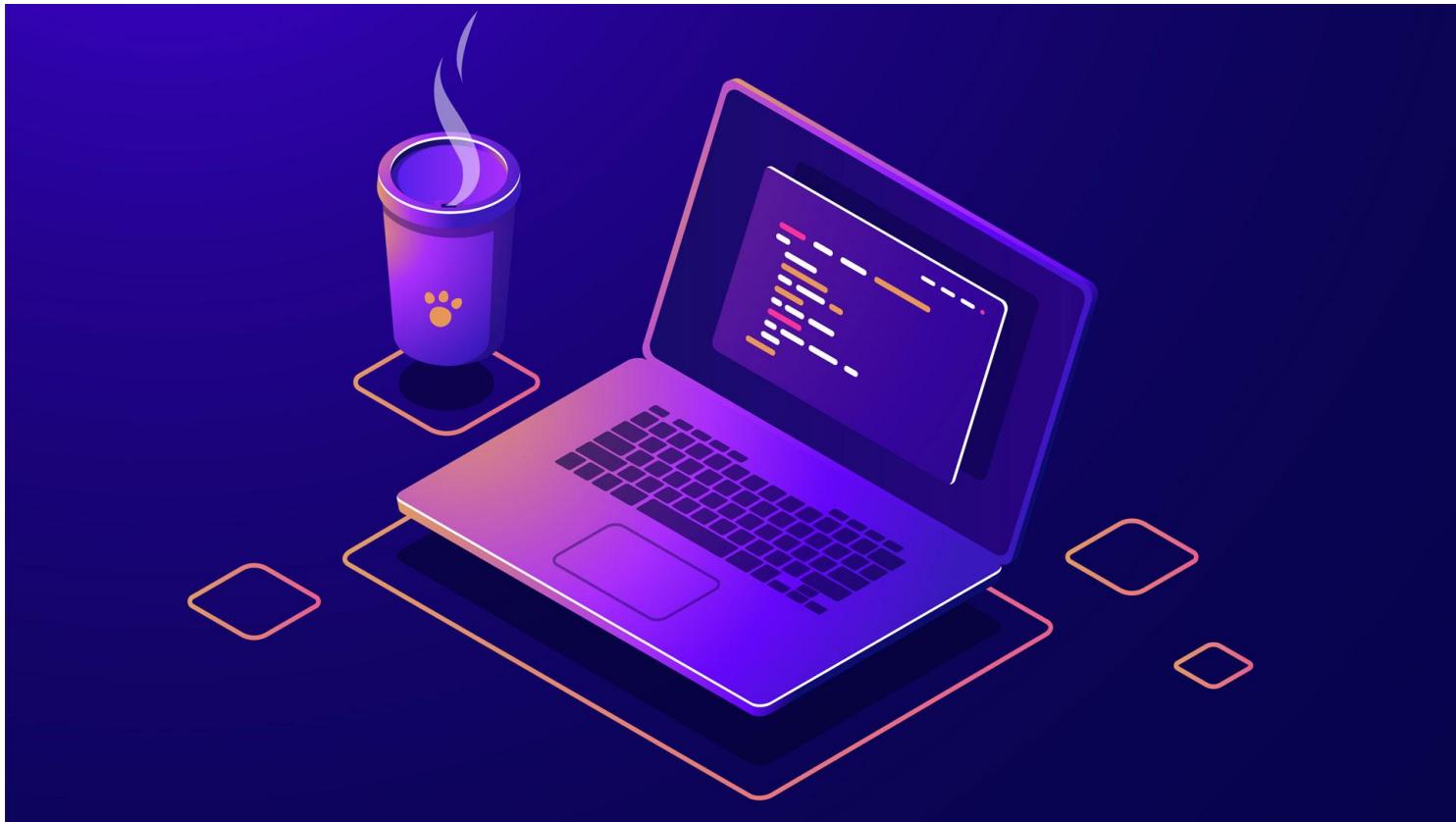
Instagram: lucaasbazilio



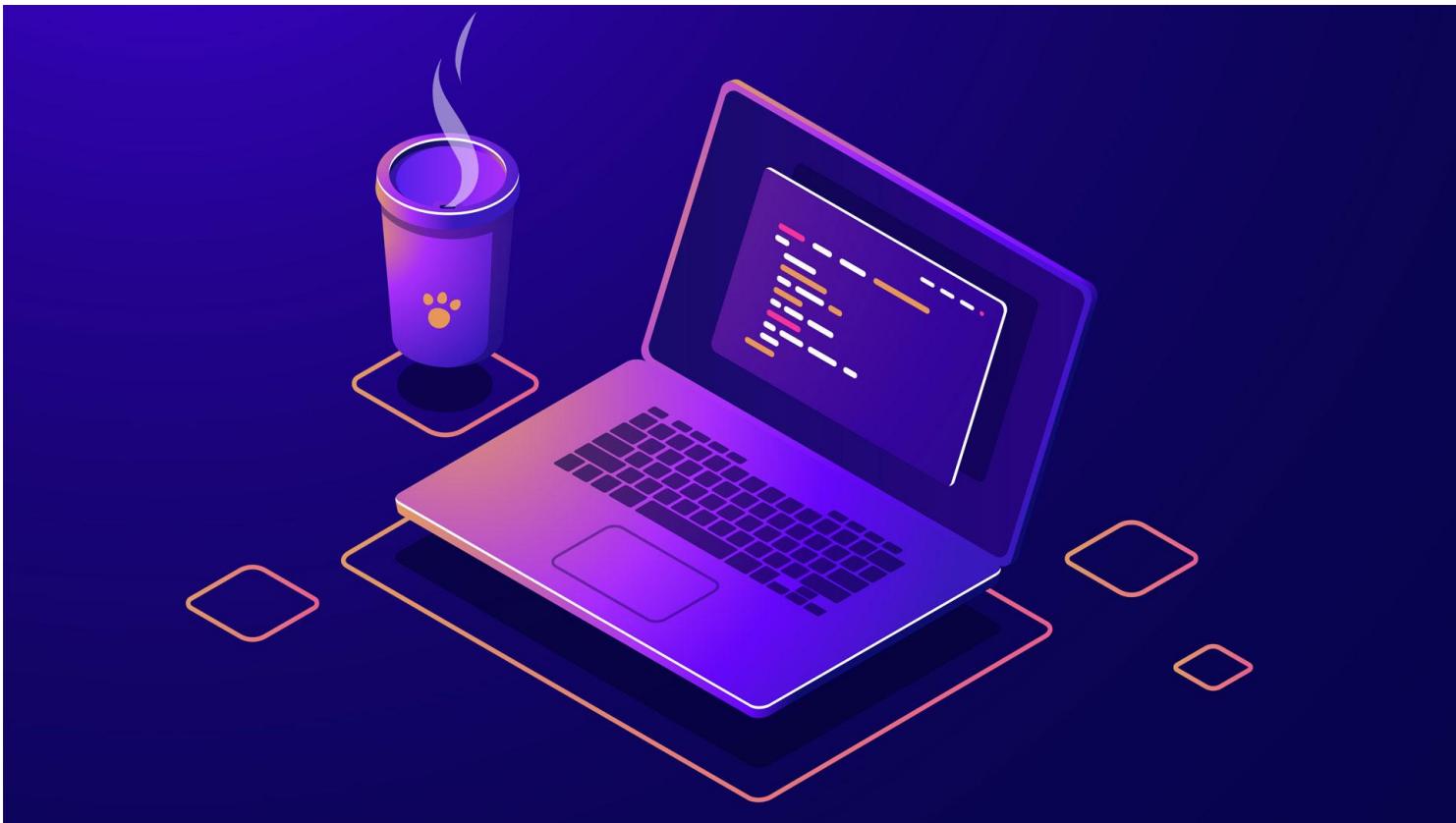
Twitter: lucasebazilio



Parallelism Fundamentals Problems



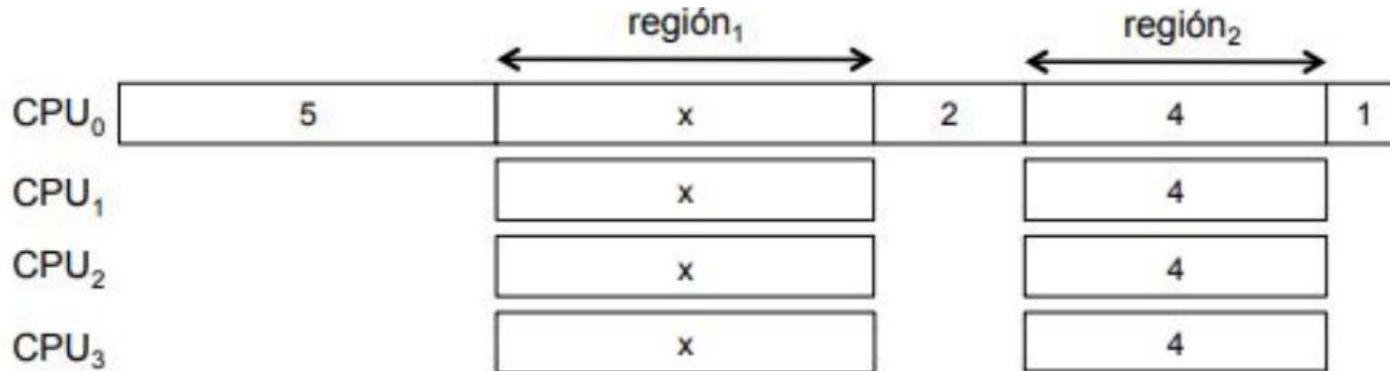
Problem 5



Problem 5



The following figure shows an incomplete time diagram for the execution of a parallel application on 4 processors:



Problem 5



The figure has a set of rectangles, each rectangle represents the execution of a task with its associated cost in time units. In the timeline there are two regions (1 and 2) with 4 parallel tasks each. The execution cost for tasks in *region*₁ is unknown (x time units each); the cost for each task in *region*₂ is 4 time units. The computation starts with a sequential task (with cost 5), then all tasks in *region*₁ running in parallel, followed by another sequential task (with cost 2), then all tasks in *region*₂ running in parallel followed by a final sequential task (with cost 1).

Knowing that an ideal speed-up of 9 could be achieved if the application could make use of infinite processors ($S_{p \rightarrow \infty} = 9$), and assuming that the two parallel regions can be decomposed ideally, with as many tasks as processors with the appropriate fraction of the original cost, **we ask:**

- (a) What is the parallel fraction (ϕ) for the application represented in the time diagram above?
- (b) Which is the "speedup" that is achieved in the execution with 4 processors (S_4)?
- (c) Which is the value x in *region*₁?

Solutions



a)

$$S_{\infty} = 1 / 1 - \varphi$$

$$9 = 1 / 1 - \varphi \Rightarrow \varphi = 0.8888$$

Instructor Social Media

Youtube: Lucas Science



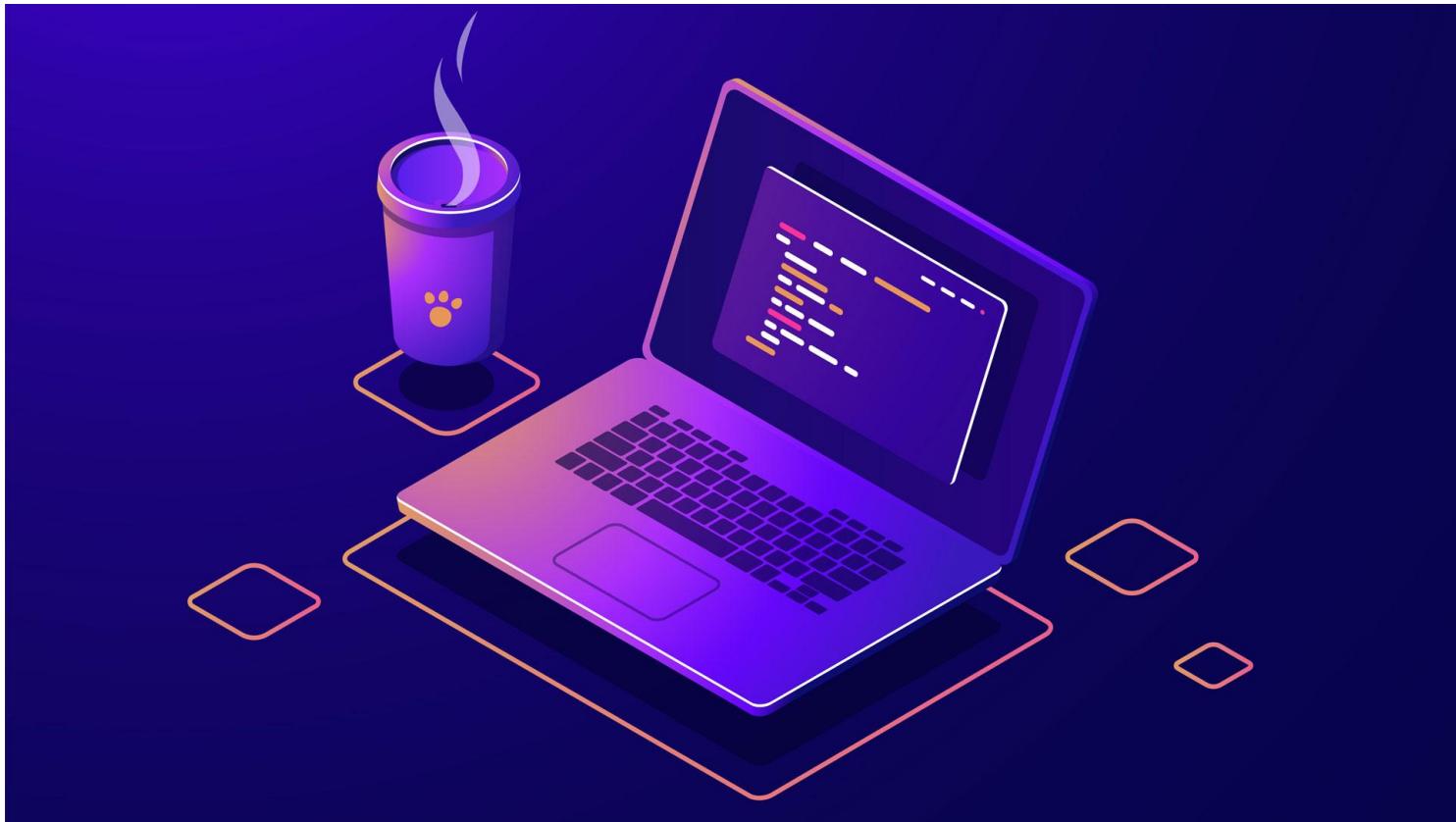
Instagram: lucaasbazilio



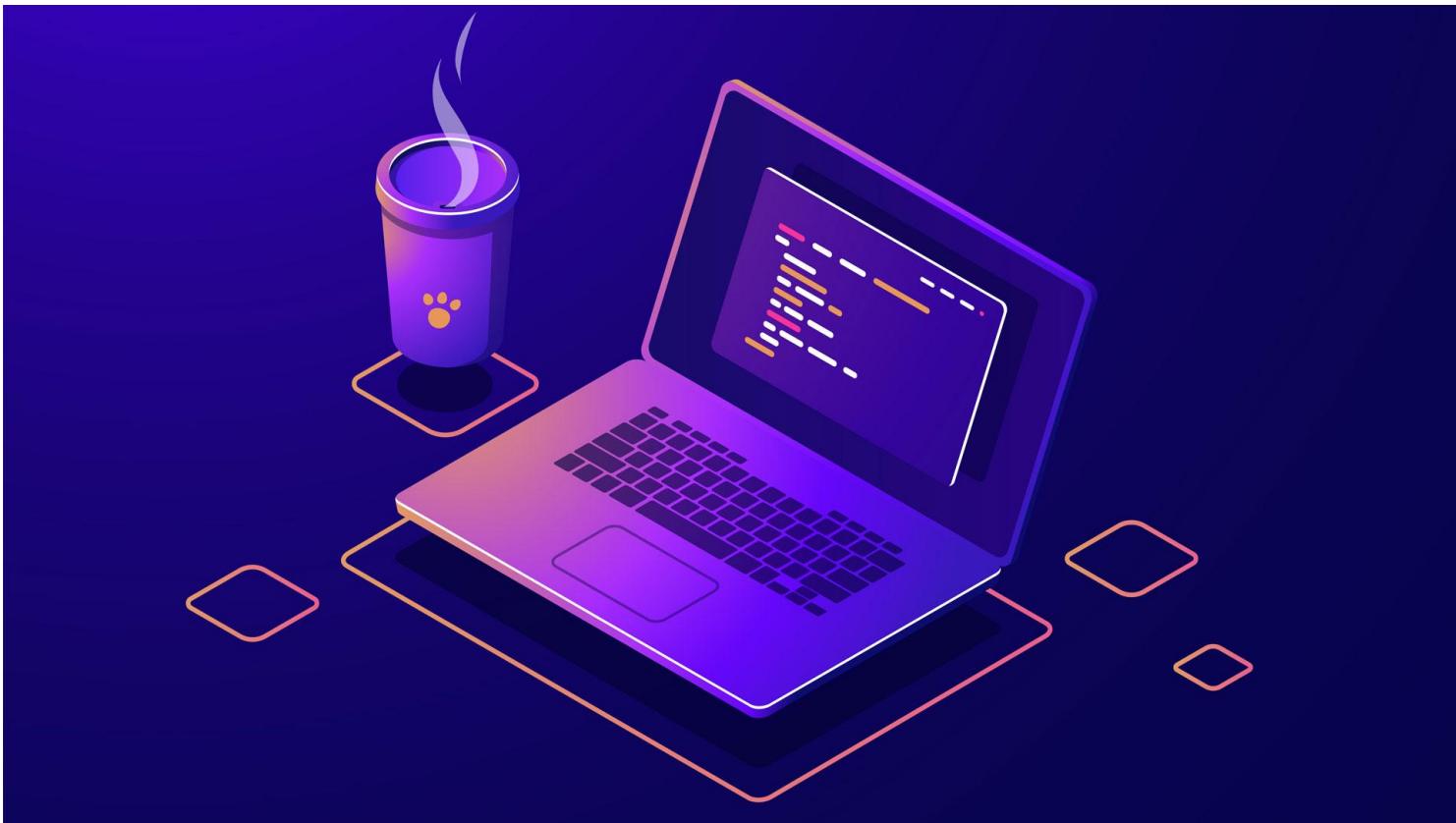
Twitter: lucasebazilio



Parallelism Fundamentals Problems



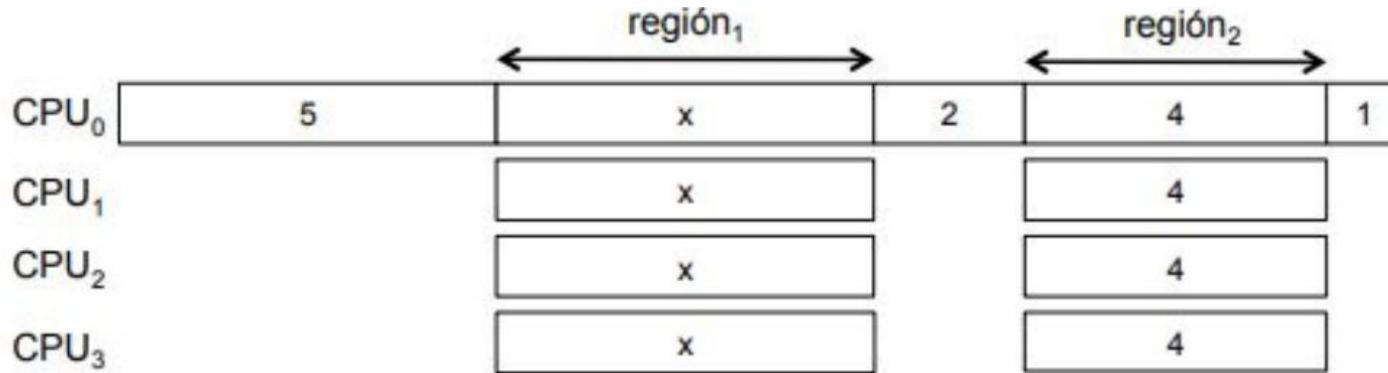
Problem 5



Problem 5



The following figure shows an incomplete time diagram for the execution of a parallel application on 4 processors:



Problem 5



The figure has a set of rectangles, each rectangle represents the execution of a task with its associated cost in time units. In the timeline there are two regions (1 and 2) with 4 parallel tasks each. The execution cost for tasks in $region_1$ is unknown (x time units each); the cost for each task in $region_2$ is 4 time units. The computation starts with a sequential task (with cost 5), then all tasks in $region_1$ running in parallel, followed by another sequential task (with cost 2), then all tasks in $region_2$ running in parallel followed by a final sequential task (with cost 1).

Knowing that an ideal speed-up of 9 could be achieved if the application could make use of infinite processors ($S_{p \rightarrow \infty} = 9$), and assuming that the two parallel regions can be decomposed ideally, with as many tasks as processors with the appropriate fraction of the original cost, **we ask:**

- (a) What is the parallel fraction (ϕ) for the application represented in the time diagram above?
- (b) Which is the "speedup" that is achieved in the execution with 4 processors (S_4)?
- (c) Which is the value x in $region_1$?

Solutions



a)

$$S_{\infty} = 1 / 1 - \varphi$$

$$9 = 1 / 1 - \varphi \Rightarrow \varphi = 0.8888$$

Solutions



b)

$$S_4 = 1 / ((1-\varphi) + \varphi/P4)$$

$$S_4 = 3$$

Instructor Social Media

Youtube: Lucas Science



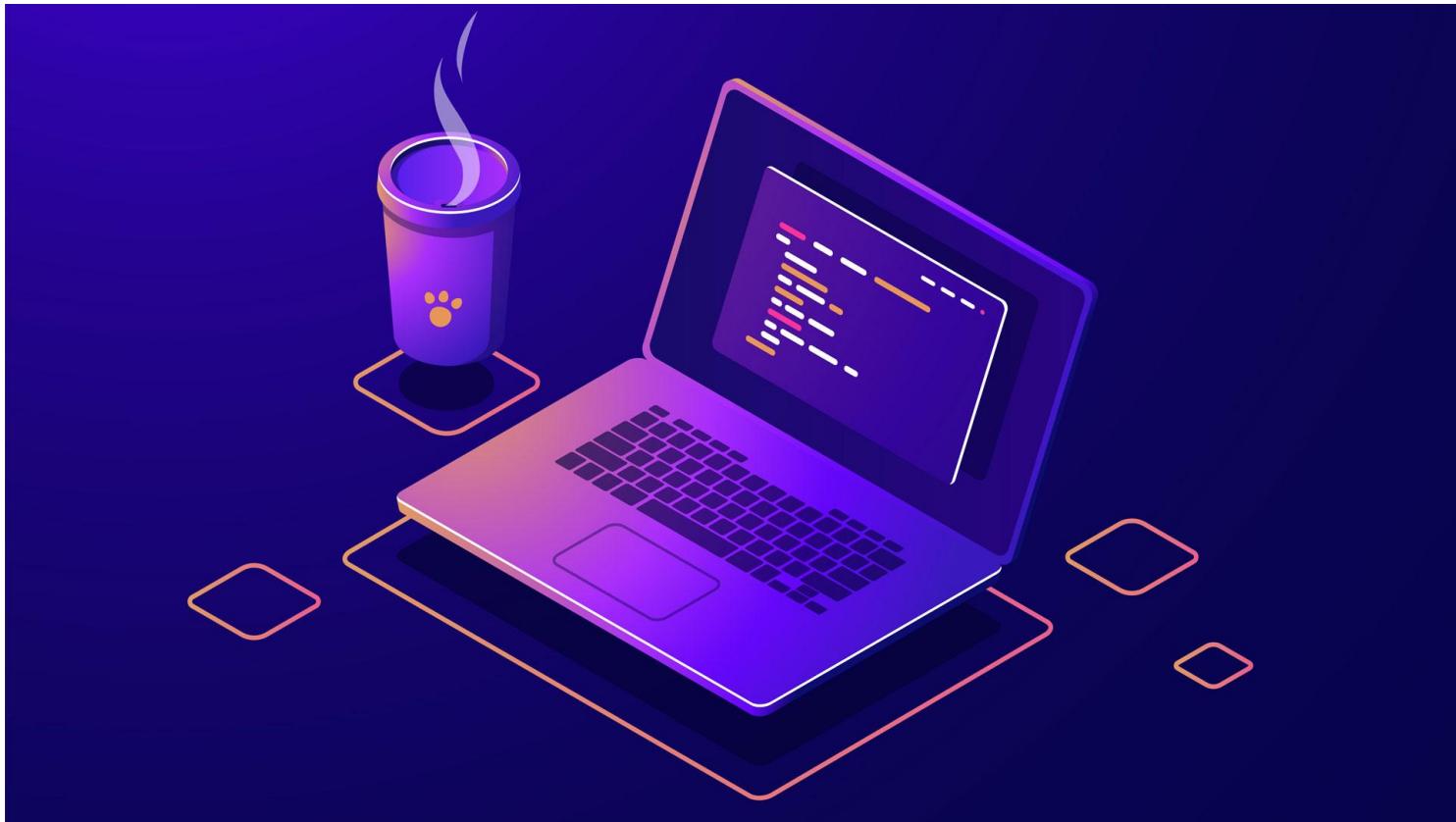
Instagram: lucaasbazilio



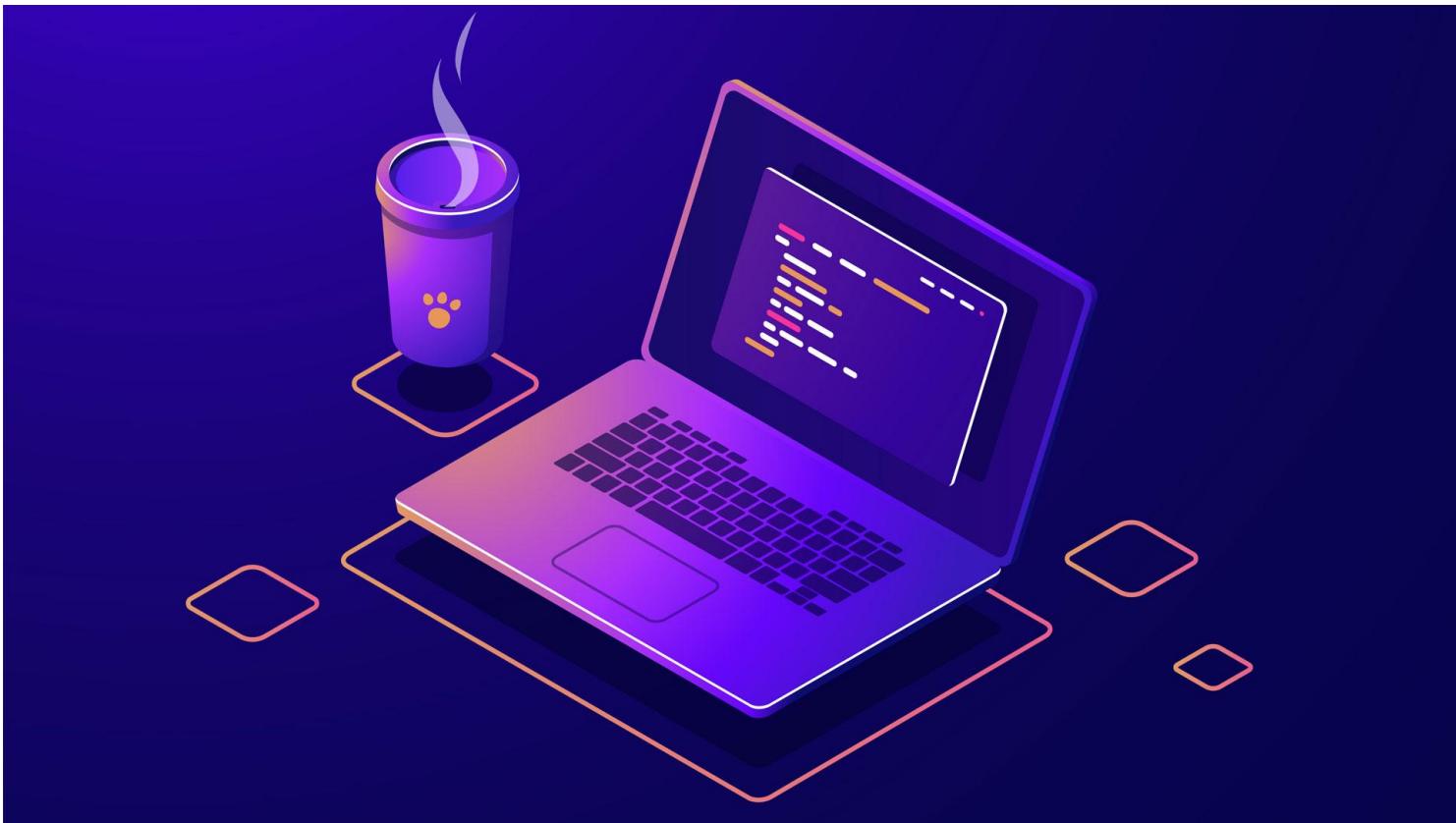
Twitter: lucasebazilio



Parallelism Fundamentals Problems



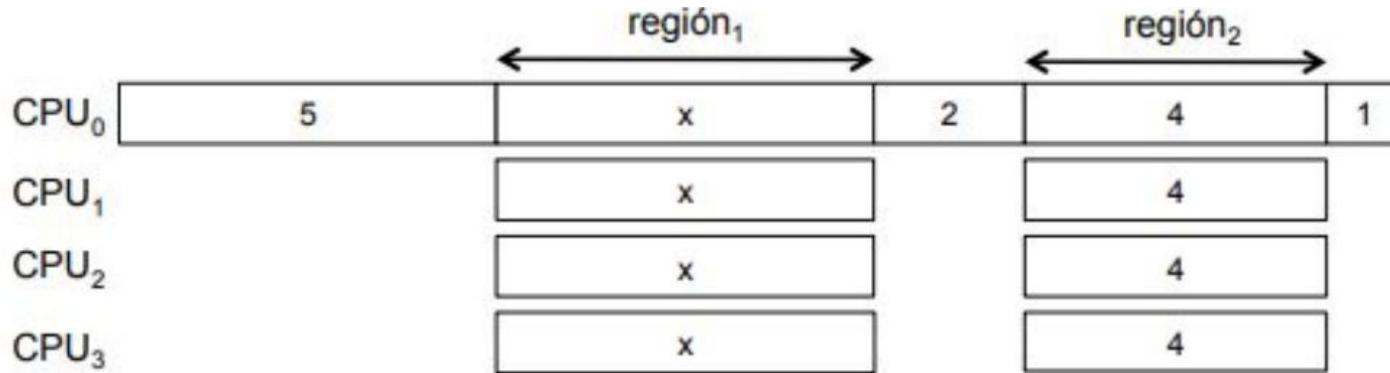
Problem 5



Problem 5



The following figure shows an incomplete time diagram for the execution of a parallel application on 4 processors:



Problem 5



The figure has a set of rectangles, each rectangle represents the execution of a task with its associated cost in time units. In the timeline there are two regions (1 and 2) with 4 parallel tasks each. The execution cost for tasks in $region_1$ is unknown (x time units each); the cost for each task in $region_2$ is 4 time units. The computation starts with a sequential task (with cost 5), then all tasks in $region_1$ running in parallel, followed by another sequential task (with cost 2), then all tasks in $region_2$ running in parallel followed by a final sequential task (with cost 1).

Knowing that an ideal speed-up of 9 could be achieved if the application could make use of infinite processors ($S_{p \rightarrow \infty} = 9$), and assuming that the two parallel regions can be decomposed ideally, with as many tasks as processors with the appropriate fraction of the original cost, **we ask:**

- (a) What is the parallel fraction (ϕ) for the application represented in the time diagram above?
- (b) Which is the "speedup" that is achieved in the execution with 4 processors (S_4)?
- (c) Which is the value x in $region_1$?

Solutions



a)

$$S_{\infty} = 1 / 1 - \varphi$$

$$9 = 1 / 1 - \varphi \Rightarrow \varphi = 0.8888$$

Solutions



b)

$$S_4 = 1 / ((1-\varphi) + \varphi/P4)$$

$$S_4 = 3$$

Solutions



c)

$$\phi = T_{\text{PAR}} / T_1 = 4x + 16 / 4x + 24$$

$$\Rightarrow x = 12$$

Instructor Social Media

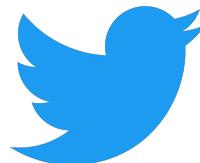
Youtube: Lucas Science



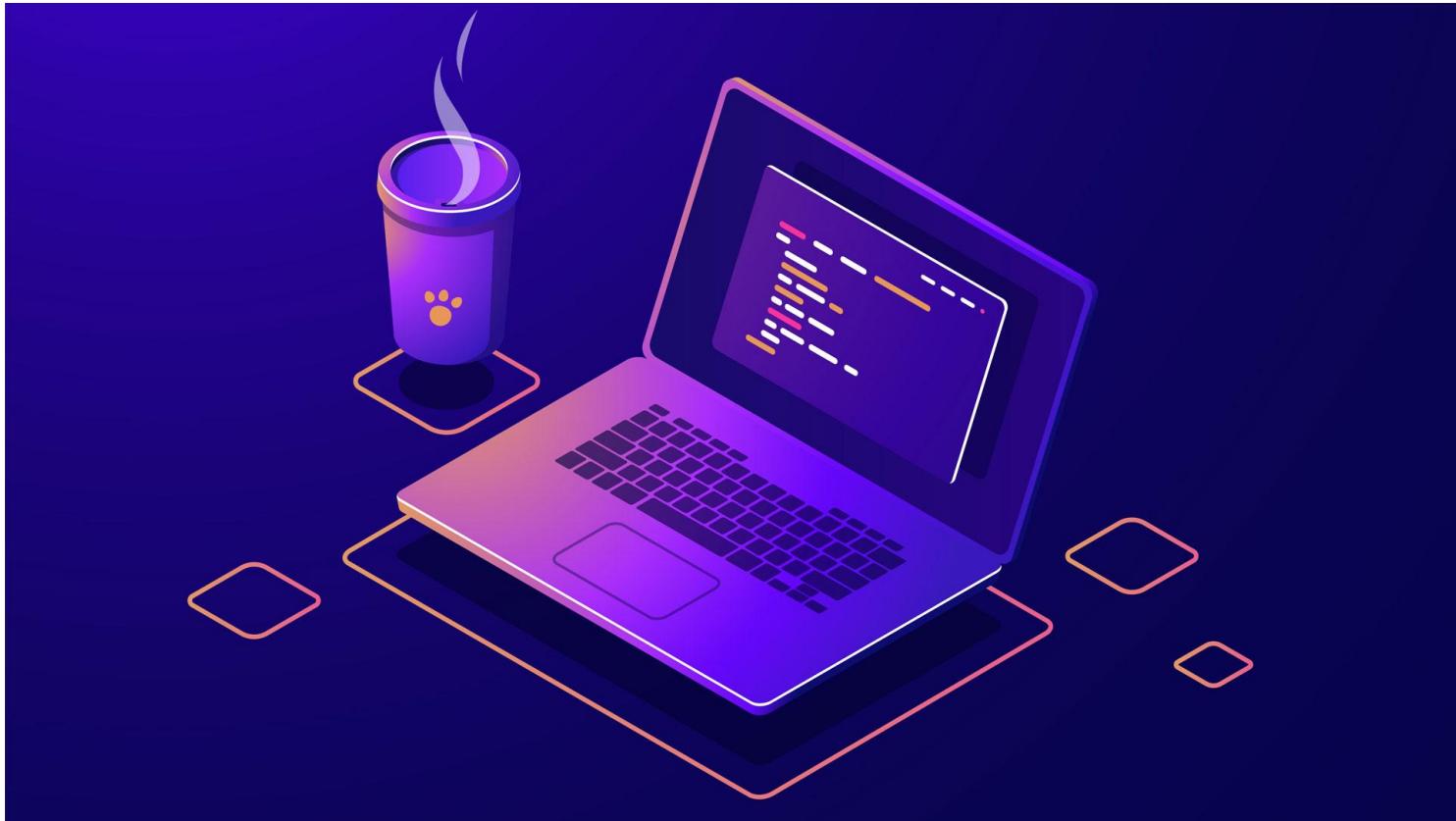
Instagram: lucaasbazilio



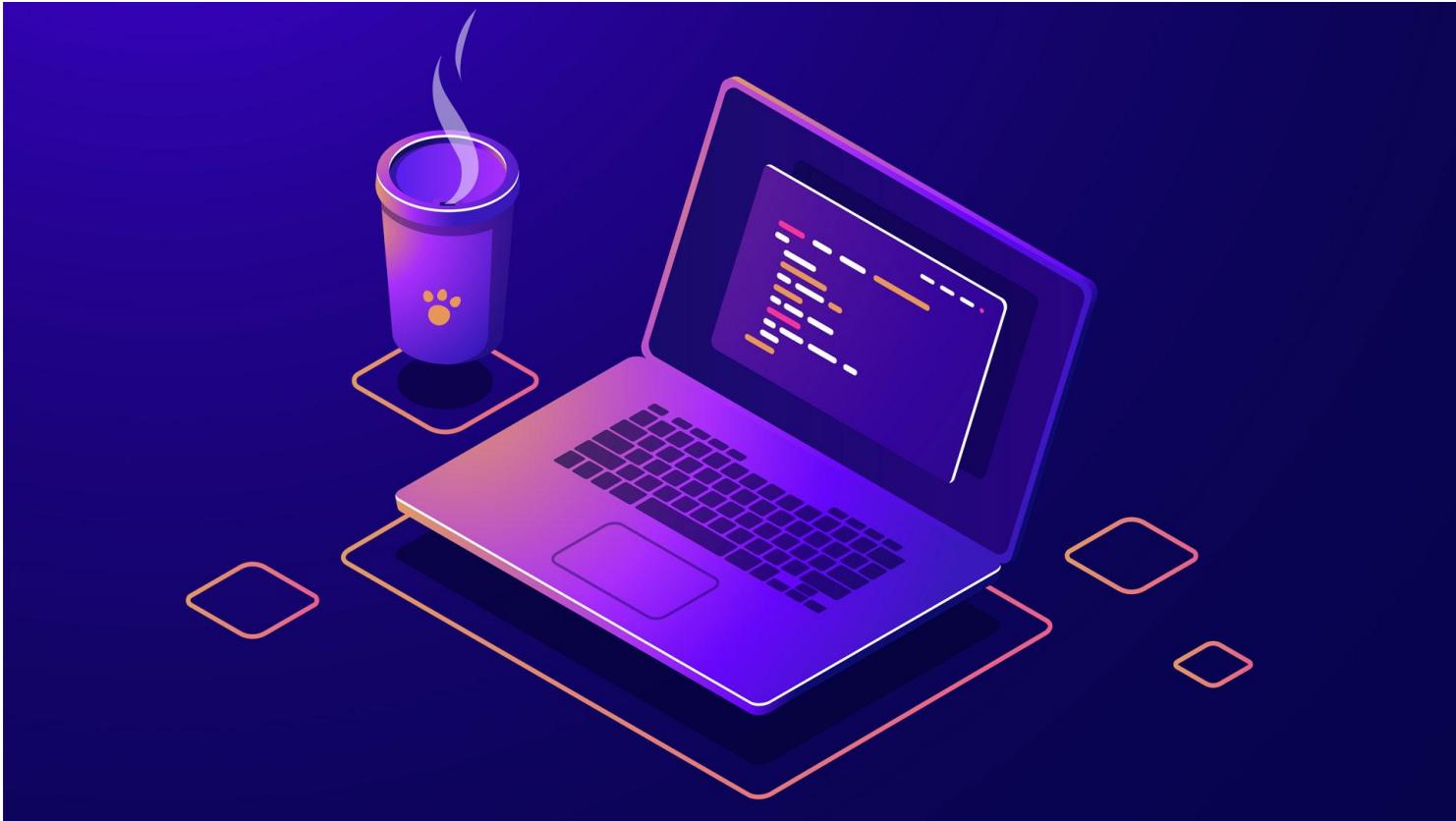
Twitter: lucasebazilio



Task Decomposition - OpenMP



Task Decomposition Strategies



Task Decomposition Strategies



- **Linear** task decomposition
 - A task is a "code block" or a procedure invocation
- **Iterative** task decomposition
 - Tasks found in iterative constructs, such as loops (countable or uncountable)
- **Recursive** task decomposition
 - Tasks found in divide-and-conquer problems and other recursive problems

Linear Task Decomposition



A task is a “ code block “ or a procedure invocation

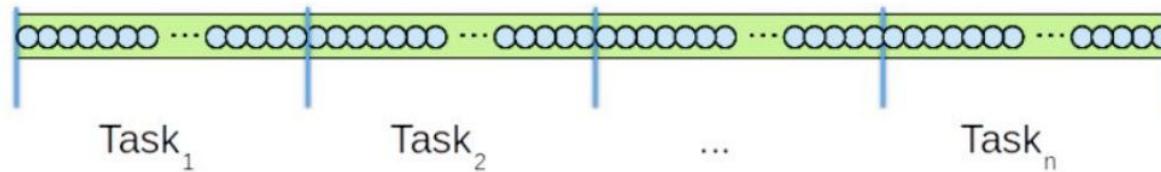
```
int main() {  
    start_task("init_A");  
    initialize(A,N);  
    end_task("init_A");  
  
    start_task("init_B");  
    initialize(B,N);  
    end_task("init_B");  
  
}
```

Iterative Task Decomposition



A task is a chunk of iterations of a loop, as for example, in the sum of two vectors

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];  
}  
  
void main() {  
    ....  
    vector_add(a, b, c, N);  
    ...  
}
```



Iterative Task Decomposition



Single loop iteration:

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; ii++)  
        .start_task("singleit");  
    C[i] = A[i] + B[i];  
    .end_task("singleit");  
}
```

Chunk of loop iterations:

```
#define BS 16  
void vector_add(int *A, int *B, int *C, int n) {  
    for (int ii=0; ii< n; ii+=BS) {  
        .start_task("chunkit");  
        for (i = ii; i < min(ii+BS, n), i++)  
            C[i] = A[i] + B[i];  
        .end_task("chunkit");  
    }  
}
```

Iterative Task Decomposition



List of elements, traversed using an uncountable (while) loop

```
int main() {
    struct node *p;

    p = init_list(n);
    ...

    while (p != NULL) {
        .start_task("computeNode");
        process_work(p);
        .end_task("computeNode");
        p = p->next;
    }
    ...
}
```

Recursive Task Decomposition



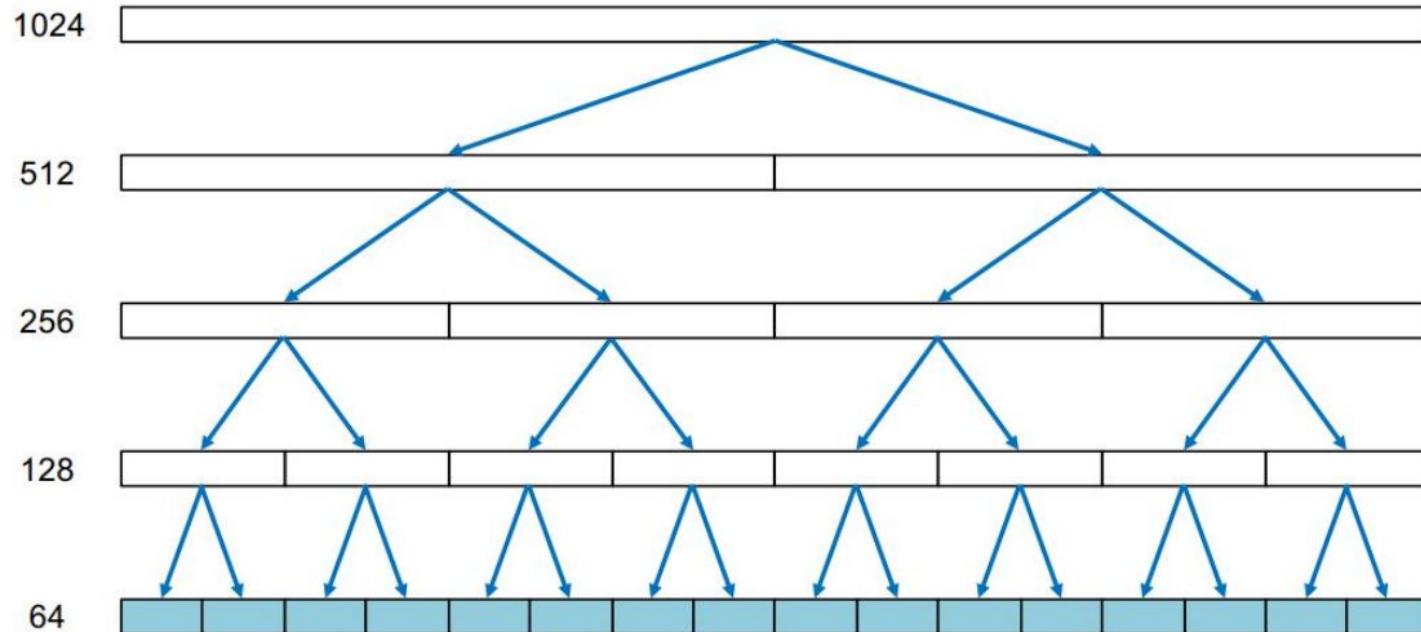
Sum of two vectors by recursively dividing the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_vector_add(A, B, C, n2);
        rec_vector_add(A+n2, B+n2, C+n2, n-n2);
    }
    else vector_add(A, B, C, n);
}
void main() {
    ...
    rec_vector_add(a, b, c, N);
    ...
}
```

Recursive Task Decomposition



Instructor Social Media

Youtube: Lucas Science



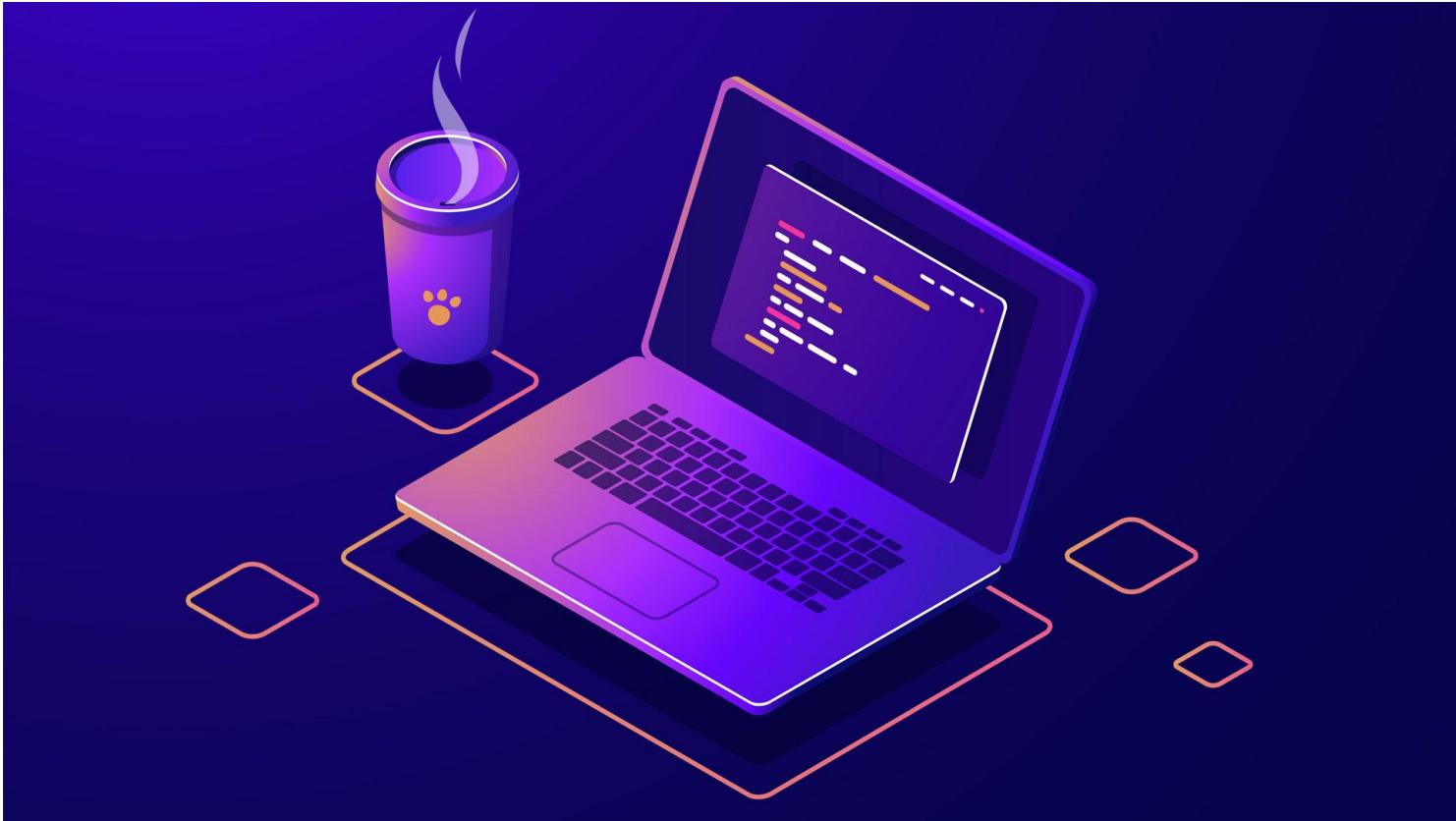
Instagram: lucaasbazilio



Twitter: lucasebazilio



Recursive Task Decomposition



Task Decomposition Strategies



- **Linear** task decomposition
 - A task is a "code block" or a procedure invocation
- **Iterative** task decomposition
 - Tasks found in iterative constructs, such as loops (countable or uncountable)
- **Recursive** task decomposition
 - Tasks found in divide-and-conquer problems and other recursive problems

Linear Task Decomposition



A task is a “ code block “ or a procedure invocation

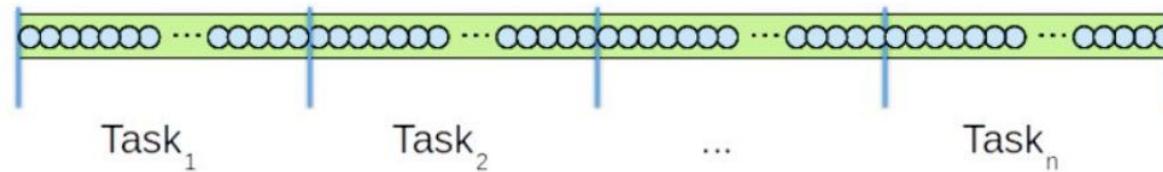
```
int main() {  
    start_task("init_A");  
    initialize(A,N);  
    end_task("init_A");  
  
    start_task("init_B");  
    initialize(B,N);  
    end_task("init_B");  
  
}
```

Iterative Task Decomposition



A task is a chunk of iterations of a loop, as for example, in the sum of two vectors

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];  
}  
  
void main() {  
    ....  
    vector_add(a, b, c, N);  
    ...  
}
```



Iterative Task Decomposition



Single loop iteration:

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; ii++)  
        .start_task("singleit");  
    C[i] = A[i] + B[i];  
    .end_task("singleit");  
}
```

Chunk of loop iterations:

```
#define BS 16  
void vector_add(int *A, int *B, int *C, int n) {  
    for (int ii=0; ii< n; ii+=BS) {  
        .start_task("chunkit");  
        for (i = ii; i < min(ii+BS, n), i++)  
            C[i] = A[i] + B[i];  
        .end_task("chunkit");  
    }  
}
```

Iterative Task Decomposition



List of elements, traversed using an uncountable (while) loop

```
int main() {
    struct node *p;

    p = init_list(n);
    ...

    while (p != NULL) {
        .start_task("computeNode");
        process_work(p);
        .end_task("computeNode");
        p = p->next;
    }
    ...
}
```

Recursive Task Decomposition



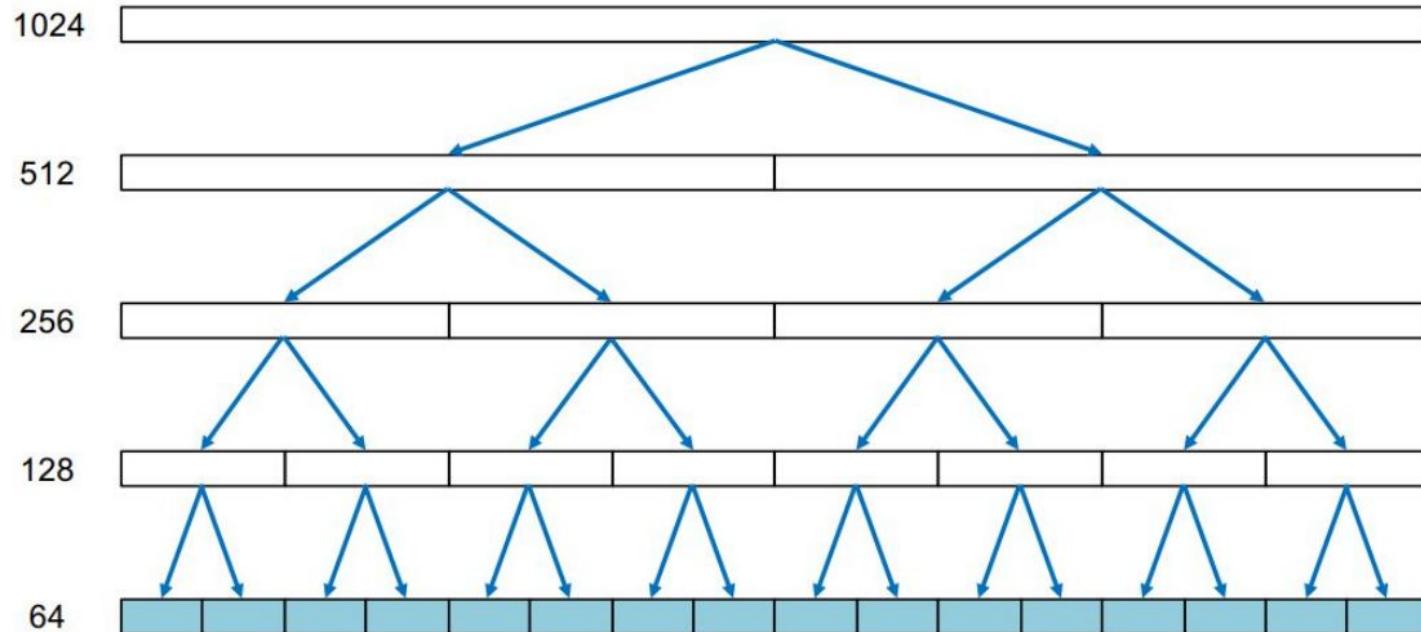
Sum of two vectors by recursively dividing the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_vector_add(A, B, C, n2);
        rec_vector_add(A+n2, B+n2, C+n2, n-n2);
    }
    else vector_add(A, B, C, n);
}
void main() {
    ...
    rec_vector_add(a, b, c, N);
    ...
}
```

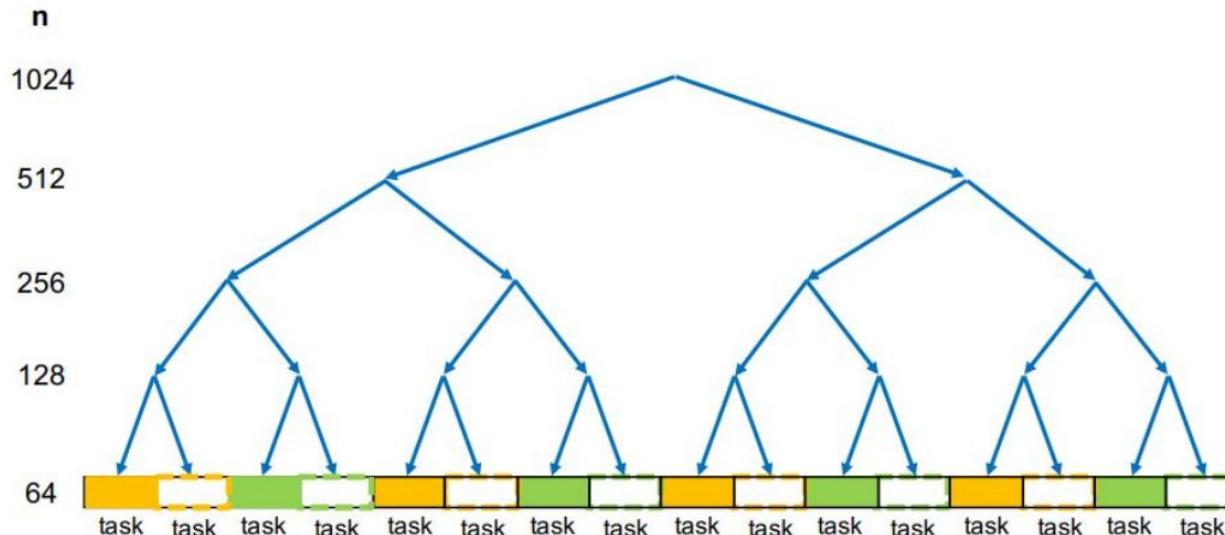
Recursive Task Decomposition



Two possible decomposition strategies



- **Leaf strategy:** a task corresponds with each invocation of `vector_add` once the recursive invocations stop



Leaf Strategy



```
#define N 1024
#define MIN_SIZE 64

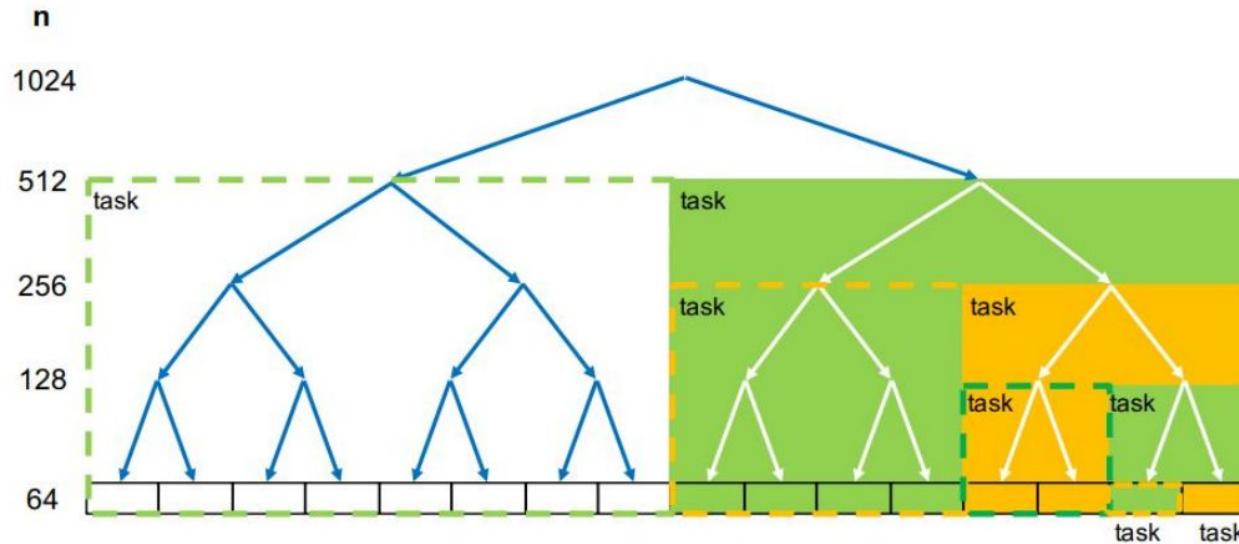
void vector_add(int *A, int *B, int *C, int n) {
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_vector_add(A, B, C, n2);
        rec_vector_add(A+n2, B+n2, C+n2, n-n2);
    }
    else
    {
        start_task("leaftask");
        vector add(A, B, C, n);
        end_task("leaftask");
    }
}
void main() {
    ...
    rec_vector_add(a, b, c, N);
    ...
}
```

Two possible decomposition strategies



- **Tree strategy:** a task corresponds with each invocation of `rec_vector_add` during the *parallel* recursive execution



Tree Strategy



```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
    for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        start_task("treetask1");
        rec_vector_add(A, B, C, n2);
        _end_task("treetask1");
        _start_task("treetask2");
        rec_vector_add(A+n2, B+n2, C+n2, n-n2);
        end_task("treetask2");
    }
    else vector_add(A, B, C, n);
}
void main() {
    ...
    rec_vector_add(a, b, c, N);
    ...
}
```

Instructor Social Media

Youtube: Lucas Science



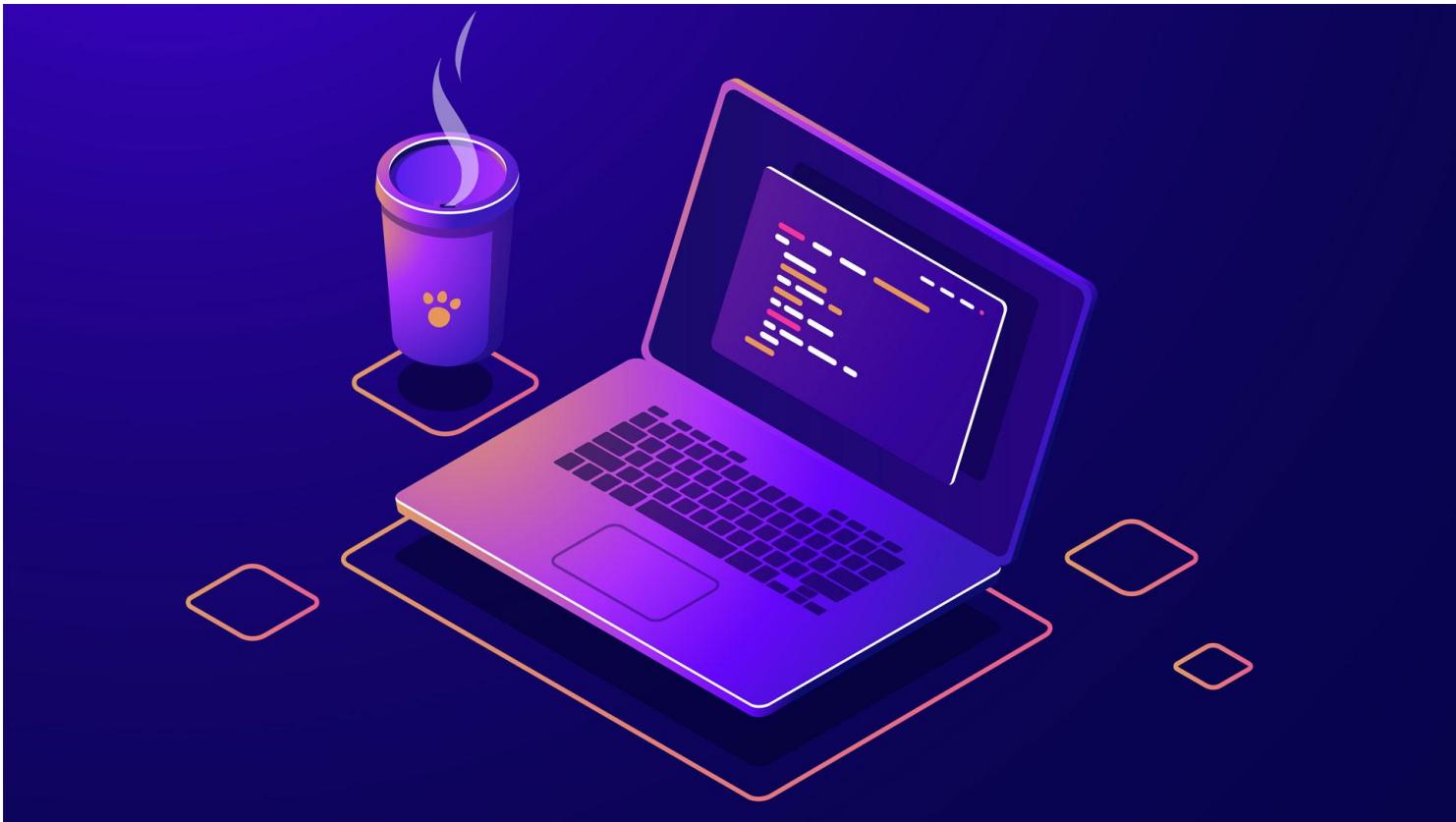
Instagram: lucaasbazilio



Twitter: lucasebazilio



Introduction to OpenMP





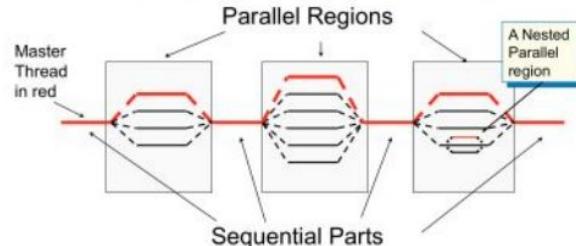
OpenMP is an application programming interface for shared memory multithreaded programming on multiple platforms. It allows adding parallelism to programs written in C, C++ and Fortran.

It consists of a set of compiler directives, library routines, and environment variables that influence runtime behavior.

Task Creation in OpenMP



- ▶ `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)

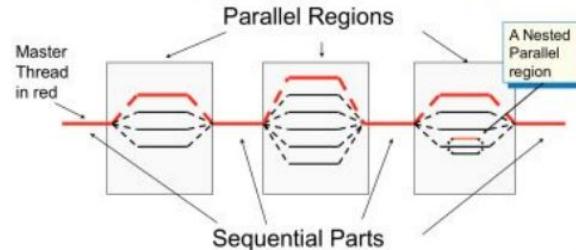


- ▶ `int omp_get_num_threads`: returns the number of threads in the current team. 1 if outside a parallel region
- ▶ `int omp_get_thread_num`: returns the identifier of the thread in the current team that is executing a task, a value between 0 and `omp_get_num_threads() - 1`

Task Creation in OpenMP



- ▶ `#pragma omp parallel`: One **implicit** task is created for each thread in the team (and immediately executed)



- ▶ `#pragma omp task`: One **explicit** task is created, packaging code and data for (possible) deferred execution
- ▶ `#pragma omp taskloop`: **Explicit** tasks created for chunks of loop iterations
 - ▶ In both cases, tasks executed by threads in the parallel region

Instructor Social Media

Youtube: Lucas Science



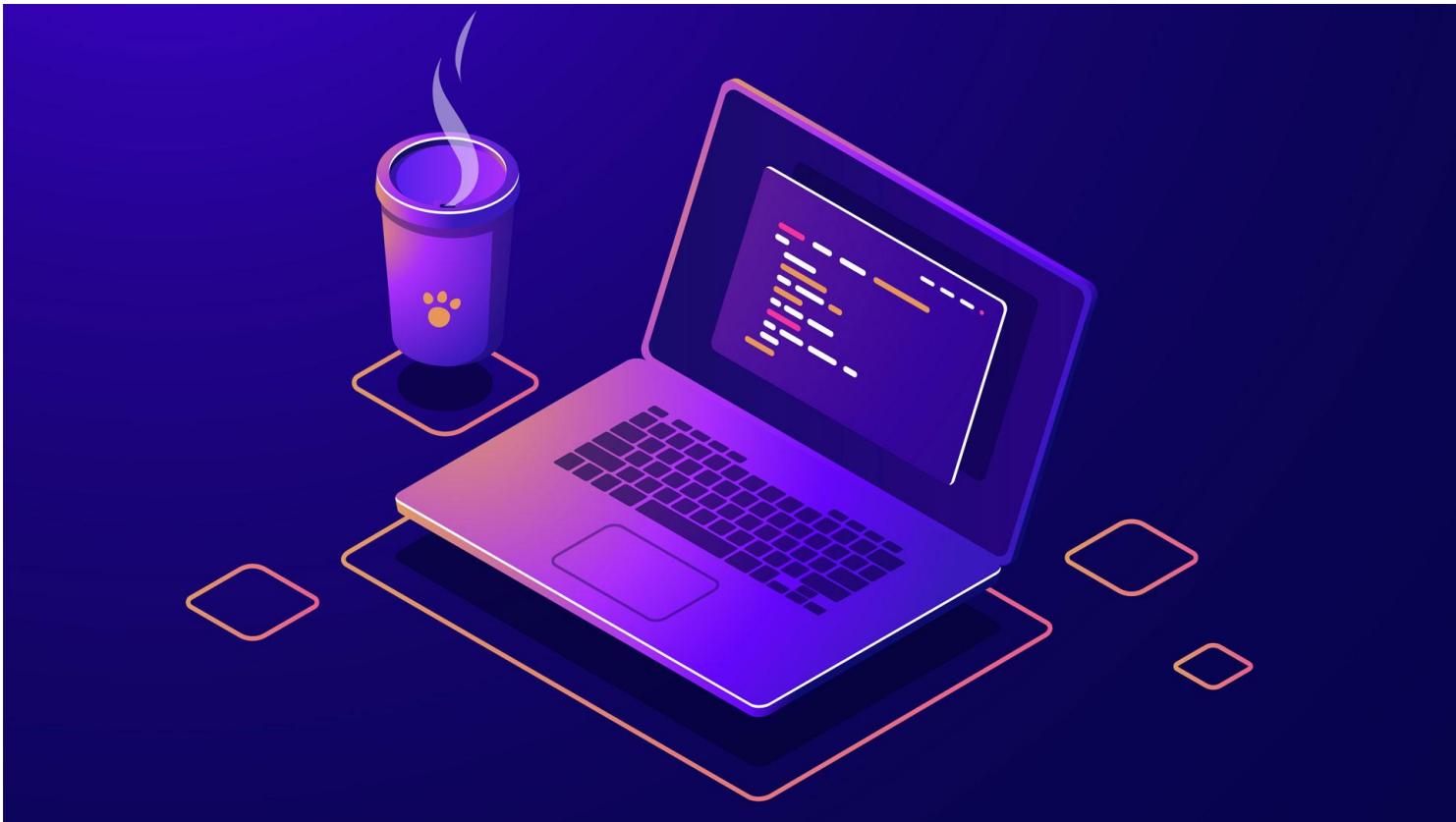
Instagram: lucaasbazilio



Twitter: lucasebazilio



Task Generation Control



Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

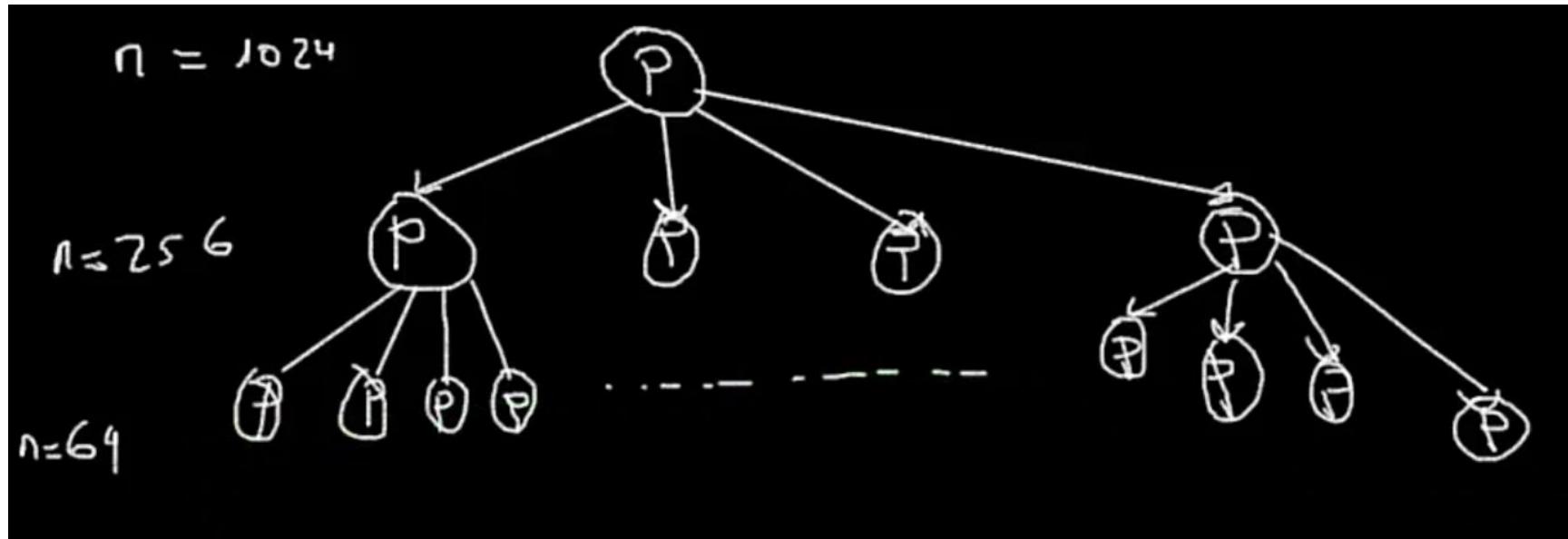
Task Generation Control



Excessive task generation may not be necessary (i.e. cause excessive overhead): need mechanisms to control number of tasks and/or their granularity

- ▶ In iterative task decomposition strategies one can control task granularity by setting the number of iterations executed by each task
- ▶ In recursive task decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (**cut-off control**)
 - ▶ after certain number of recursive calls (static control)
 - ▶ when the size of the vector is too small (static control)
 - ▶ when there are sufficient tasks pending to be executed (dynamic control)

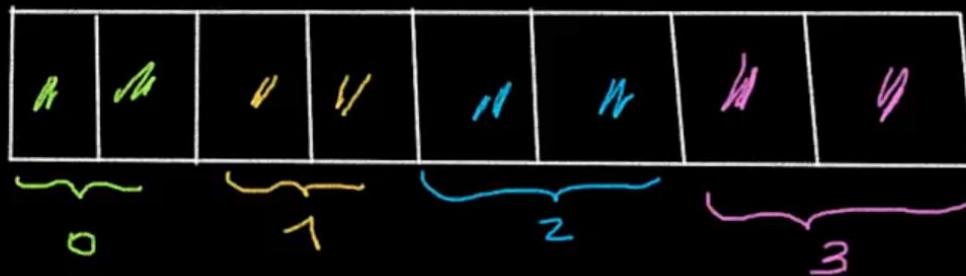
Example of Cut Off (We will study deeply later)



Another Example on the size of vector

$n_t = 4$ threads

$n = 8$



$$BS = \frac{8}{4} = 2$$

Task Generation Control



Iterative task decomposition (1)

Task granularity defined by the number of iterations out of the loop each task executes. For example, using **implicit tasks**:

```
void vector_add(int *A, int *B, int *C, int n) {
    int who = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int BS = n / nt;
    for (int i = who*BS; i < (who+1)*BS; i++)
        C[i] = A[i] + B[i];
}

void main() {
    ...
#pragma omp parallel
vector_add(a, b, c, N);
...
}
```

Each implicit task executes a subset of iterations, based in the thread identifier executing the implicit task and the total number of implicit tasks (i.e., number of threads in the team).

Task Generation Control



Iterative task decomposition (2)

Task granularity defined by the number of iterations each task executes. For example, using **explicit tasks**:

```
void vector_add(int *A, int *B, int *C, int n) {  
    for (int i=0; i< n; i++)  
        #pragma omp task  
        C[i] = A[i] + B[i];  
}  
  
void main() {  
    ...  
    #pragma omp parallel  
    #pragma omp single  
    vector_add(a, b, c, N);  
    ...  
}
```

each explicit task executes a single iteration of the *i* loop, large task creation overhead, very fine granularity!

Task Generation Control



How do we make Explicit Tasks execute a chunk of iterations?

Task Generation Control



Iterative task decomposition (3)

Granularity: chunk of BS loop iterations

- ▶ **Option 1:** requires loop transformation

```
void vector_add(int *A, int *B, int *C, int n) {  
    int BS = ...  
    for (int ii=0; ii< n; ii+=BS)  
        #pragma omp task  
        for (int i = ii; i < min(ii+BS, n); i++)  
            C[i] = A[i] + B[i];  
    }  
    void main() {  
        ...  
        #pragma omp parallel  
        #pragma omp single  
        vector_add(a, b, c, N);  
        ...  
    }
```

Outer loop jumps over chunks of BS iterations, inner loop traverses each chunk

Task Generation Control



Iterative task decomposition (4)

- ▶ **Option 2:** `taskloop` construct to specify tasks out of loop iterations:

```
void vector_add(int *A, int *B, int *C, int n) {
    int BS = ...
    #pragma omp taskloop grainsize(BS)          // or alternatively num_tasks(n/BS)
    for (int i=0; i< n; i++)
        C[i] = A[i] + B[i];
}
void main() {
    #pragma omp parallel
    #pragma omp single
    ... vector_add(a, b, c, N); ...
}
```

- ▶ `grainsize(m)`: each task executes $[min(m, n) \dots 2 \times m]$ consecutive iterations, being n the total number of iterations
- ▶ `num_tasks(m)`: creates as many tasks as $min(m, n)$

Task Generation Control



Iterative task decomposition: uncountable loop

List of elements, traversed using a while loop while not end of list

```
int main() {
    struct node *p;

    p = init_list(n);
    ...

#pragma omp parallel
#pragma omp single
    while (p != NULL) {
        #pragma omp task firstprivate(p) // see note below
        process_work(p);
        p = p->next;
    }
    ...
}
```

Granularity is one iteration, hopefully with sufficient work to amortise task creation overhead.

Note: `firstprivate` needed to capture the value of `p` at task creation time to allow its deferred execution.

Task Generation Control



firstprivate : Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

private : Specifies that each thread should have its own instance of a variable.

Instructor Social Media

Youtube: Lucas Science



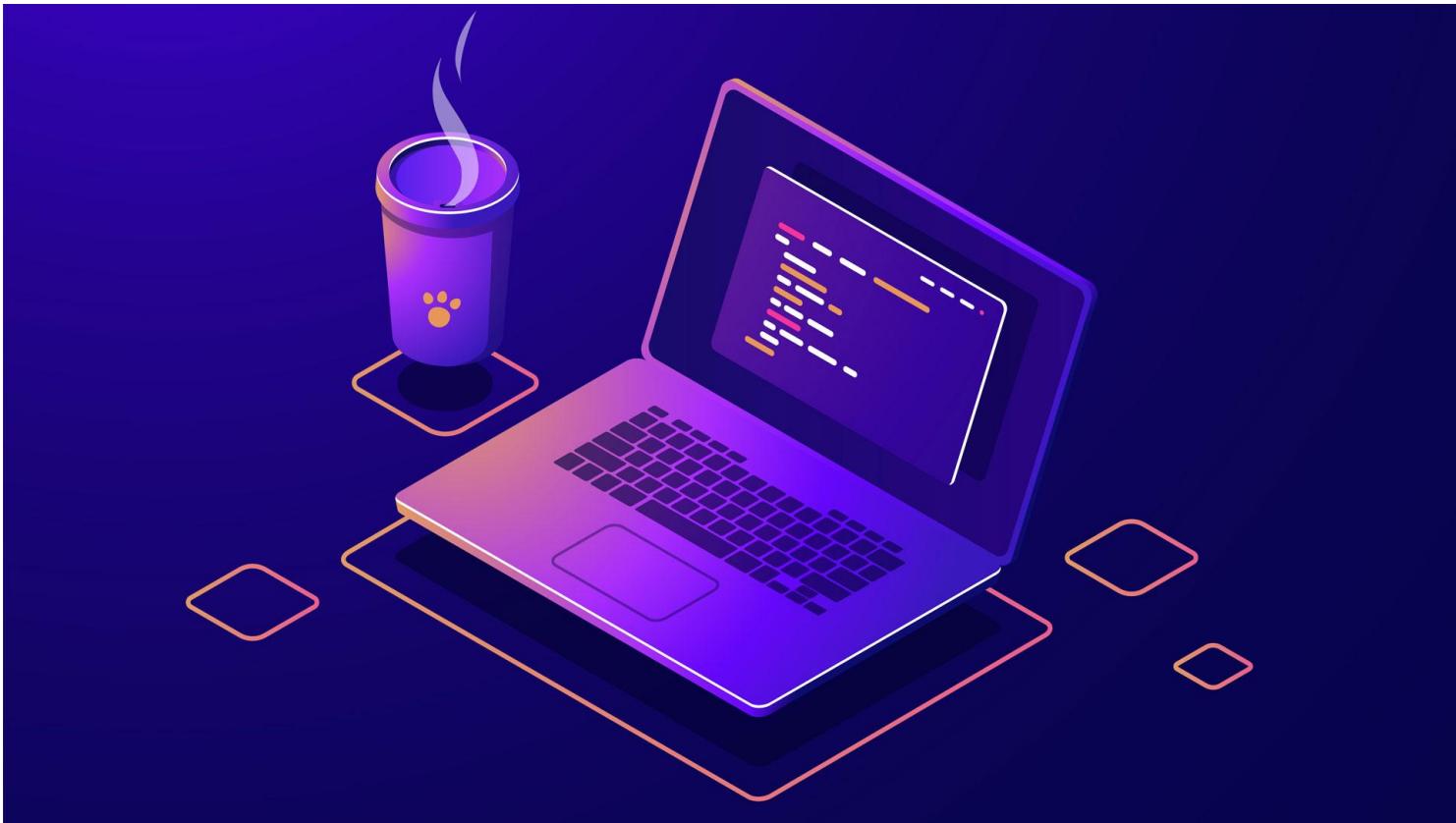
Instagram: lucaasbazilio



Twitter: lucasebazilio



Recursive Task Decomposition in OpenMP



Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control



Excessive task generation may not be necessary (i.e. cause excessive overhead): need mechanisms to control number of tasks and/or their granularity

- ▶ In iterative task decomposition strategies one can control task granularity by setting the number of iterations executed by each task
- ▶ In recursive task decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (**cut-off control**)
 - ▶ after certain number of recursive calls (static control)
 - ▶ when the size of the vector is too small (static control)
 - ▶ when there are sufficient tasks pending to be executed (dynamic control)

Task Generation Control



Recursive task decomposition: divide-and-conquer (1)

Recursively divide the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        dot_product(A, B, n);
}

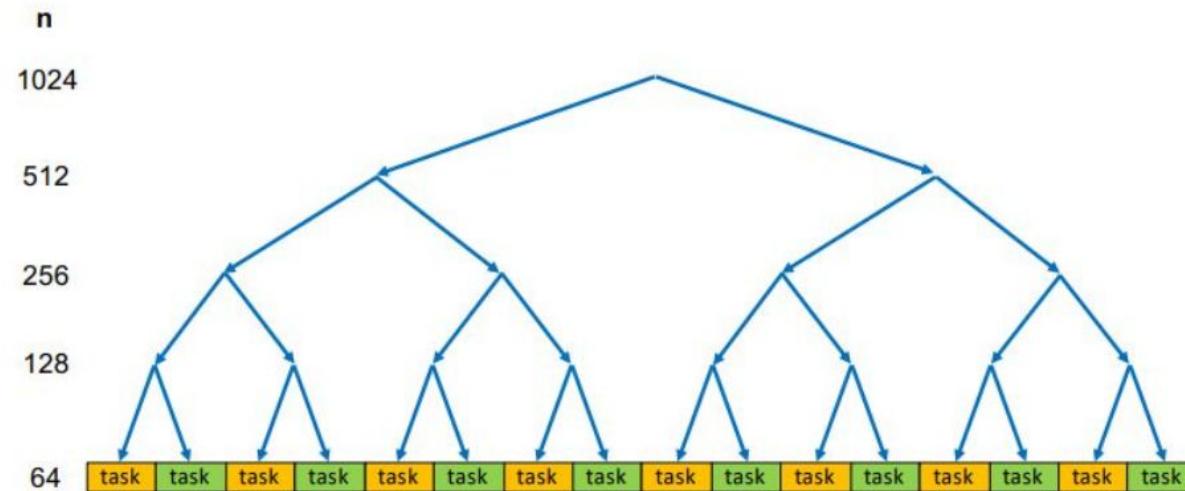
void main() {
    rec_dot_product(a, b, N);
}
```

Task Generation Control



Recursive task decomposition: leaf strategy (1)

A task corresponds with each invocation of `dot_product` once the recursive invocations stop



- ▶ Sequential generation of tasks

Task Generation Control



Recursive task decomposition: leaf strategy (2)

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)

        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        #pragma omp task
        dot_product(A, B, n);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    rec_dot_product(a, b, N);
}
```

Task Generation Control



Recursive task decomposition: leaf strategy (2)

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)

        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        #pragma omp task
        dot_product(A, B, n);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    rec_dot_product(a, b, N);
}
```

Also...



We also need to use the atomic directive.

atomic : ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Task Generation Control



Recursive task decomposition: leaf strategy (3)

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        #pragma omp atomic
        result += A[i] * B[i];
}

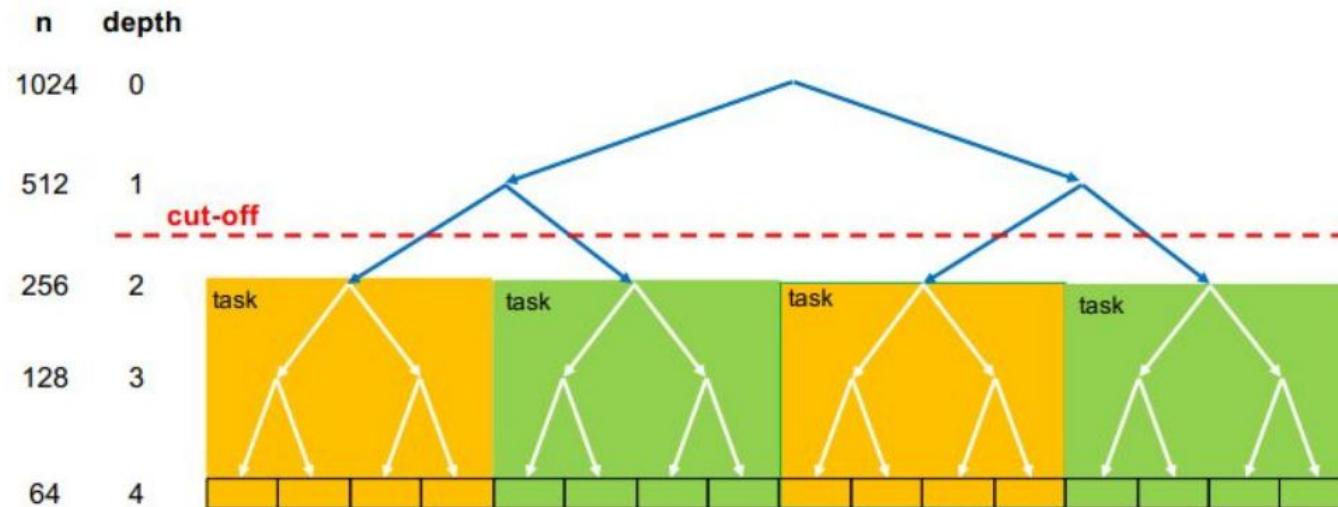
void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        #pragma omp task
        dot_product(A, B, n);
}
```

Task Generation Control



How to control task granularity in leaf strategy (1)

Leaf parallelization with **depth recursion control**



Task Generation Control



How to control task granularity in leaf strategy (2)

Leaf strategy with **depth recursion control**

```
#define CUTOFF 2
...
void rec_dot_product(int *A, int *B, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth == CUTOFF)
            #pragma omp task
        {
            rec_dot_product(A, B, n2, depth+1);
            rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
        else {
            rec_dot_product(A, B, n2, depth+1);
            rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else // if recursion finished, need to check if task has been generated
        if (depth <= CUTOFF)
            #pragma omp task
            dot_product(A, B, n);
        else
            dot_product(A, B, n);
}
...
```

Instructor Social Media

Youtube: Lucas Science



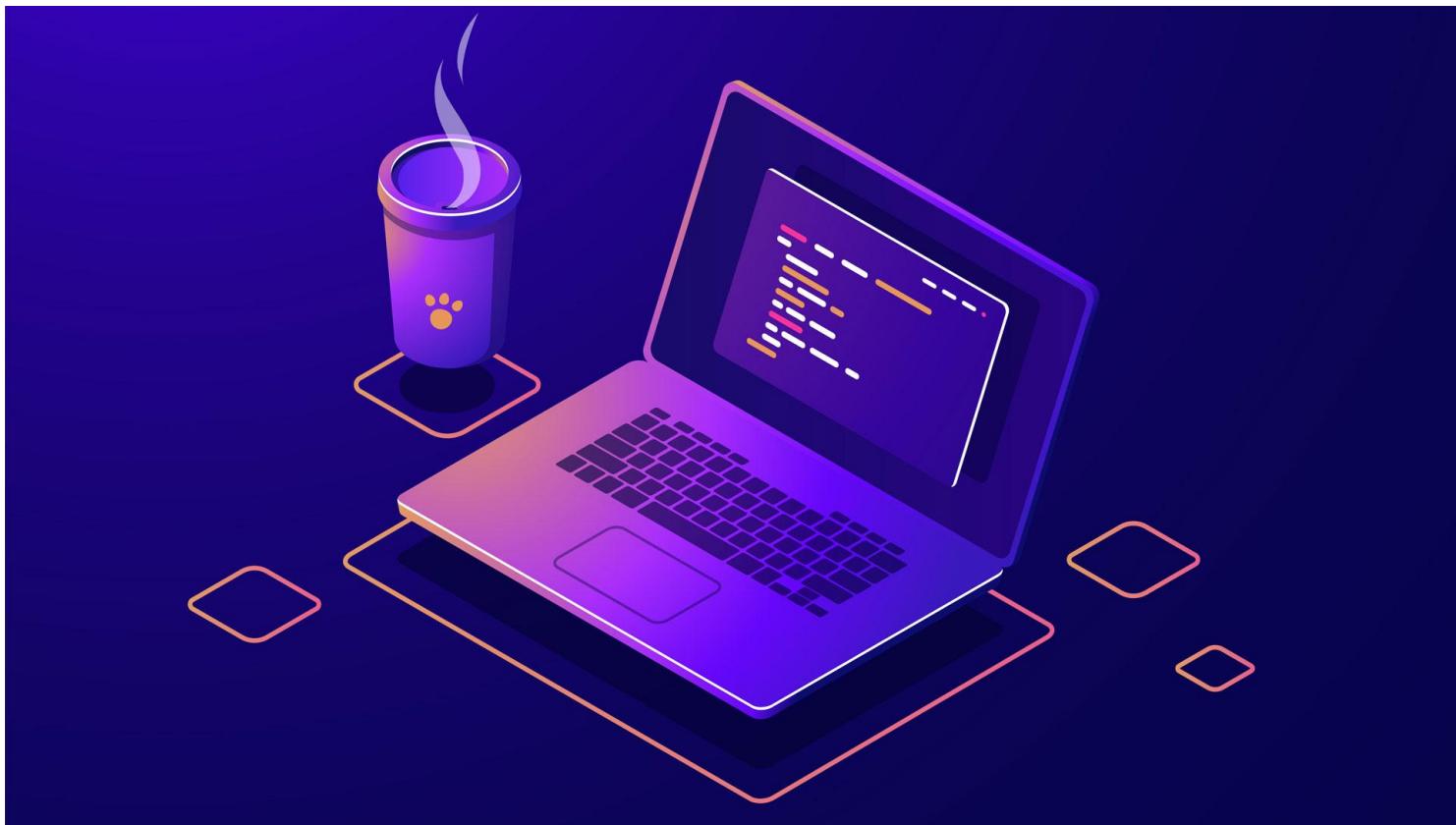
Instagram: lucaasbazilio



Twitter: lucasebazilio



Recursive Task Decomposition in OpenMP



Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control

Recursive task decomposition: divide-and-conquer (1)

Recursively divide the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        result += A[i] * B[i];
}

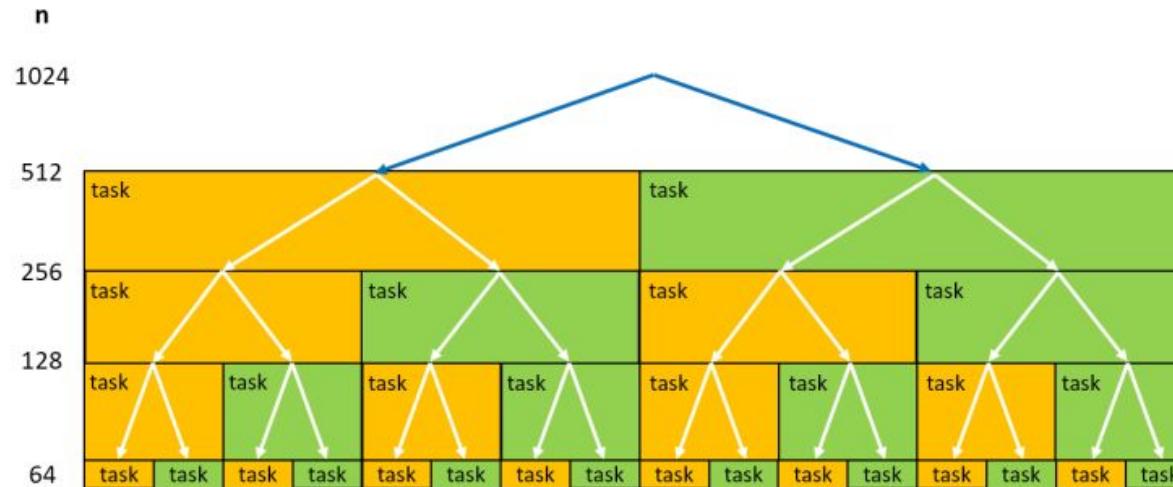
void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        dot_product(A, B, n);
}

void main() {
    rec_dot_product(a, b, N);
}
```

Task Generation Control

Recursive task decomposition: tree strategy (1)

A task corresponds with each invocation of `rec_dot_product`



- ▶ Parallel generation of tasks
- ▶ Granularity: some tasks simply generate new tasks

Task Generation Control

Recursive task decomposition: different sequential code ...

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        tmp1 = rec_dot_product(A, B, n2);
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    result = rec_dot_product(a, b, N);
}
```

Task Generation Control

Recursive task decomposition: tree strategy (2)

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    result = rec_dot_product(a, b, N);
}
```

Shared Clause

shared: Specifies that one or more variables should be shared among all threads.

taskwait: Specifies a wait on the completion of child tasks of the current task.

Task Generation Control

Recursive task decomposition: tree strategy (2)

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

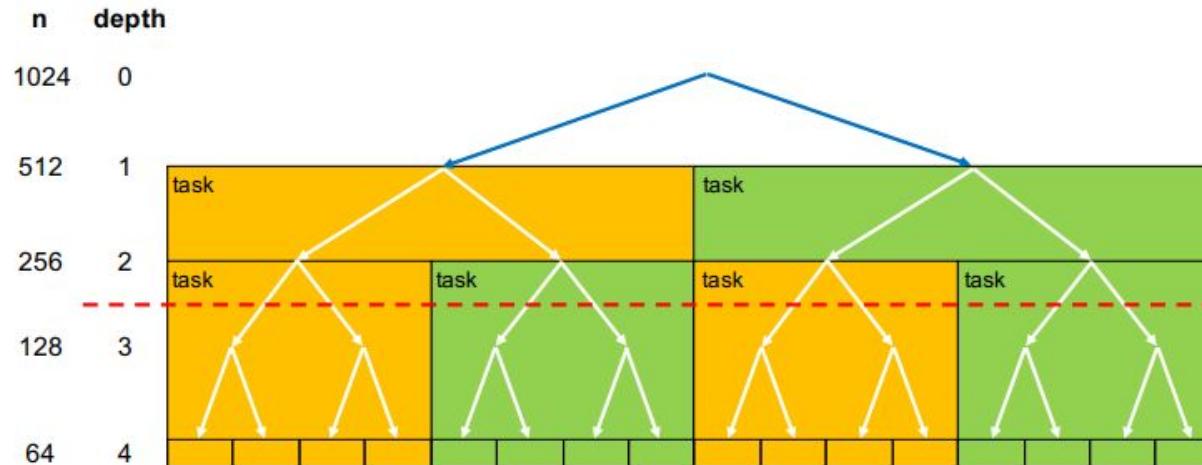
int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    result = rec_dot_product(a, b, N);
}
```

Task Generation Control

How to control task granularity in tree strategy (1)

Tree strategy with **depth recursion control**



Task Generation Control

How to control task granularity in tree strategy (2)

Tree strategy with **depth recursion control**

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
            #pragma omp task shared(tmp1)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else tmp = dot_product(A, B, n);
    return(tmp1+tmp2);
}
```

Task Generation Control

OpenMP support for cut-off

- ▶ `final` clause: If the expression of a `final` clause evaluates to *true* the generated task and **all of its descendent tasks** will be final. The execution of a final task is sequentially **included** in the generating task (but the task is still generated)
- ▶ `omp_in_final()` intrinsic function: it returns true when executed in a final task region; otherwise, it returns false.

Task Generation Control

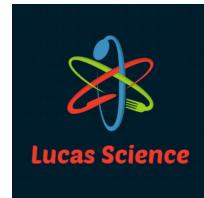
OpenMP support for cut-off: tree strategy

Making use of `omp_in_final`:

```
#define MIN_SIZE 64
#define CUTOFF 3
...
int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (!omp_in_final()) {
            #pragma omp task shared(tmp1) final(depth >= CUTOFF)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2) final(depth >= CUTOFF)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}
```

Instructor Social Media

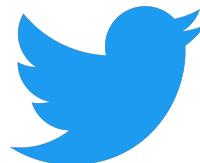
Youtube: Lucas Science



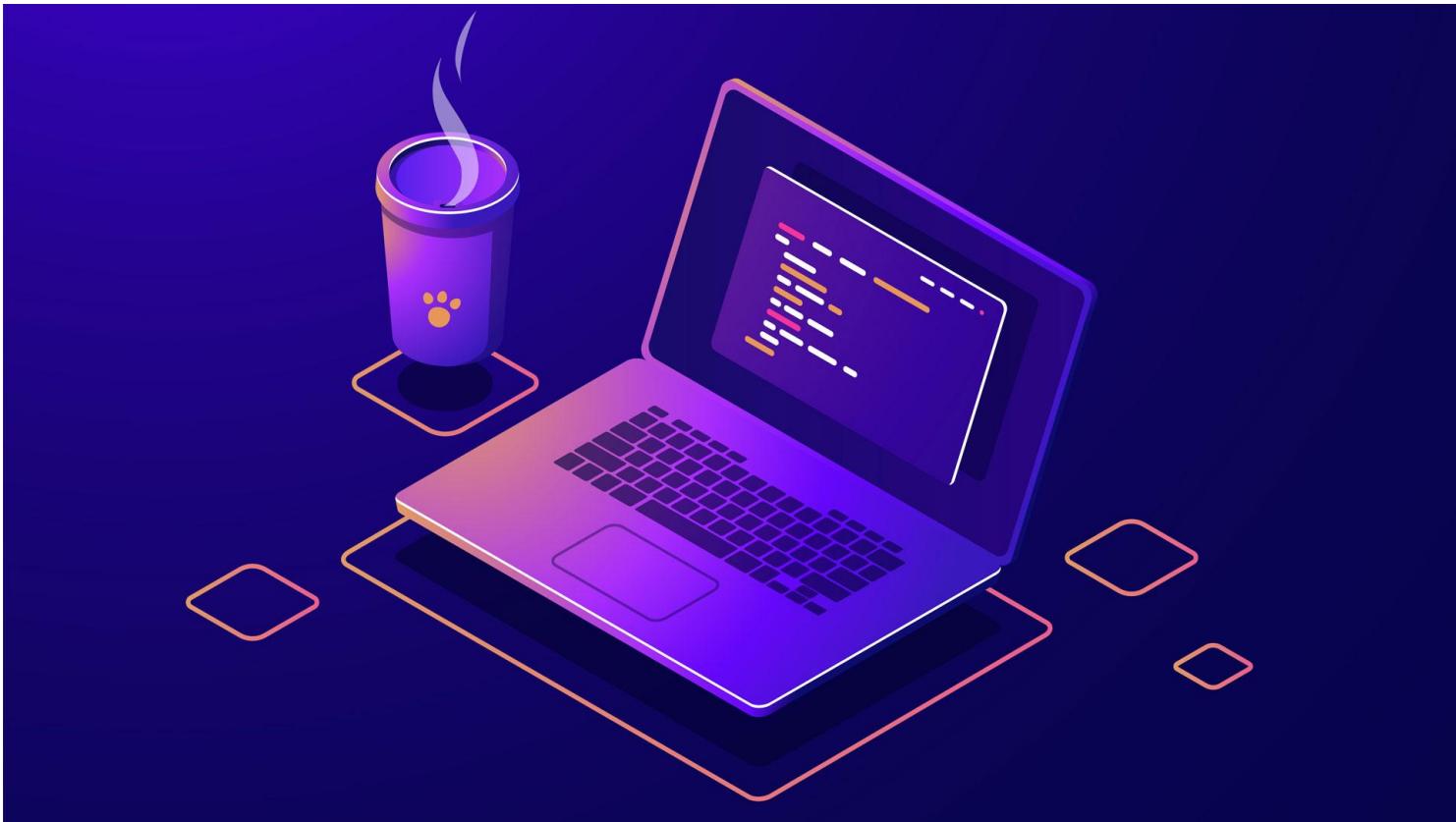
Instagram: lucaasbazilio



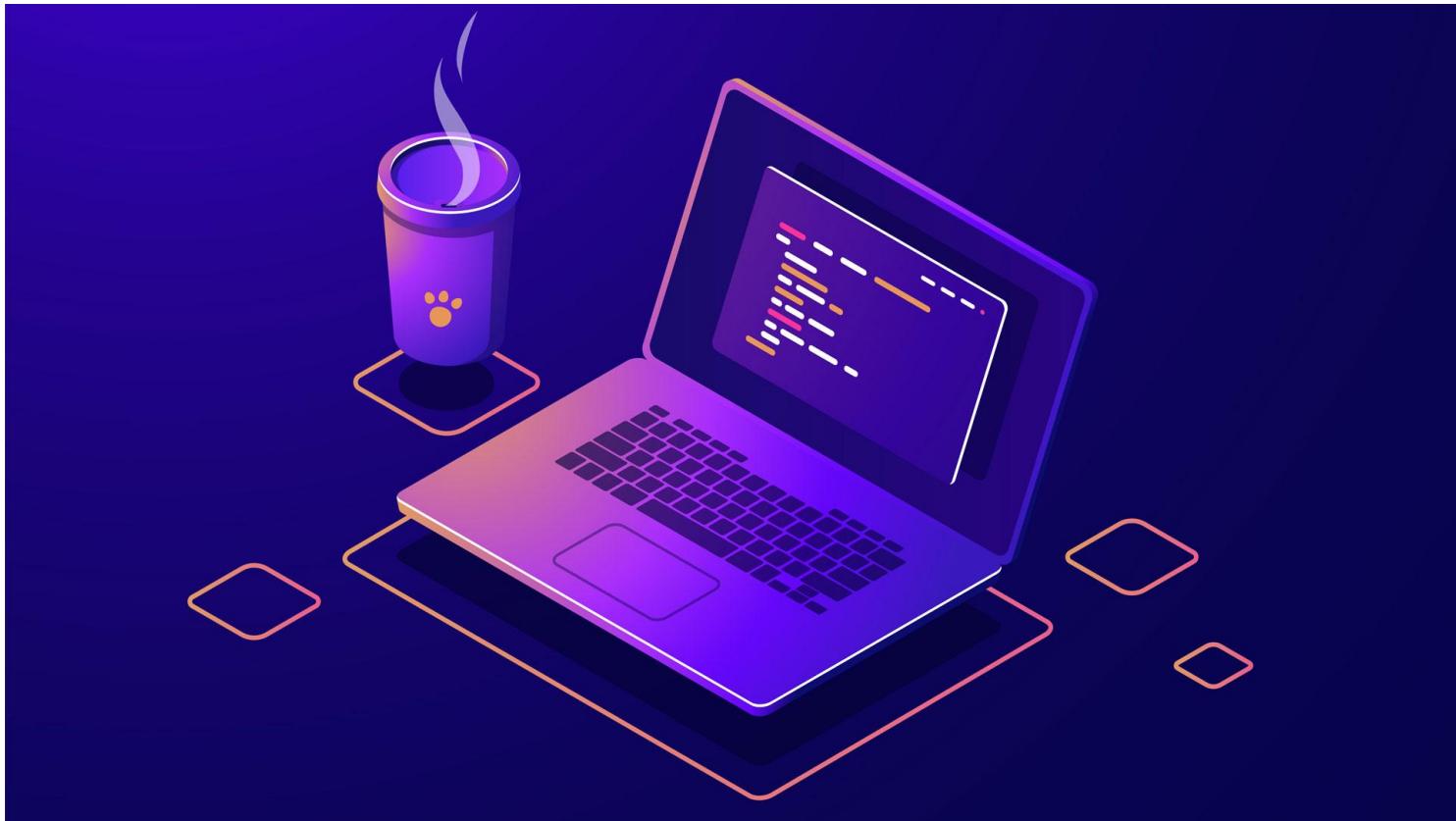
Twitter: lucasebazilio



Depth Recursion Control



Tree Strategy with recursion control



Task Generation Control



- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control



- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control



- Iterative Task Decompositions
- Recursive Task Decompositions

OpenMP®

Task Generation Control



Recursive task decomposition: divide-and-conquer (1)

Recursively divide the problem into smaller sub-problems

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        rec_dot_product(A, B, n2);
        rec_dot_product(A+n2, B+n2, n-n2);
    }
    else
        dot_product(A, B, n);
}

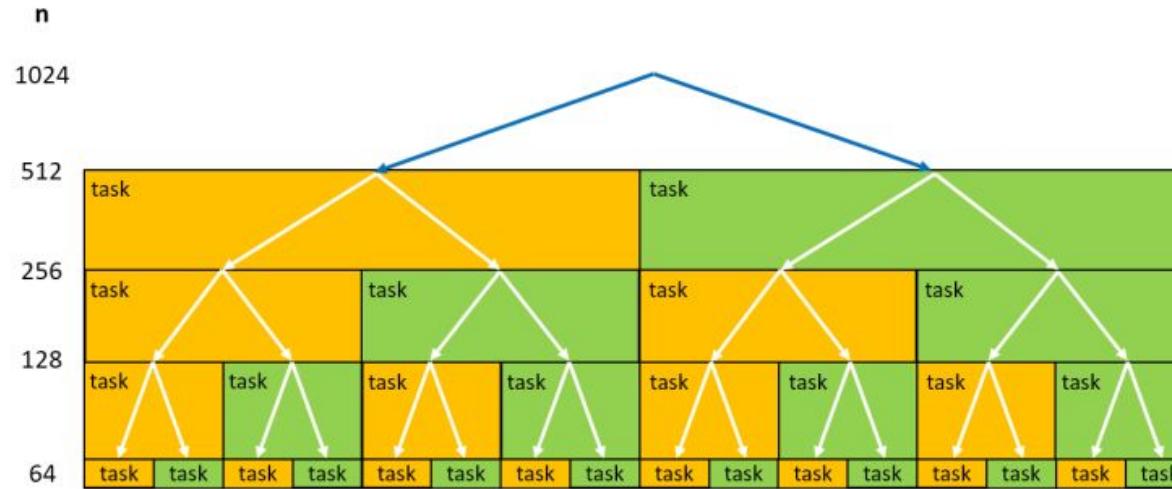
void main() {
    rec_dot_product(a, b, N);
}
```

Task Generation Control



Recursive task decomposition: tree strategy (1)

A task corresponds with each invocation of `rec_dot_product`



- ▶ Parallel generation of tasks
- ▶ Granularity: some tasks simply generate new tasks

Task Generation Control

Recursive task decomposition: different sequential code ...

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        tmp1 = rec_dot_product(A, B, n2);
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    result = rec_dot_product(a, b, N);
}
```

Task Generation Control

Recursive task decomposition: tree strategy (2)

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    result = rec_dot_product(a, b, N);
}
```

Shared Clause

shared: Specifies that one or more variables should be shared among all threads.

taskwait: Specifies a wait on the completion of child tasks of the current task.

Task Generation Control

Recursive task decomposition: tree strategy (2)

```
int dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++) tmp += A[i] * B[i];
    return(tmp);
}

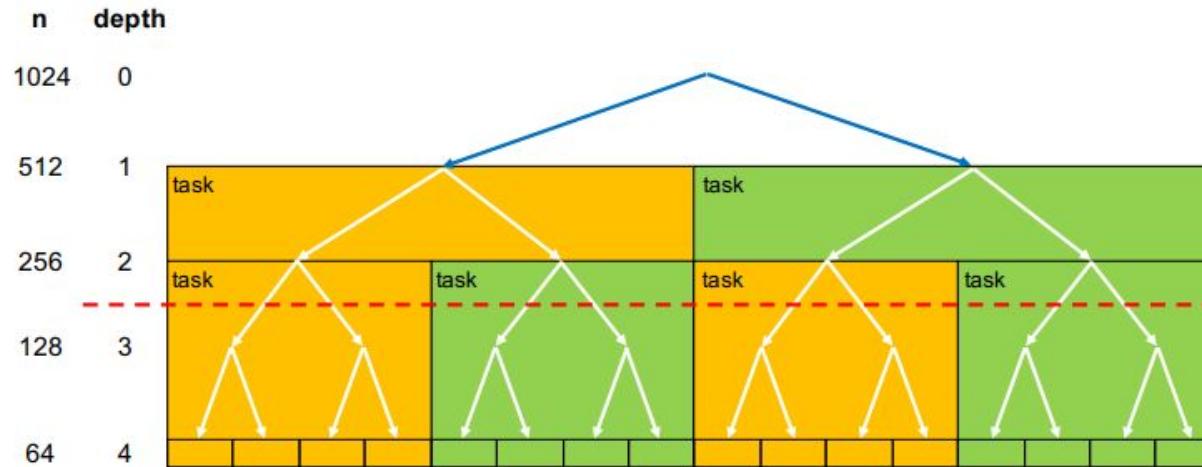
int rec_dot_product(int *A, int *B, int n) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
    } else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}

void main() {
    #pragma omp parallel
    #pragma omp single
    result = rec_dot_product(a, b, N);
}
```

Task Generation Control

How to control task granularity in tree strategy (1)

Tree strategy with **depth recursion control**



Task Generation Control

How to control task granularity in tree strategy (2)

Tree strategy with **depth recursion control**

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
            #pragma omp task shared(tmp1)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else tmp = dot_product(A, B, n);
    return(tmp1+tmp2);
}
```

Task Generation Control

OpenMP support for cut-off

- ▶ `final` clause: If the expression of a `final` clause evaluates to *true* the generated task and **all of its descendent tasks** will be final. The execution of a final task is sequentially **included** in the generating task (but the task is still generated)
- ▶ `omp_in_final()` intrinsic function: it returns true when executed in a final task region; otherwise, it returns false.

Task Generation Control

OpenMP support for cut-off: tree strategy

Making use of `omp_in_final`:

```
#define MIN_SIZE 64
#define CUTOFF 3
...
int rec_dot_product(int *A, int *B, int n, int depth) {
    int tmp1, tmp2 = 0;
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (!omp_in_final()) {
            #pragma omp task shared(tmp1) final(depth >= CUTOFF)
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            #pragma omp task shared(tmp2) final(depth >= CUTOFF)
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
            #pragma omp taskwait
        } else {
            tmp1 = rec_dot_product(A, B, n2, depth+1);
            tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
    }
    else tmp1 = dot_product(A, B, n);
    return(tmp1+tmp2);
}
```

Instructor Social Media

Youtube: Lucas Science



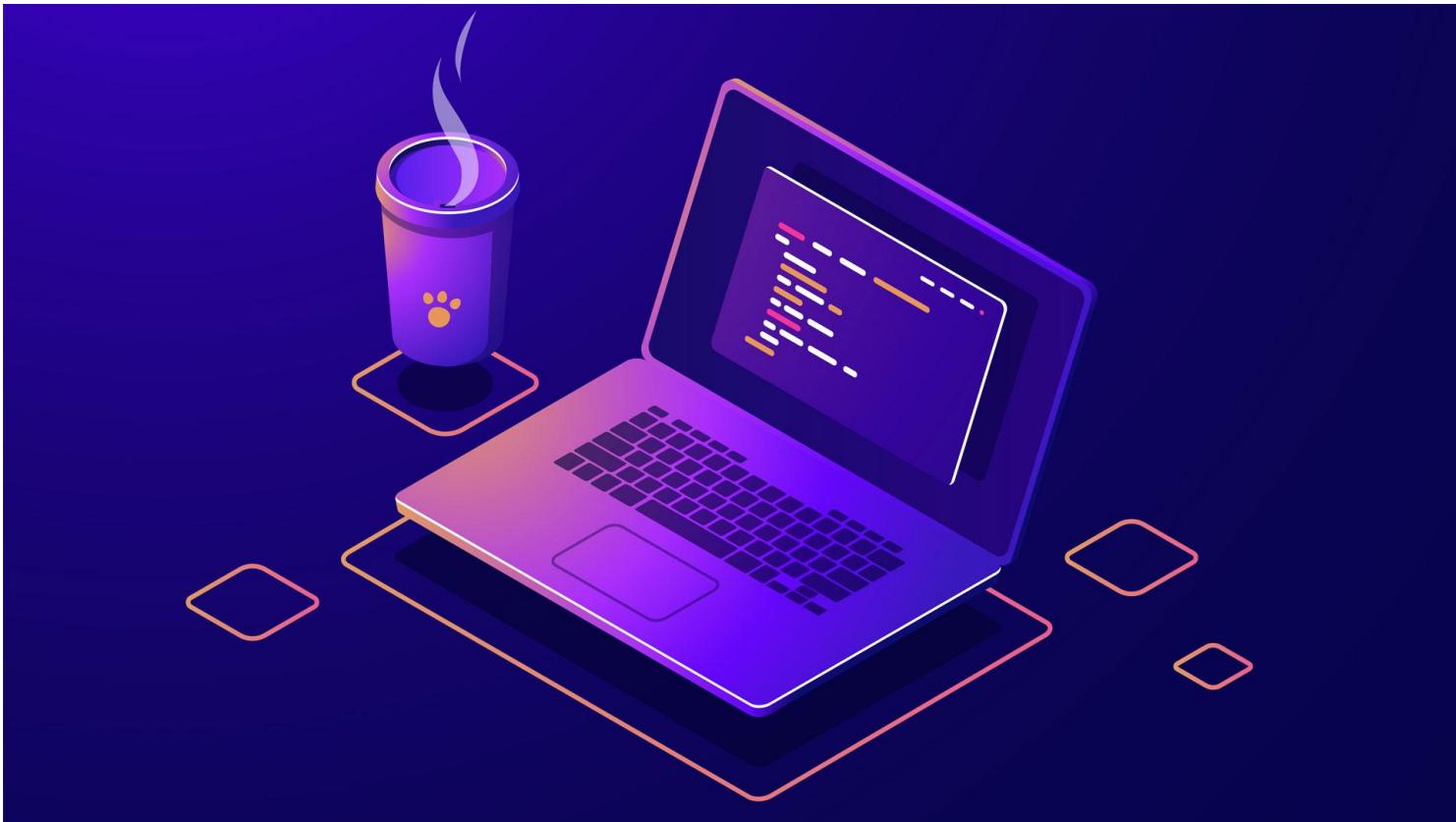
Instagram: lucaasbazilio



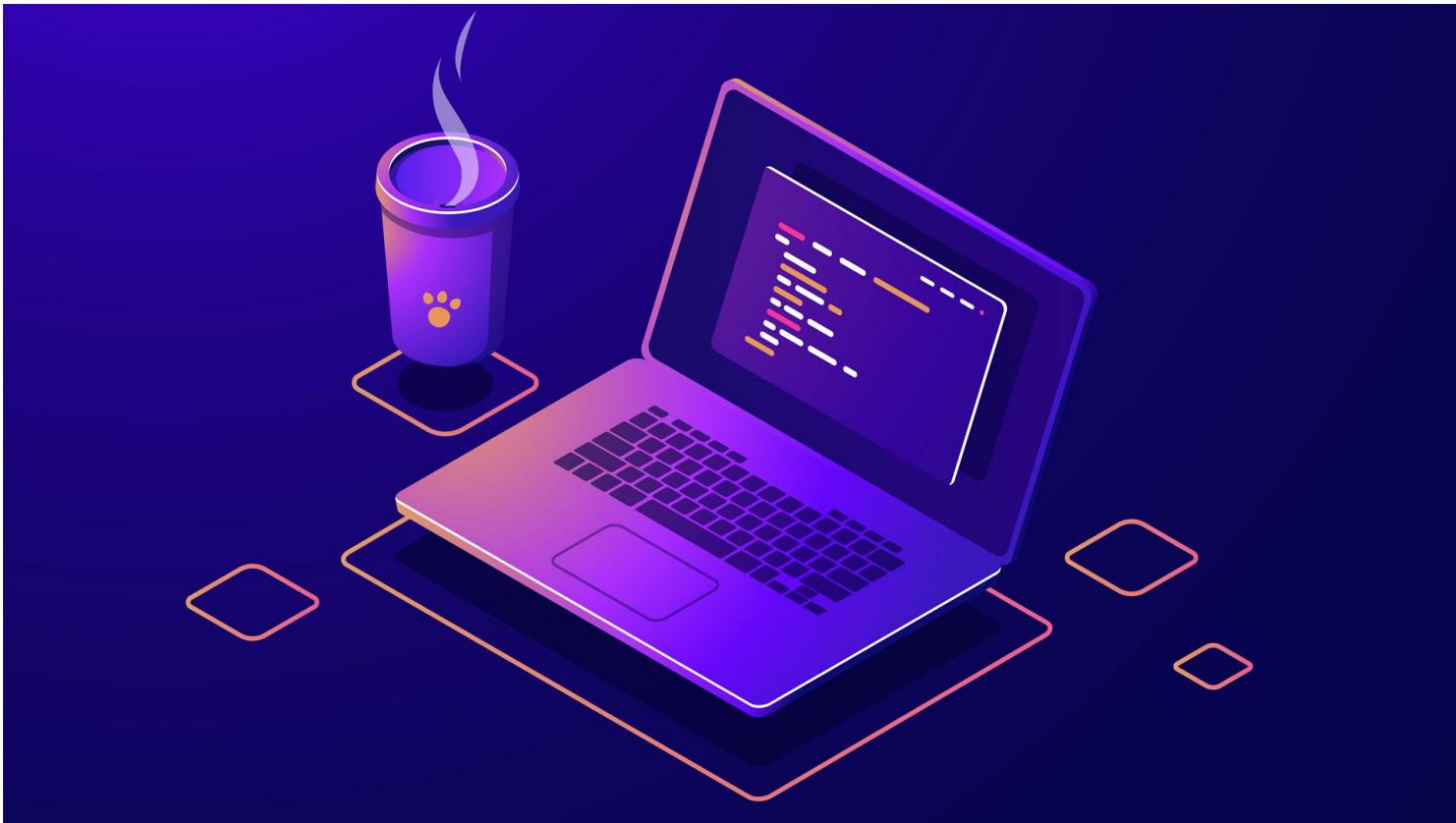
Twitter: lucasebazilio



Reduce overhead and serialization



Atomic Directive





Atomic Directive

Atomic accesses: mechanism to guarantee atomicity in load/store instructions



Atomic Directive

Atomic accesses: mechanism to guarantee atomicity in load/store instructions

```
#pragma omp atomic [update | read | write]  
    expression
```

- ▶ Atomic updates: `x += 1`, `x = x - foo()`, `x[index[i]]++`
- ▶ Atomic reads: `value = *p`
- ▶ Atomic writes: `*p = value`

#pragma omp atomic update



Updates the value of a variable atomically.

Guarantees that only one thread at a time updates the shared variable, avoiding errors from simultaneous writes to the same variable.

An `omp atomic` directive without a clause is equivalent to an `omp atomic update`.



#pragma omp atomic write

Writes the value of a variable atomically.

The value of a shared variable can be written exclusively to avoid errors from simultaneous writes.



#pragma omp atomic read

Reads the value of a variable atomically.

The value of a shared variable can be read safely, avoiding the danger of reading an intermediate value of the variable when it is accessed simultaneously by a concurrent thread.

Instructor Social Media

Youtube: Lucas Science



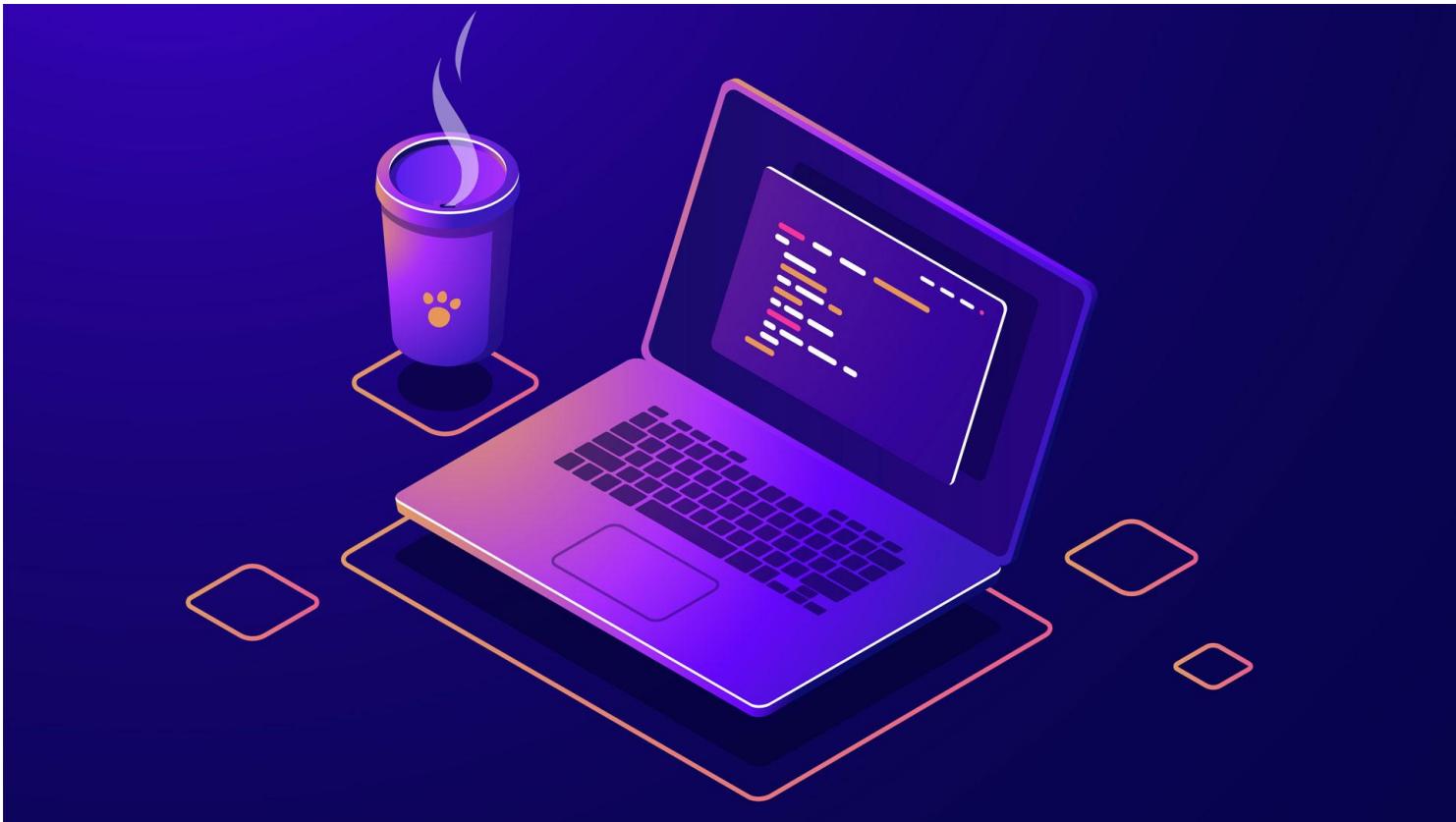
Instagram: lucaasbazilio



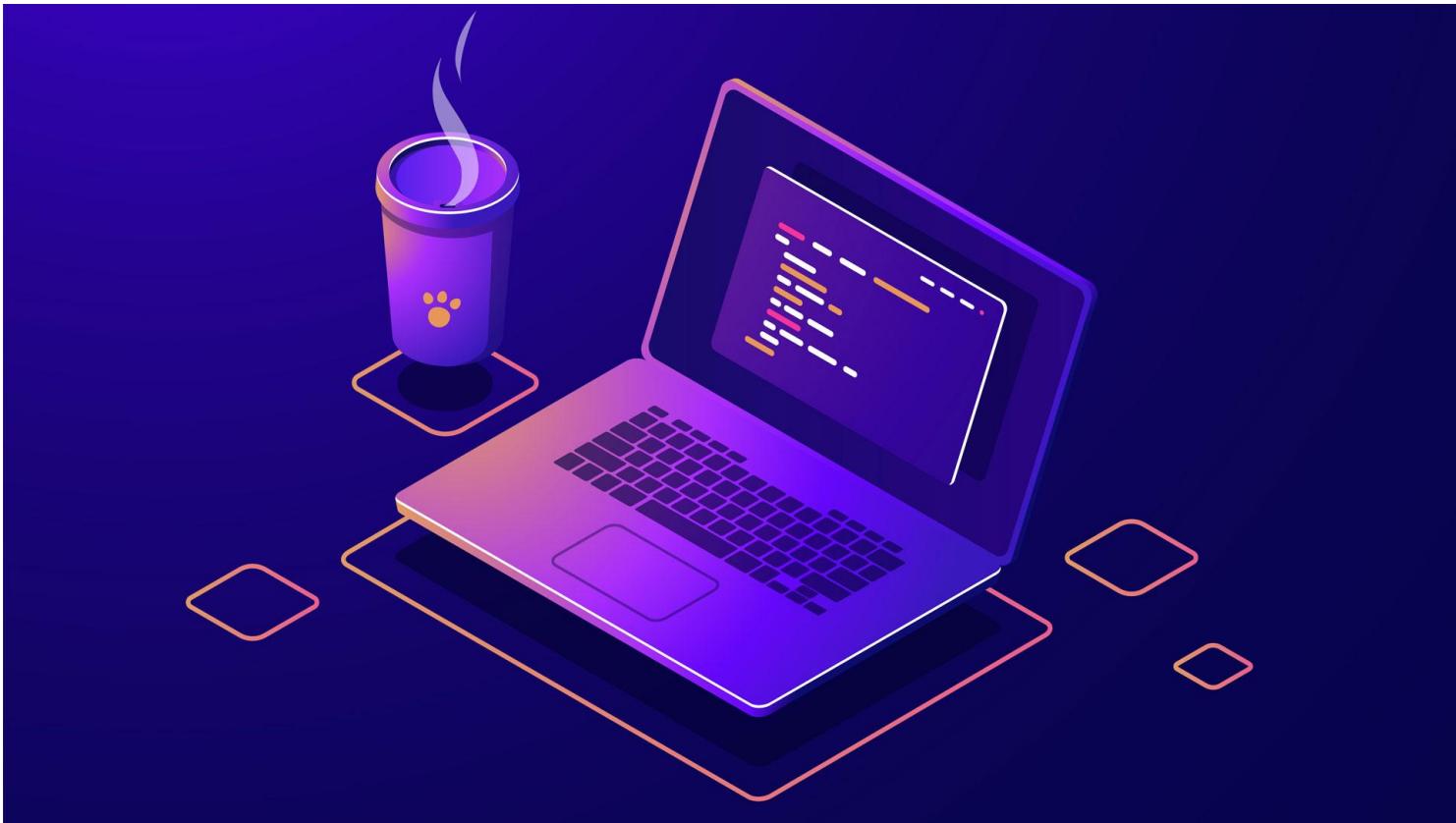
Twitter: lucasebazilio



Reduce overhead and serialization



Critical Directive





Critical Directive

```
#pragma omp critical [(name)]
    structured block
```

- ▶ Provides a region of mutual exclusion where only one thread can be working at any given time
- ▶ By default all critical regions are the same
- ▶ Multiple mutual exclusion regions by providing them with a name
 - ▶ Only those with the same name synchronize



Example: Computation of Pi

```
void main ()
{
    int i, id;
    double x, pi, sum=0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i - 0.5)*step;
            #pragma omp critical
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```



Critical Directive

```
int x=1,y=0;  
#pragma omp parallel num_threads(4)  
{  
#pragma omp critical (x)  
    x++; ←  
#pragma omp critical (y)  
    y++; ←  
}
```

Different names: One thread can update `x` while another updates `y`

Instructor Social Media

Youtube: Lucas Science



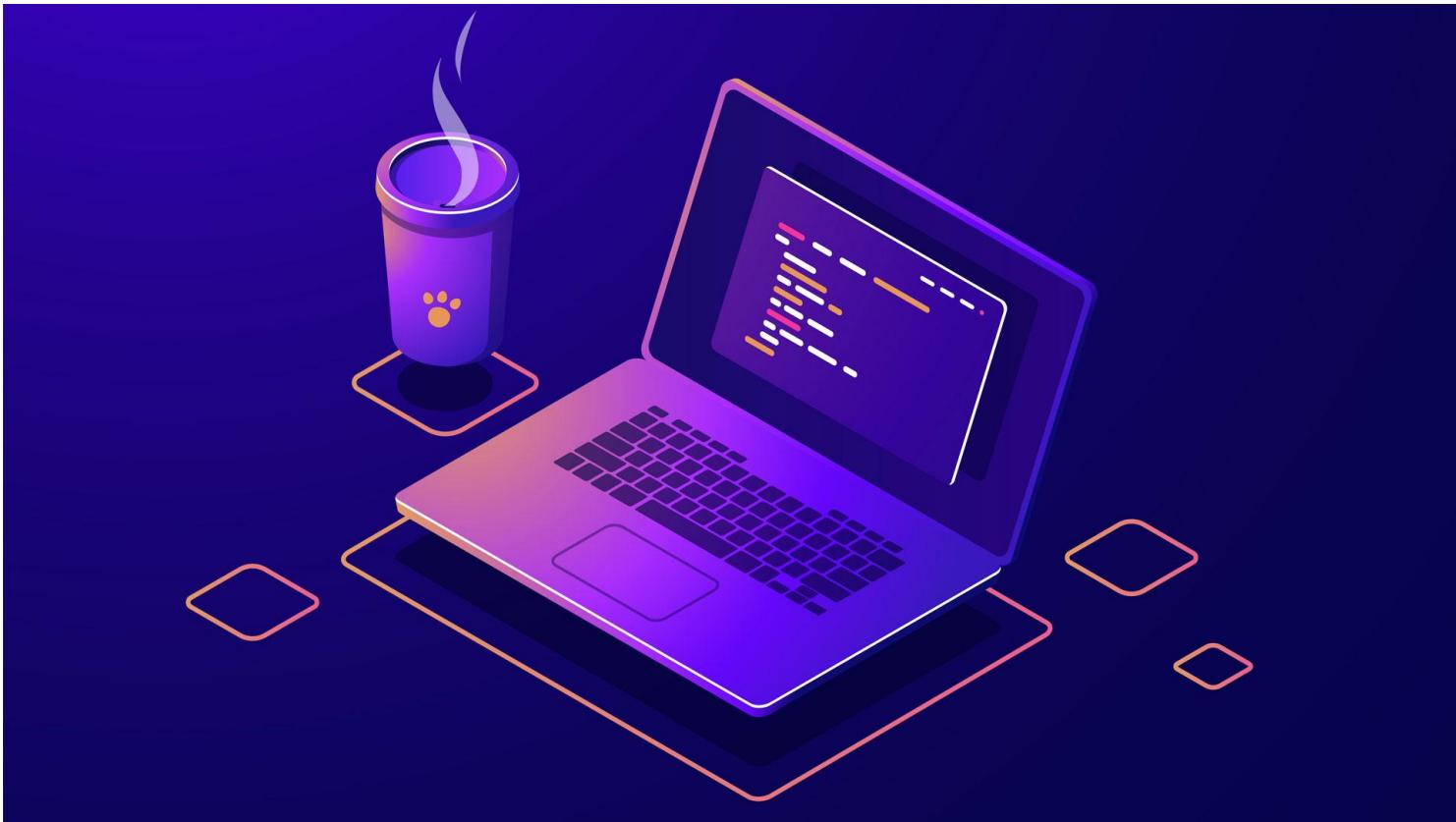
Instagram: lucaasbazilio



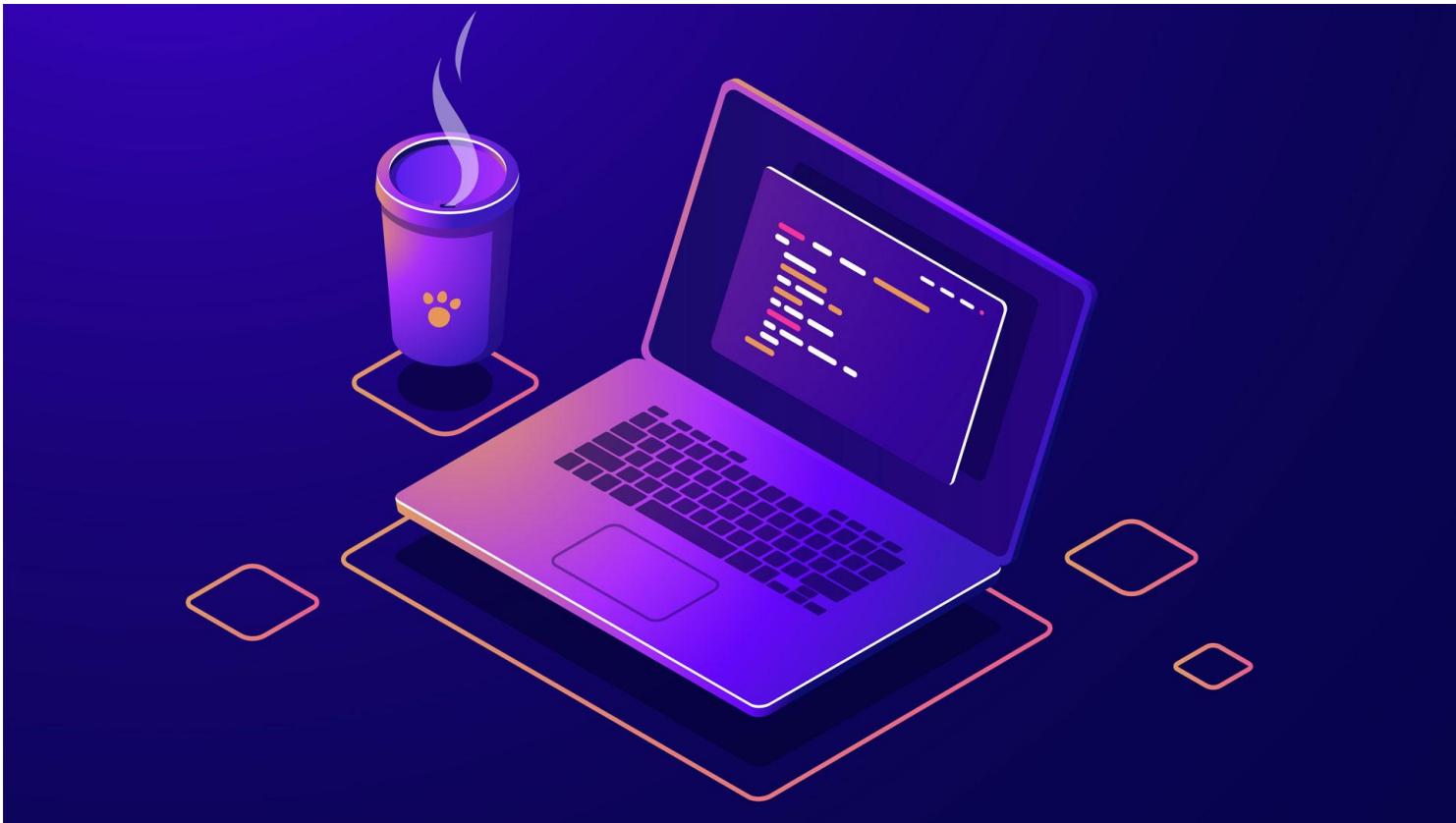
Twitter: lucasebazilio



Reduce overhead and serialization



Reduction Clause





Reduction Clause

Reduction is a very common pattern where all threads accumulate values into a single variable

```
reduction(operator: list)
```

- ▶ Valid operators are: `+,-,*|,||,&,&&,^,min, max`
- ▶ The compiler creates a `private` copy of each variable in `list` that is properly initialized to the identity value
- ▶ At the end of the region, the compiler ensures that the `shared` variable is properly (and safely) updated with the partial values of each thread, using the specified operator



Example: Computation of Pi

```
void main ()
{
    int i, id;
    double x, pi, sum;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel private(x, i, id) reduction(+:sum)
    {
        id = omp_get_thread_num();
        for (i=id+1; i<=num_steps; i=i+NUM_THREADS) {
            x = (i - 0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = sum * step;
}
```



Specifying reduction operations in explicit tasks generated with either task:

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup task_reduction(+: sum)
    for (i=0; i< SIZE; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
        sum += X[i];
}
```

or taskloop:

```
#pragma omp parallel
#pragma omp single
{
    // implicit taskgroup in taskloop construct
    #pragma omp taskloop reduction(+: sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];
}
```

Instructor Social Media

Youtube: Lucas Science



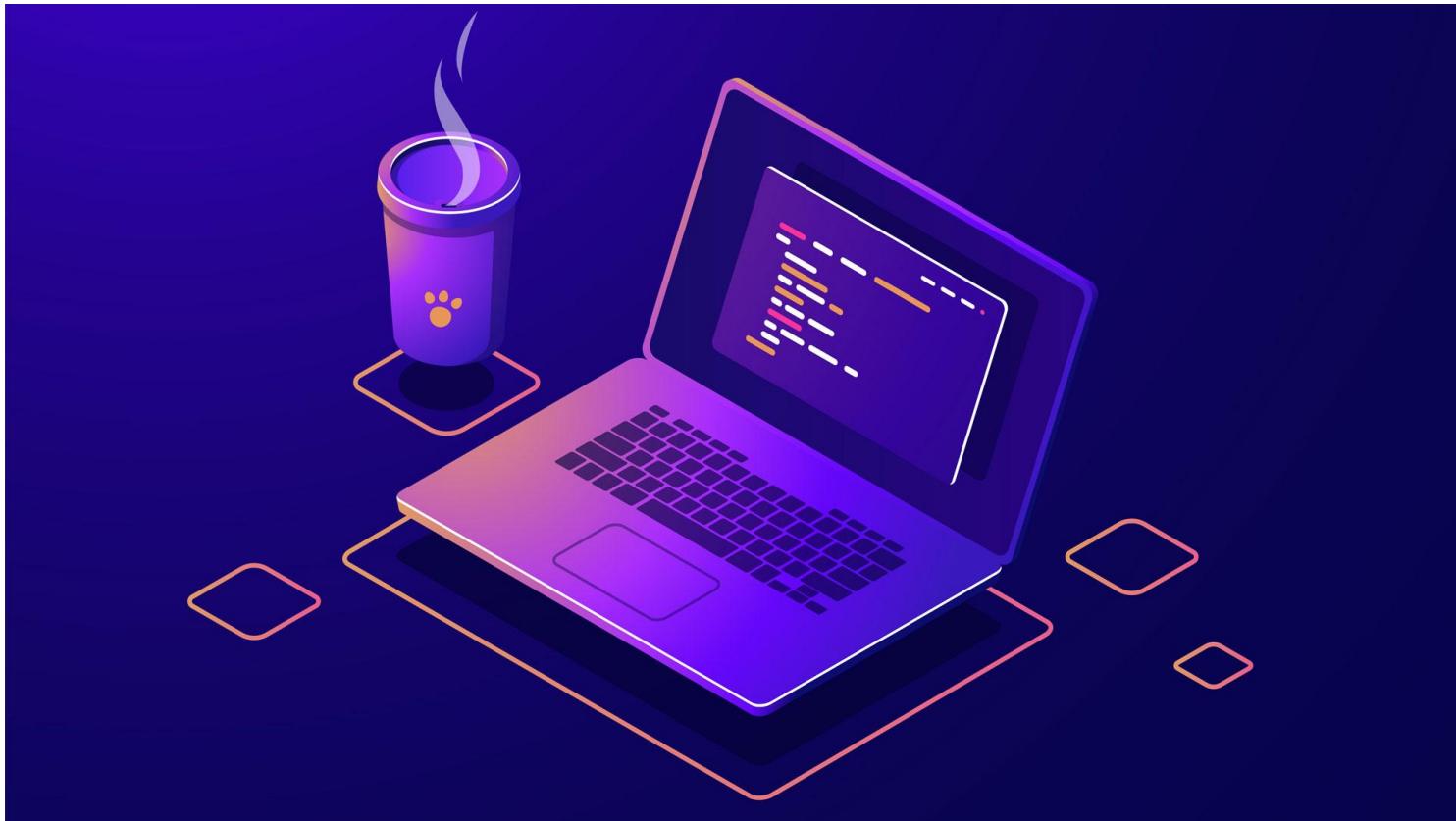
Instagram: lucaasbazilio



Twitter: lucasebazilio



Locks





Locks

Locks: special variables that live in memory with two basic operations:

- ▶ Acquire: while a thread has the lock, nobody else gets it; this allows the thread to do its work in private, not bothered by other threads
- ▶ Release: allow other threads to acquire the lock and do their work (one at a time) in private

Type definition and intrinsics:

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)

void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)

int omp_test_lock(omp_lock_t *lock)
```



Locks

OpenMP provides `lock` primitives for low-level synchronization

<code>omp_init_lock</code>	Initialize the lock
<code>omp_set_lock</code>	Acquires the lock
<code>omp_unset_lock</code>	Releases the lock
<code>omp_test_lock</code>	Tries to acquire the lock (won't block)
<code>omp_destroy_lock</code>	Frees lock resources



Example

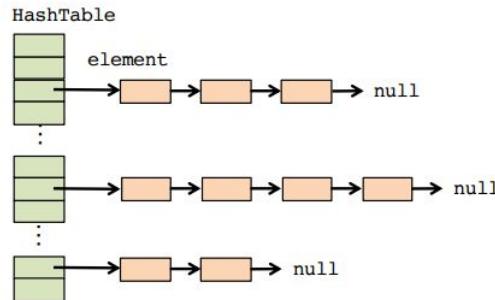
```
#include <omp.h>
void foo ()
{
    omp_lock_t lock;

    omp_init_lock(&lock); ← Lock must be initialized before being used
#pragma omp parallel
{
    omp_set_lock(&lock);
    // mutual exclusion region ← Only one thread at a time here
    omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
}
```

Reducing task interactions: serialization



Example: inserting elements in hash table defined as a collection of linked lists



```
typedef struct {
    int data;
    element *next;
} element;

int dataTable[SIZE_TABLE];
element * HashTable[SIZE_HASH];

for (i = 0; i < SIZE_TABLE; i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    insert_element (dataTable[i], index, HashTable);
}
```

Reducing task interactions: serialization



Easily parallelizable using an iterative task decomposition using taskloop. However ...

- ▶ ... updates to the list in any particular slot must be protected to prevent a race condition

```
typedef struct {
    int data;
    element *next;
} element;

int dataTable[SIZE_TABLE];
element * HashTable[SIZE_HASH];

#pragma omp taskloop
for (i = 0; i < elements; i++) {
    int index = hash_function (dataTable[i], SIZE_HASH);
    #pragma omp critical // atomic not possible here
    insert_element (dataTable[i], index, HashTable);
}
```

- ▶ Serialization in the insertion of elements

Reducing task interactions: serialization



Associate a lock variable with each slot in the hash table, protecting the chain of elements in an slot

```
omp_lock_t hash_lock[SIZE_HASH];  
  
#pragma omp parallel  
#pragma omp single  
{  
    for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);  
  
    #pragma omp taskloop  
    for (i = 0; i < SIZE_TABLE; i++) {  
        int index = hash_function (dataTable[i], SIZE_HASH);  
        omp_set_lock (&hash_lock[index]);  
        insert_element (dataTable[i], index, HashTable);  
        omp_unset_lock (&hash_lock[index]);  
    }  
  
    for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);  
}
```

Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Instructor Social Media

Youtube: Lucas Science



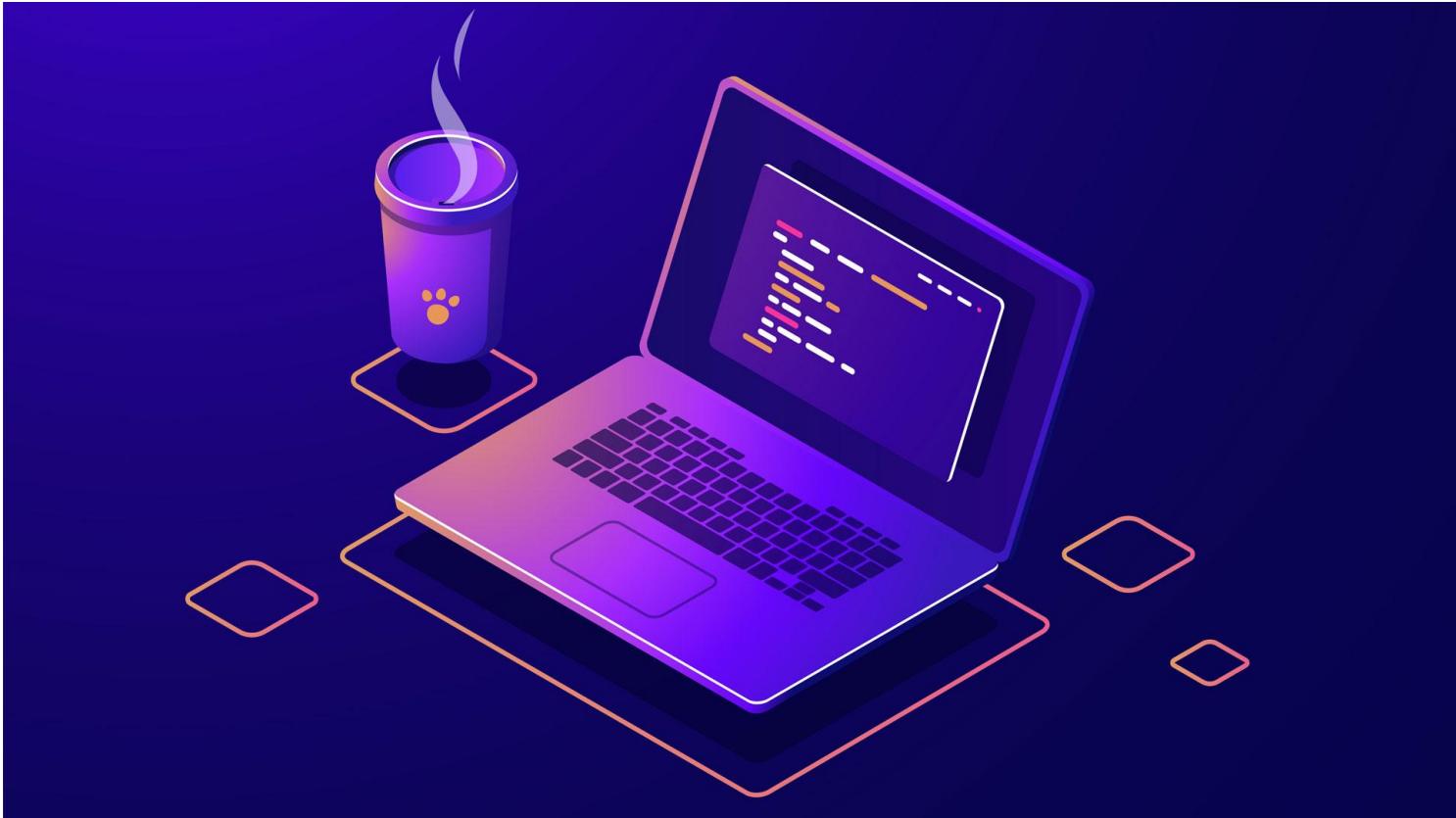
Instagram: lucaasbazilio



Twitter: lucasebazilio



Taskwait and Taskgroup





Taskwait and Taskgroup

- ▶ Task barriers:
 - ▶ taskwait: Suspends the execution of the current task, waiting on the completion of its **child tasks**. The taskwait construct is a stand-alone directive.
 - ▶ taskgroup: Suspends the execution of the current task at the end of structured block, waiting on the completion of **child tasks** of the current task **and their descendant** tasks.



Example of taskwait

```
#pragma omp taskwait  
  
#pragma omp task {}      // T1  
#pragma omp task {}      // T2  
{  
    #pragma omp task {} // T3  
}  
#pragma omp task {}      // T4  
  
#pragma omp taskwait
```

Only T1, T2 and T4 are guaranteed to have finished here



Example of taskgroup

```
#pragma omp taskgroup  
structured block
```

```
#pragma omp task {} // T1  
#pragma omp taskgroup  
{  
    #pragma omp task // T2  
    {  
        #pragma omp task {} // T3  
    }  
    #pragma omp task {} // T4  
}
```

Only T2, T3 and T4 are guaranteed to have finished here



taskwait vs. taskgroup

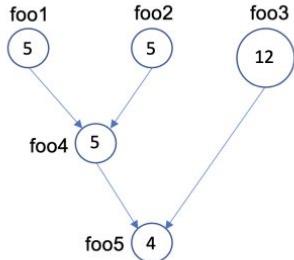
```
#pragma omp task {}          // T1
#pragma omp task            // T2
{
    #pragma omp task {} // T3
}
#pragma omp task {}          // T4

#pragma omp taskwait
// Only T1, T2 and T4 are guaranteed to have finished at this point when T5 is created
#pragma omp task {}          // T5

#pragma omp task {}          // T1
#pragma omp taskgroup
{
    #pragma omp task          // T2
    {
        #pragma omp task {} // T3
    }
    #pragma omp task {}    // T4
}
// Only T2, T3 and T4 are guaranteed to have finished at this point when T5 is created
#pragma omp task {}          // T5
```

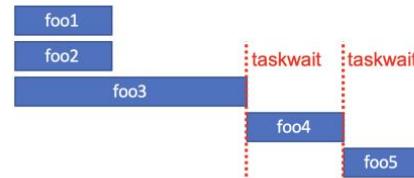
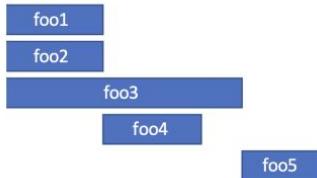


Given a TDG to implement with the OpenMP tasking model:



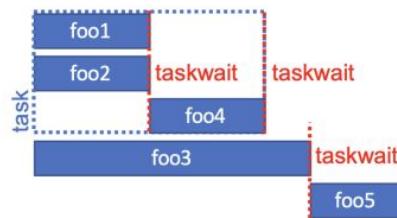
```
#pragma omp task  
foo1()  
#pragma omp task  
foo2()  
#pragma omp task  
foo3()  
#pragma omp taskwait  
#pragma omp task  
foo4()  
#pragma omp taskwait  
#pragma omp task  
foo5()
```

```
#pragma omp task  
foo1()  
#pragma omp task  
foo2()  
#pragma omp taskwait  
#pragma omp task  
foo3()  
#pragma omp task  
foo4()  
#pragma omp taskwait  
#pragma omp task  
foo5()
```

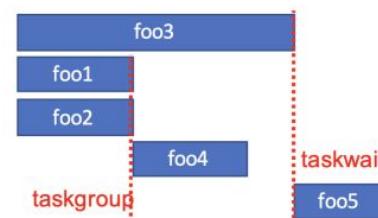




```
#pragma omp task
{
    #pragma omp task
    foo1()
    #pragma omp task
    foo2()
    #pragma omp taskwait
    #pragma omp task
    foo4()
    #pragma omp taskwait
}
#pragma omp task
foo3()
#pragma omp taskwait
```

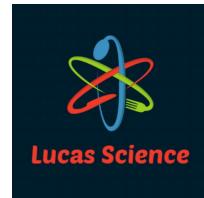


```
#pragma omp task
foo3()
#pragma omp taskgroup
{
    #pragma omp task
    foo1()
    #pragma omp task
    foo2()
}
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```



Instructor Social Media

Youtube: Lucas Science



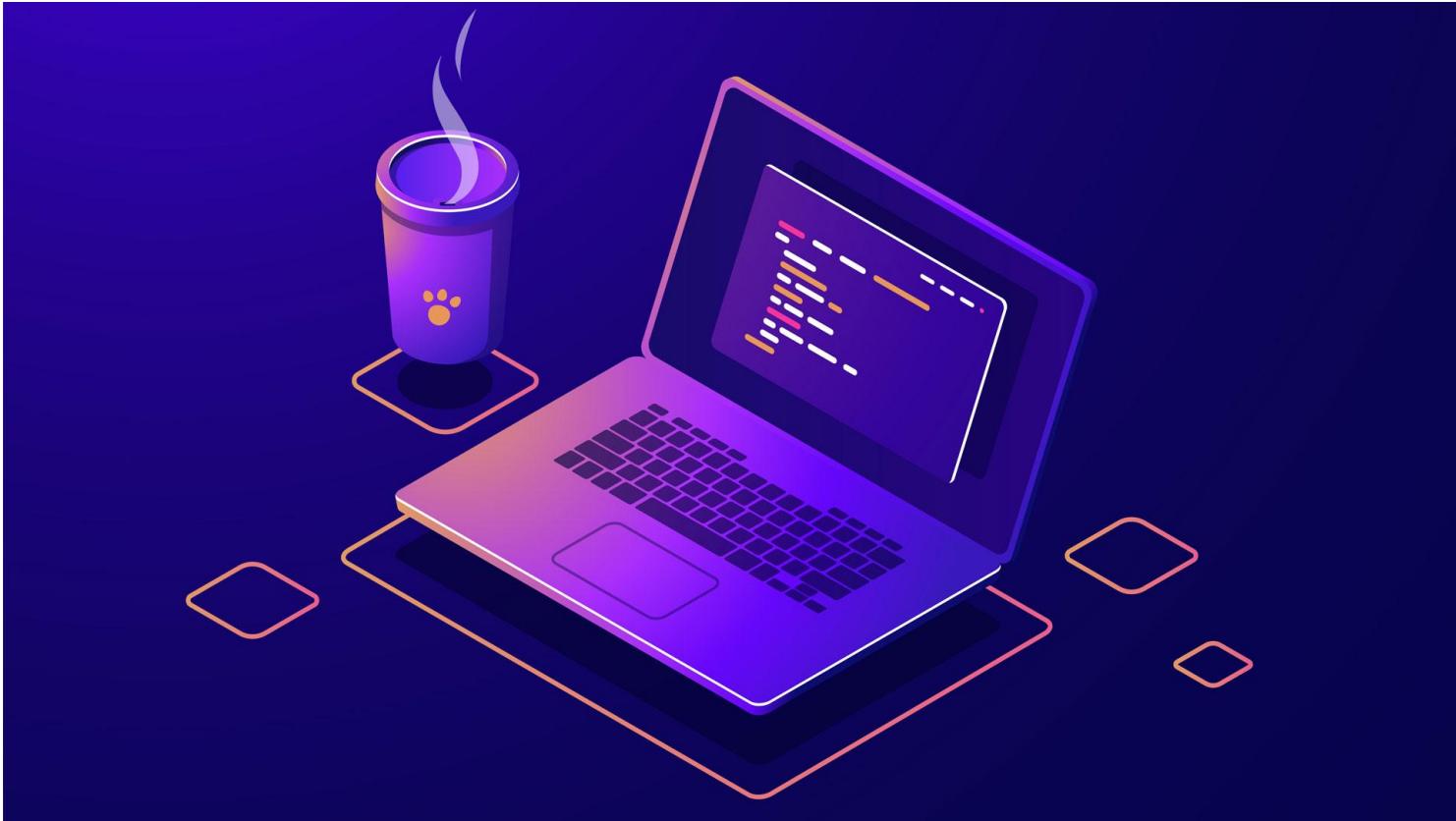
Instagram: lucaasbazilio



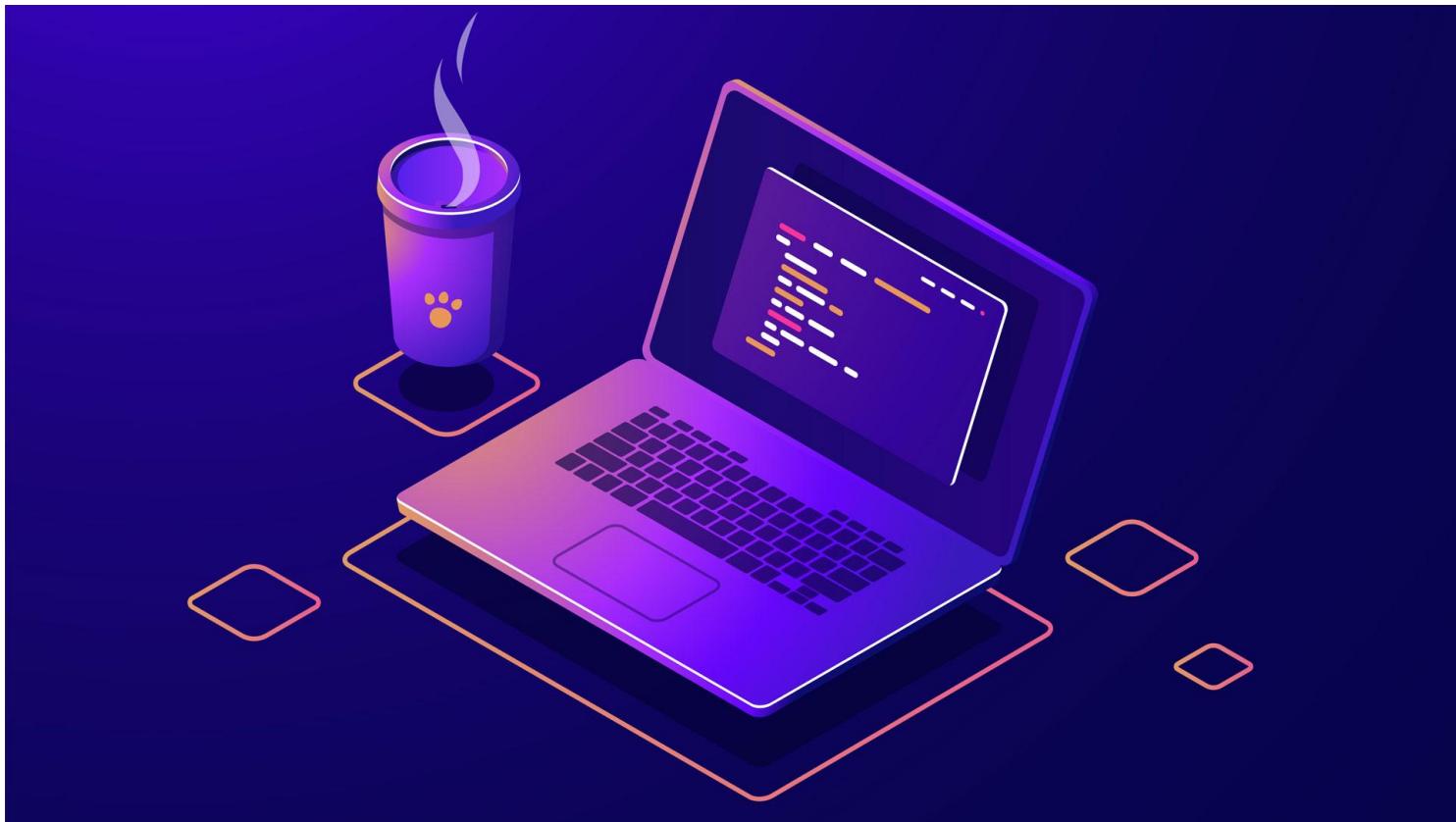
Twitter: lucasebazilio



OpenMP Problems



Problem 1



Problem 1



Given the following sequential code to count the number of times a value **key** appears in vector **a**

```
#define N 131072
long count_key(long Nlen, long *a, long key) {
    long count = 0;
    for (int i=0; i<Nlen; i++)
        if(a[i]==key) count++;
    return count;
}

int main() {
    long a[N], key = 42, nkey=0;
    for (long i=0;i<N;i++) a[i] = random()%N;
    a[N%43]=key; a[N%73]=key; a[N%3]=key;
    nkey = count_key(N, a, key);      // count key sequentially
    nkey = count_iter(N, a, key);    // count key in a using an iterative decomposition
    nkey = count_recur(N, a, key);   // count key in a with divide and conquer
}
```



Problem 1 - Point A

- (a) Write a parallel OpenMP version using an iterative task decomposition (`count_iter`), in which as many tasks as threads are generated.
- (b) Write a parallel OpenMP version using a recursive task decomposition (divide and conquer, `count_recur`). The implementation should take into account the overhead due to task creation, limiting their creation once a certain level in the recursive tree is reached.

Instructor Social Media

Youtube: Lucas Science



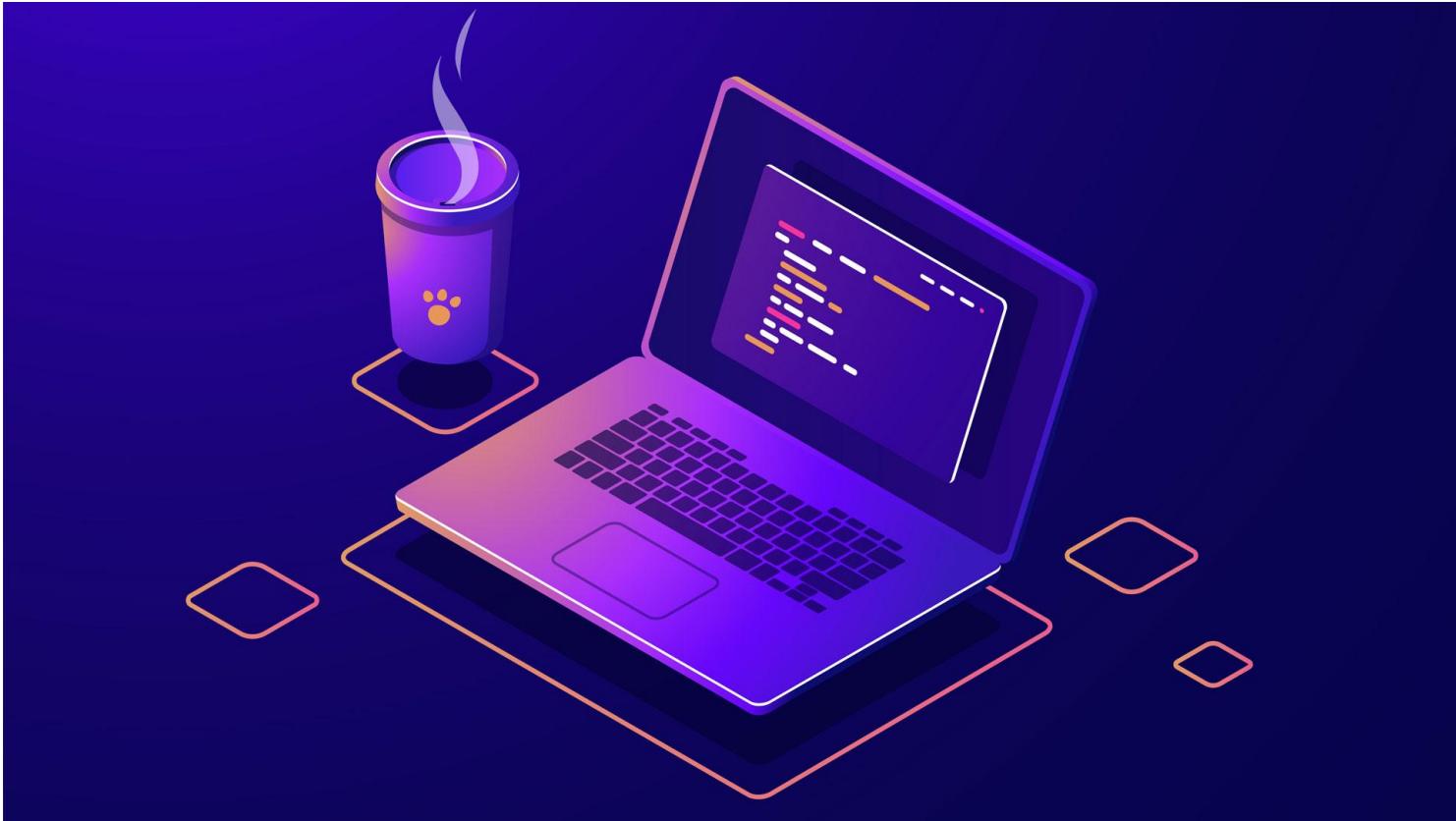
Instagram: lucaasbazilio



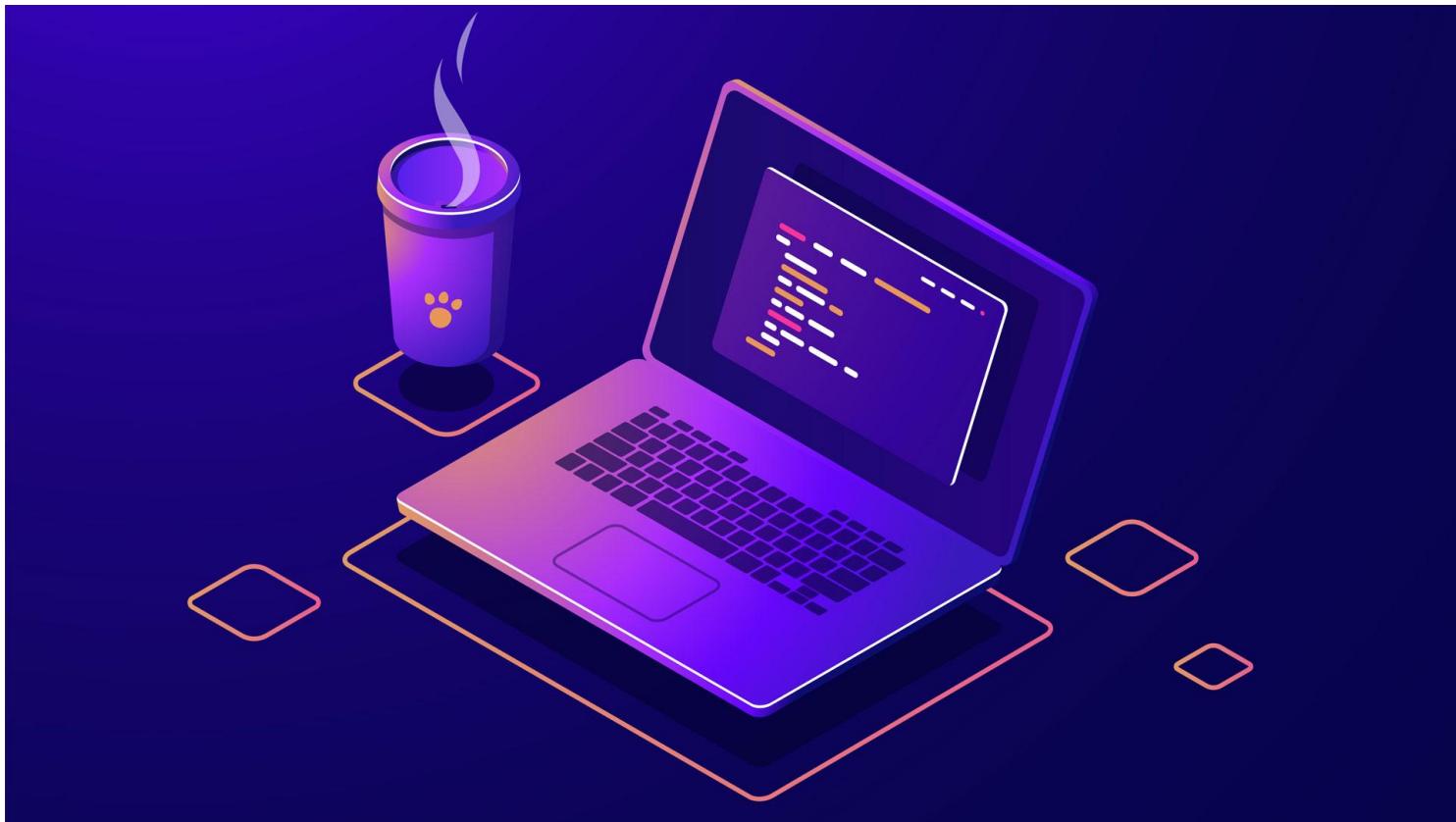
Twitter: lucasebazilio



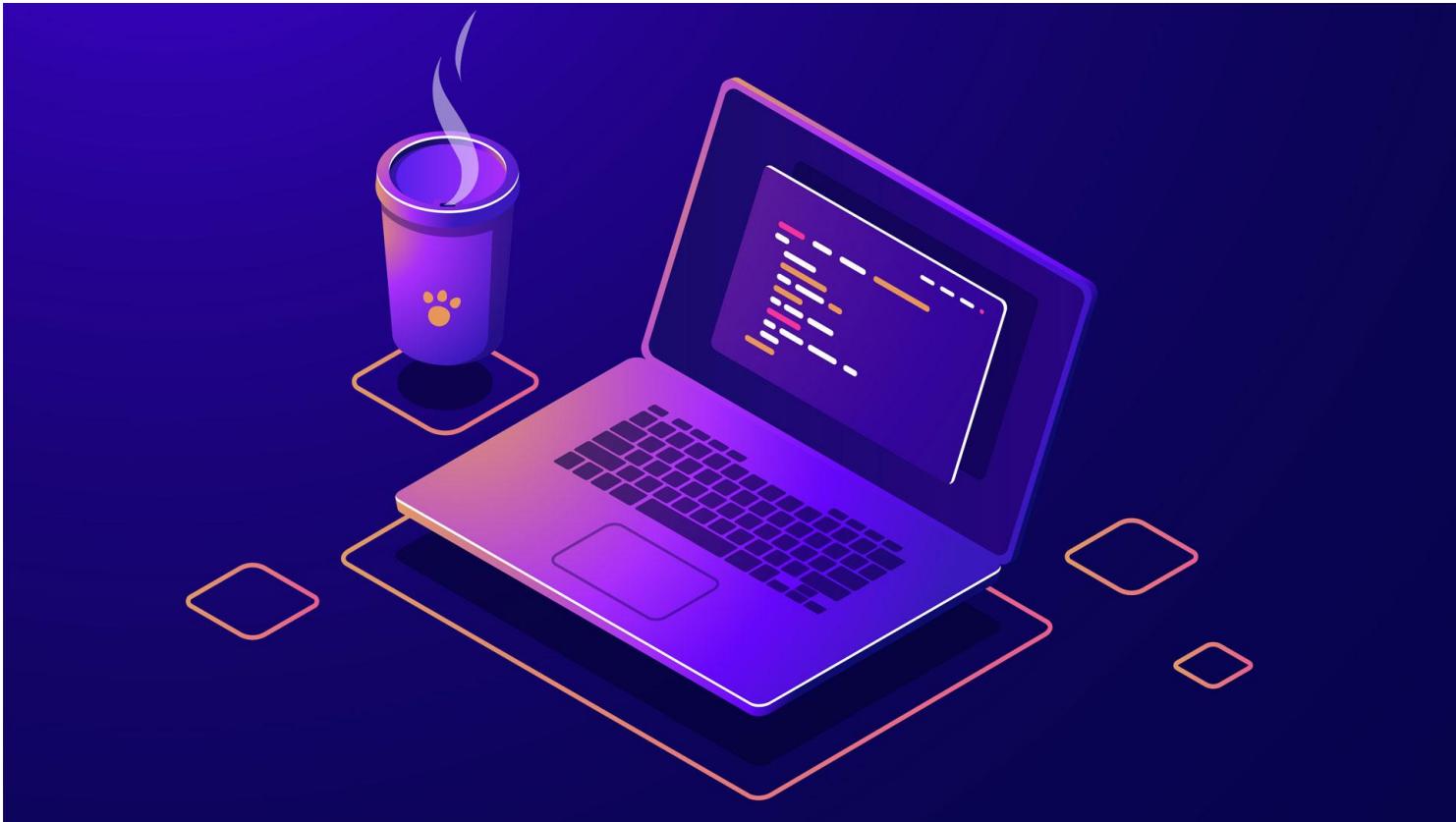
OpenMP Problems



Problem 1



Problem 1 - Point B



Problem 1



Given the following sequential code to count the number of times a value **key** appears in vector **a**

```
#define N 131072
long count_key(long Nlen, long *a, long key) {
    long count = 0;
    for (int i=0; i<Nlen; i++)
        if(a[i]==key) count++;
    return count;
}

int main() {
    long a[N], key = 42, nkey=0;
    for (long i=0;i<N;i++) a[i] = random()%N;
    a[N%43]=key; a[N%73]=key; a[N%3]=key;
    nkey = count_key(N, a, key);      // count key sequentially
    nkey = count_iter(N, a, key);    // count key in a using an iterative decomposition
    nkey = count_recur(N, a, key);   // count key in a with divide and conquer
}
```

Problem 1 - Point B



- (a) Write a parallel OpenMP version using an iterative task decomposition (`count_iter`), in which as many tasks as threads are generated.
- (b) Write a parallel OpenMP version using a recursive task decomposition (divide and conquer, `count_recur`). The implementation should take into account the overhead due to task creation, limiting their creation once a certain level in the recursive tree is reached.

Recursive Task Decomposition

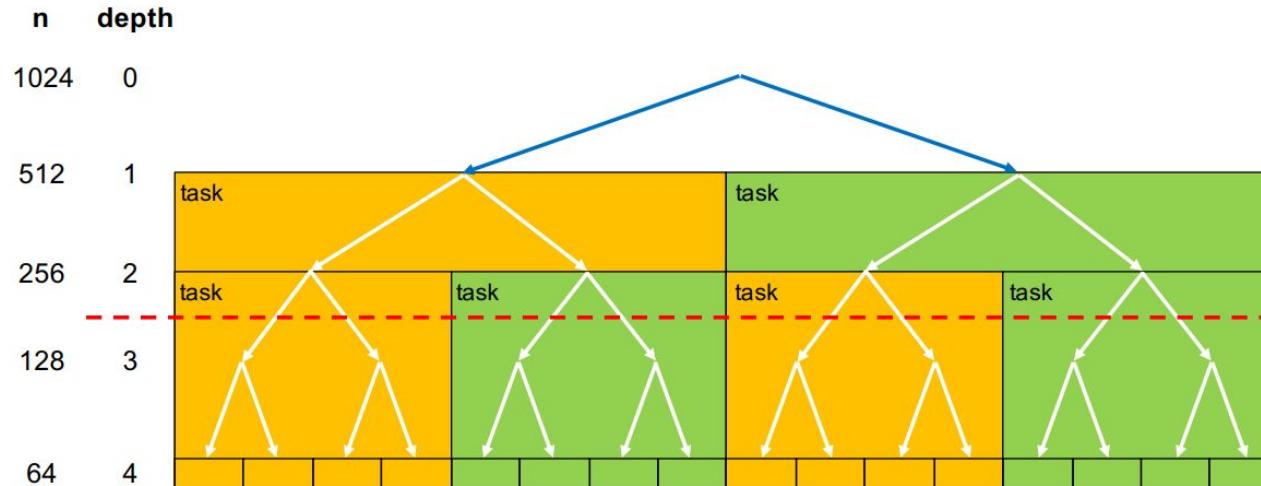


In recursive task decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (cut-off control).



Recursive Task Decomposition

Tree strategy with **depth recursion control**



Instructor Social Media

Youtube: Lucas Science



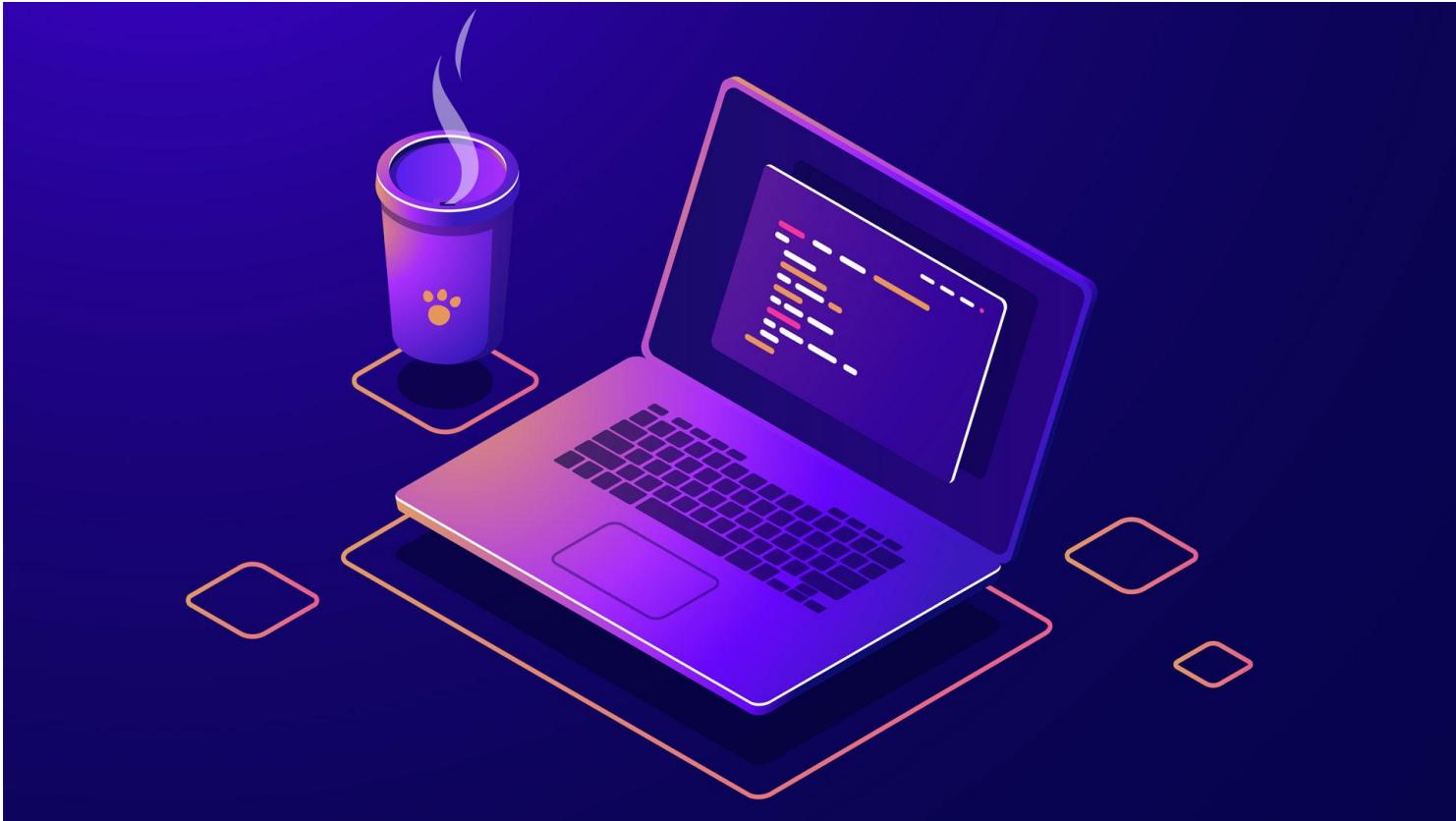
Instagram: lucaasbazilio



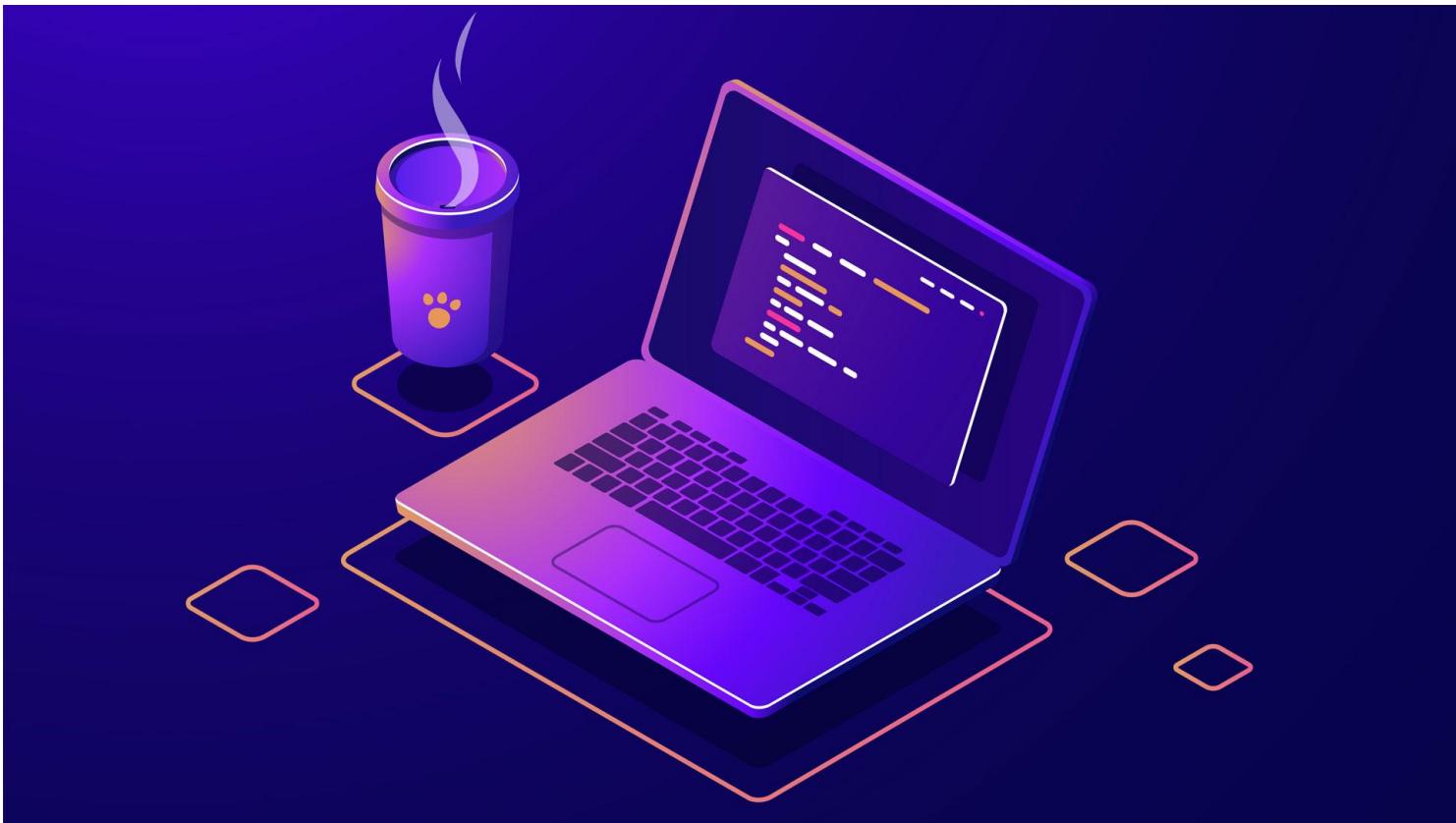
Twitter: lucasebazilio



OpenMP Problems



Problem 2



Problem 2



The following sequential code in C finds all positions in vector DBin in which a set of keys (contained in vector keys) appear. Positions where keys appear are stored in a new vector DBout (the order in DBout of the positions found is irrelevant).

```
int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```

Problem 2



```
#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```



Problem 2

- (a) Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop *i*, making use of the **taskloop** directive, in which you minimise the serialisation introduced by the synchronisation that you may introduce. **Note:** you are not allowed to change the structure of the two loops.
- (b) Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop *k*, again making use of the **taskloop** directive, in which you maximise the parallelism that can be exploited. **Notes:** 1) **taskloop** has an implicit **taskgroup** synchronisation that you can omit with the **nogroup** clause; 2) observe that the number of keys is not large when compared to the possible number of processors to use; and 3) you are not allowed to change the structure of the two loops.
- (c) Finally, write a third *OpenMP* parallelisation that implements a **task-based recursive divide-and-conquer decomposition strategy**, with the following requirements: 1) the recursion splits the input vector **DBin** in two almost identical halves, with a base case that corresponds to checking a single element of **DBin**; 2) uses a **cut-off strategy based on the size of the input vector**, so that tasks are only generated while that size is larger than **CUT_SIZE**; 3) only uses *OpenMP* pragmas and clauses for the implementation of the cut-off strategy; and 4) you have to use the synchronisation mechanism, if needed, that maximises the parallelism in the program.



Mutual Exclusion

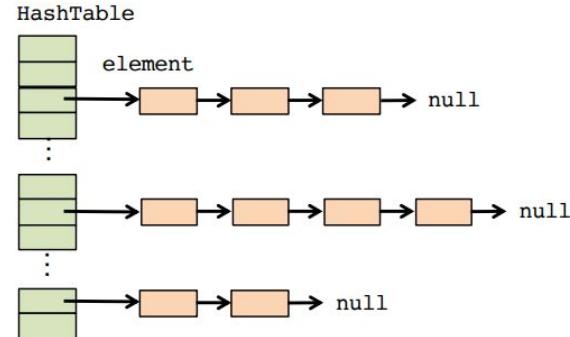
Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a region.



Mutual Exclusion

Associate a lock variable with each slot in the hash table,
protecting the chain of elements in an slot

```
omp_lock_t hash_lock[SIZE_HASH];  
  
#pragma omp parallel  
#pragma omp single  
{  
    for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);  
  
#pragma omp taskloop  
for (i = 0; i < SIZE_TABLE; i++) {  
    int index = hash_function (dataTable[i], SIZE_HASH);  
    omp_set_lock (&hash_lock[index]);  
    insert_element (dataTable[i], index, HashTable);  
    omp_unset_lock (&hash_lock[index]);  
}  
  
for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);  
}
```



Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Instructor Social Media

Youtube: Lucas Science



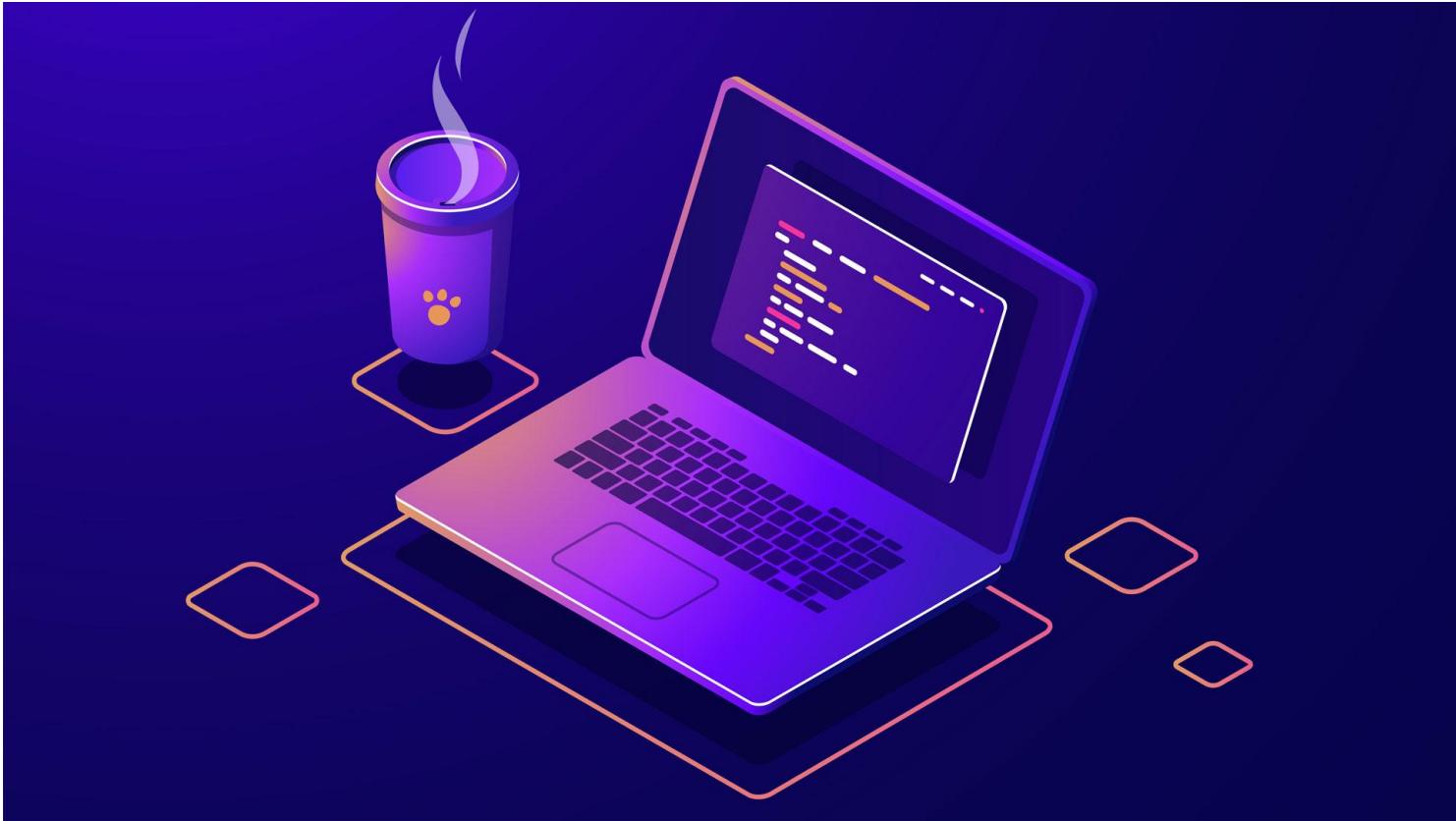
Instagram: lucaasbazilio



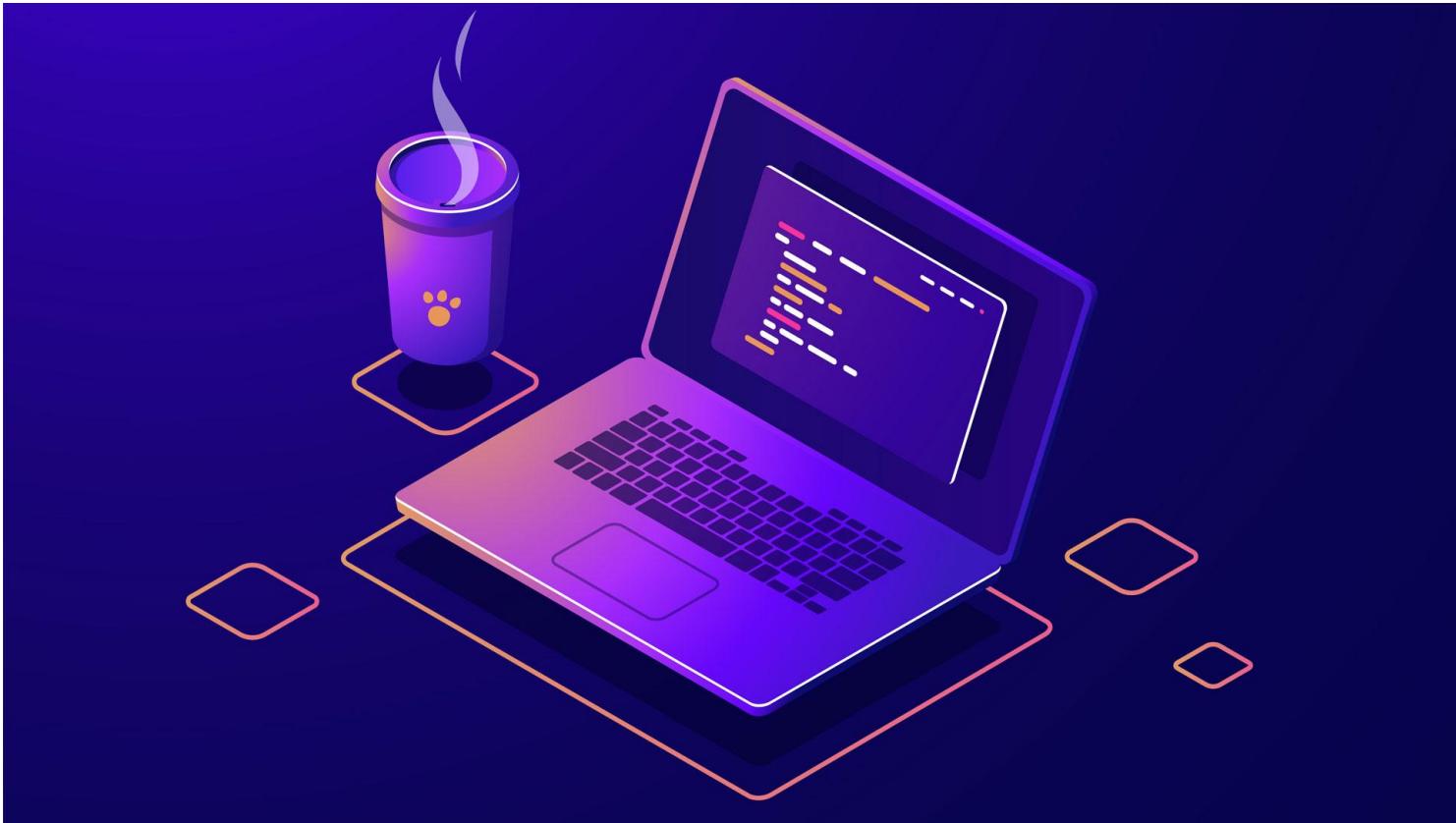
Twitter: lucasebazilio



OpenMP Problems



Problem 2 - Point B



Problem 2



The following sequential code in C finds all positions in vector DBin in which a set of keys (contained in vector keys) appear. Positions where keys appear are stored in a new vector DBout (the order in DBout of the positions found is irrelevant).

```
int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```

Problem 2



```
#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```



Problem 2

- (a) Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop *i*, making use of the **taskloop** directive, in which you minimise the serialisation introduced by the synchronisation that you may introduce. **Note:** you are not allowed to change the structure of the two loops.
- (b) Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop *k*, again making use of the **taskloop** directive, in which you maximise the parallelism that can be exploited. **Notes:** 1) **taskloop** has an implicit **taskgroup** synchronisation that you can omit with the **nogroup** clause; 2) observe that the number of keys is not large when compared to the possible number of processors to use; and 3) you are not allowed to change the structure of the two loops.
- (c) Finally, write a third *OpenMP* parallelisation that implements a **task-based recursive divide-and-conquer decomposition strategy**, with the following requirements: 1) the recursion splits the input vector **DBin** in two almost identical halves, with a base case that corresponds to checking a single element of **DBin**; 2) uses a **cut-off strategy based on the size of the input vector**, so that tasks are only generated while that size is larger than **CUT_SIZE**; 3) only uses *OpenMP* pragmas and clauses for the implementation of the cut-off strategy; and 4) you have to use the synchronisation mechanism, if needed, that maximises the parallelism in the program.



Mutual Exclusion

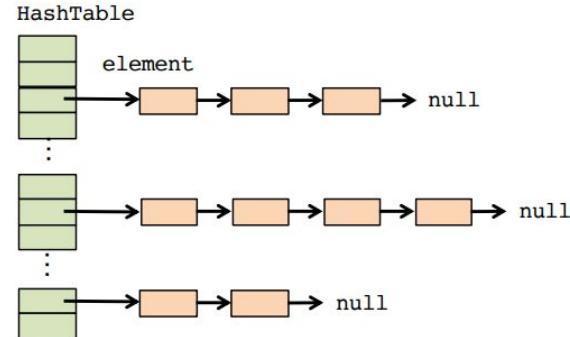
Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a region.



Mutual Exclusion

Associate a lock variable with each slot in the hash table,
protecting the chain of elements in an slot

```
omp_lock_t hash_lock[SIZE_HASH];  
  
#pragma omp parallel  
#pragma omp single  
{  
    for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);  
  
#pragma omp taskloop  
for (i = 0; i < SIZE_TABLE; i++) {  
    int index = hash_function (dataTable[i], SIZE_HASH);  
    omp_set_lock (&hash_lock[index]);  
    insert_element (dataTable[i], index, HashTable);  
    omp_unset_lock (&hash_lock[index]);  
}  
  
for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);  
}
```



Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Instructor Social Media

Youtube: Lucas Science



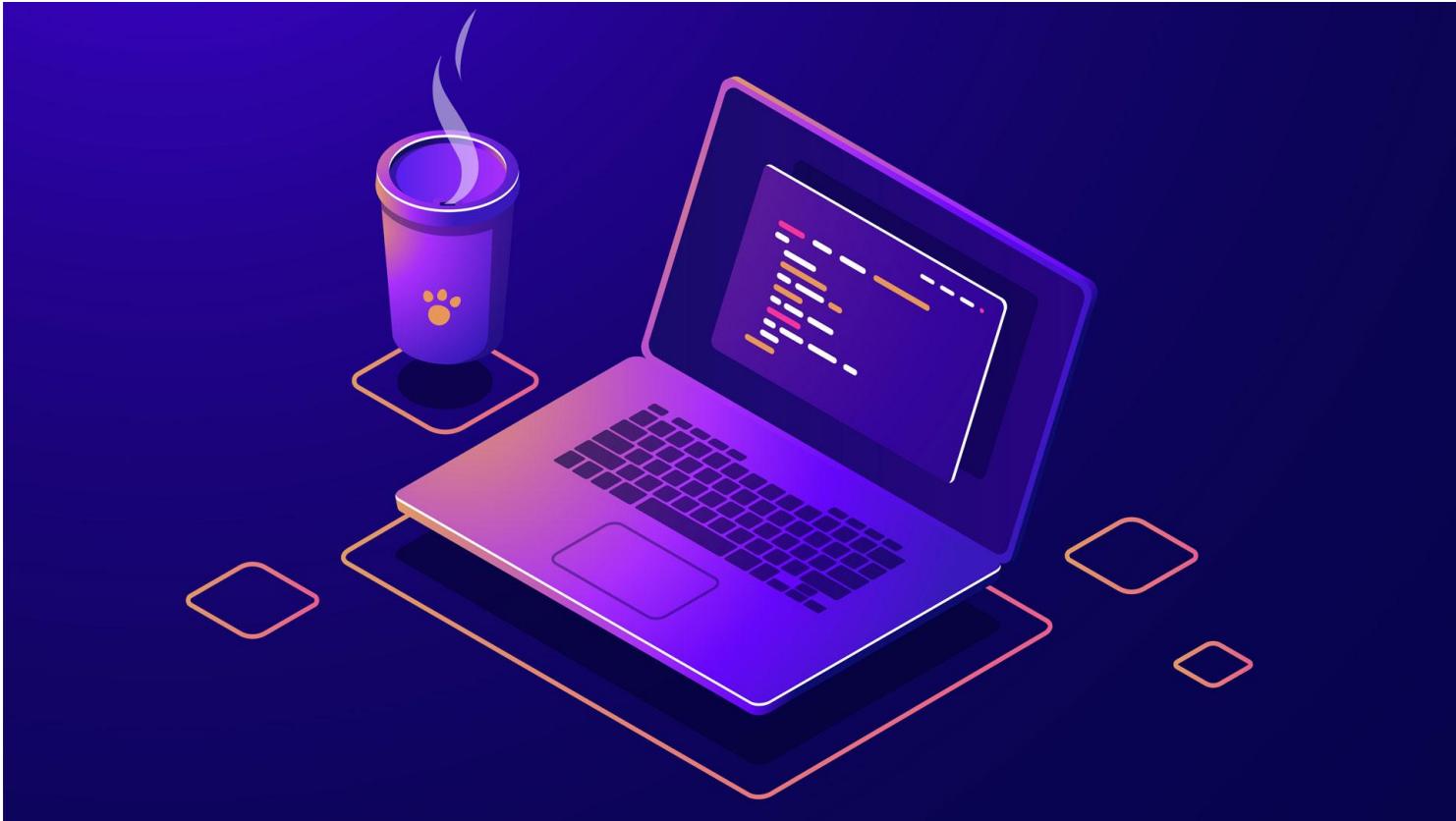
Instagram: lucaasbazilio



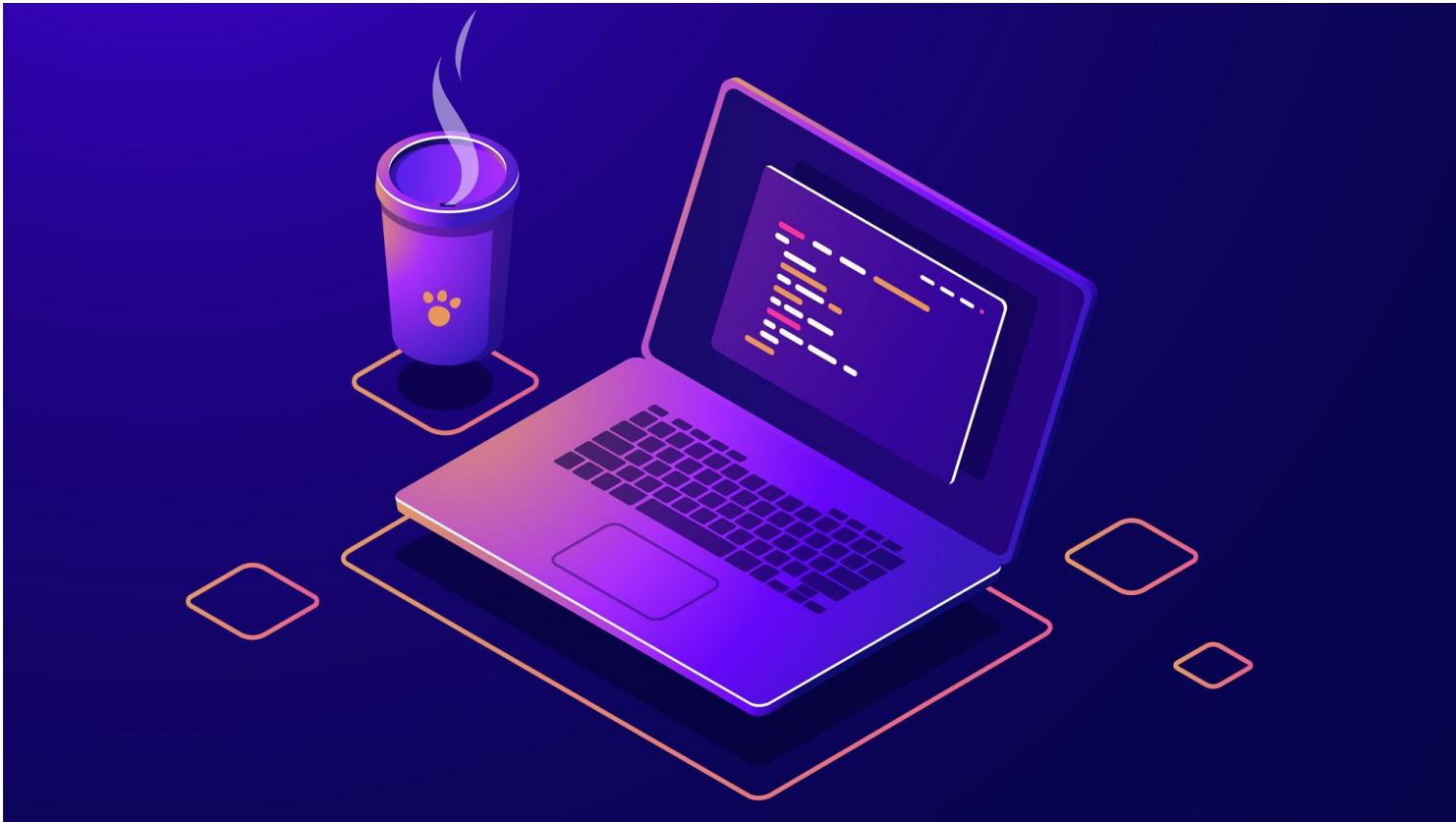
Twitter: lucasebazilio



OpenMP Problems



Problem 2 - Point C



Problem 2



The following sequential code in C finds all positions in vector DBin in which a set of keys (contained in vector keys) appear. Positions where keys appear are stored in a new vector DBout (the order in DBout of the positions found is irrelevant).

```
int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```

Problem 2



```
#define DBsize 1048576
#define nkeys 16 // the number of processors can be larger than the number of keys

int main() {
    double keys[nkeys], DBin[DBsize], DBout[nkeys][DBsize];
    unsigned int i, k, counter[nkeys];

    getkeys(keys, nkeys);           // get keys
    init_DBin(DBin, DBsize);       // initialize elements in DBin
    clear_DBout(DBout, nkeys, DBsize); // initialize elements in DBout
    clear_counter(counter, nkeys); // initialize counter to zero

    for (i = 0; i < DBsize; i++)
        for (k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) DBout[k][counter[k]++] = i;
}
```



Problem 2

- (a) Write a first *OpenMP* parallelisation that implements an **iterative task decomposition strategy** of the outermost loop *i*, making use of the **taskloop** directive, in which you minimise the serialisation introduced by the synchronisation that you may introduce. **Note:** you are not allowed to change the structure of the two loops.
- (b) Write a second *OpenMP* parallelisation that also implements an **iterative task decomposition strategy**, but this time applied to the innermost loop *k*, again making use of the **taskloop** directive, in which you maximise the parallelism that can be exploited. **Notes:** 1) **taskloop** has an implicit **taskgroup** synchronisation that you can omit with the **nogroup** clause; 2) observe that the number of keys is not large when compared to the possible number of processors to use; and 3) you are not allowed to change the structure of the two loops.
- (c) Finally, write a third *OpenMP* parallelisation that implements a **task-based recursive divide-and-conquer decomposition strategy**, with the following requirements: 1) the recursion splits the input vector **DBin** in two almost identical halves, with a base case that corresponds to checking a single element of **DBin**; 2) uses a **cut-off strategy based on the size of the input vector**, so that tasks are only generated while that size is larger than **CUT_SIZE**; 3) only uses *OpenMP* pragmas and clauses for the implementation of the cut-off strategy; and 4) you have to use the synchronisation mechanism, if needed, that maximises the parallelism in the program.



Mutual Exclusion

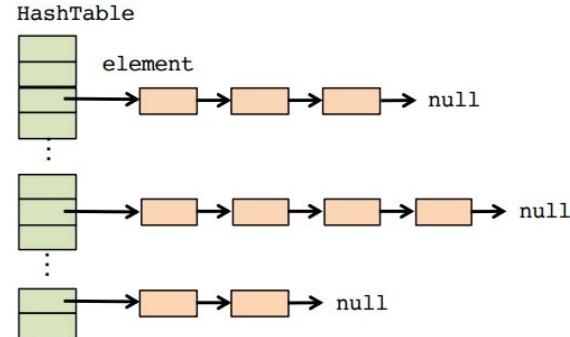
Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a region.



Mutual Exclusion

Associate a lock variable with each slot in the hash table,
protecting the chain of elements in an slot

```
omp_lock_t hash_lock[SIZE_HASH];  
  
#pragma omp parallel  
#pragma omp single  
{  
    for (i = 0; i < SIZE_HASH; i++) omp_init_lock(&hash_lock[i]);  
  
#pragma omp taskloop  
for (i = 0; i < SIZE_TABLE; i++) {  
    int index = hash_function (dataTable[i], SIZE_HASH);  
    omp_set_lock (&hash_lock[index]);  
    insert_element (dataTable[i], index, HashTable);  
    omp_unset_lock (&hash_lock[index]);  
}  
  
for (i = 0; i < SIZE_HASH; i++) omp_destroy_lock(&hash_lock[i]);  
}
```



Threads may be inserting elements into the hash table in parallel, as long as these elements hash to different slots

Recursive Task Decomposition

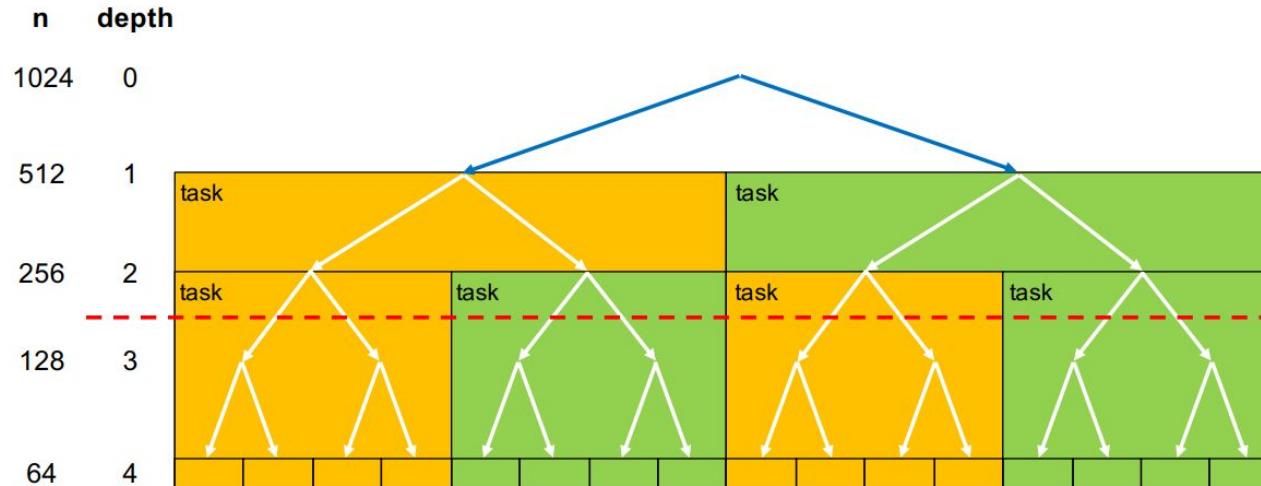


In recursive task decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (cut-off control).



Recursive Task Decomposition

Tree strategy with **depth recursion control**



Instructor Social Media

Youtube: Lucas Science



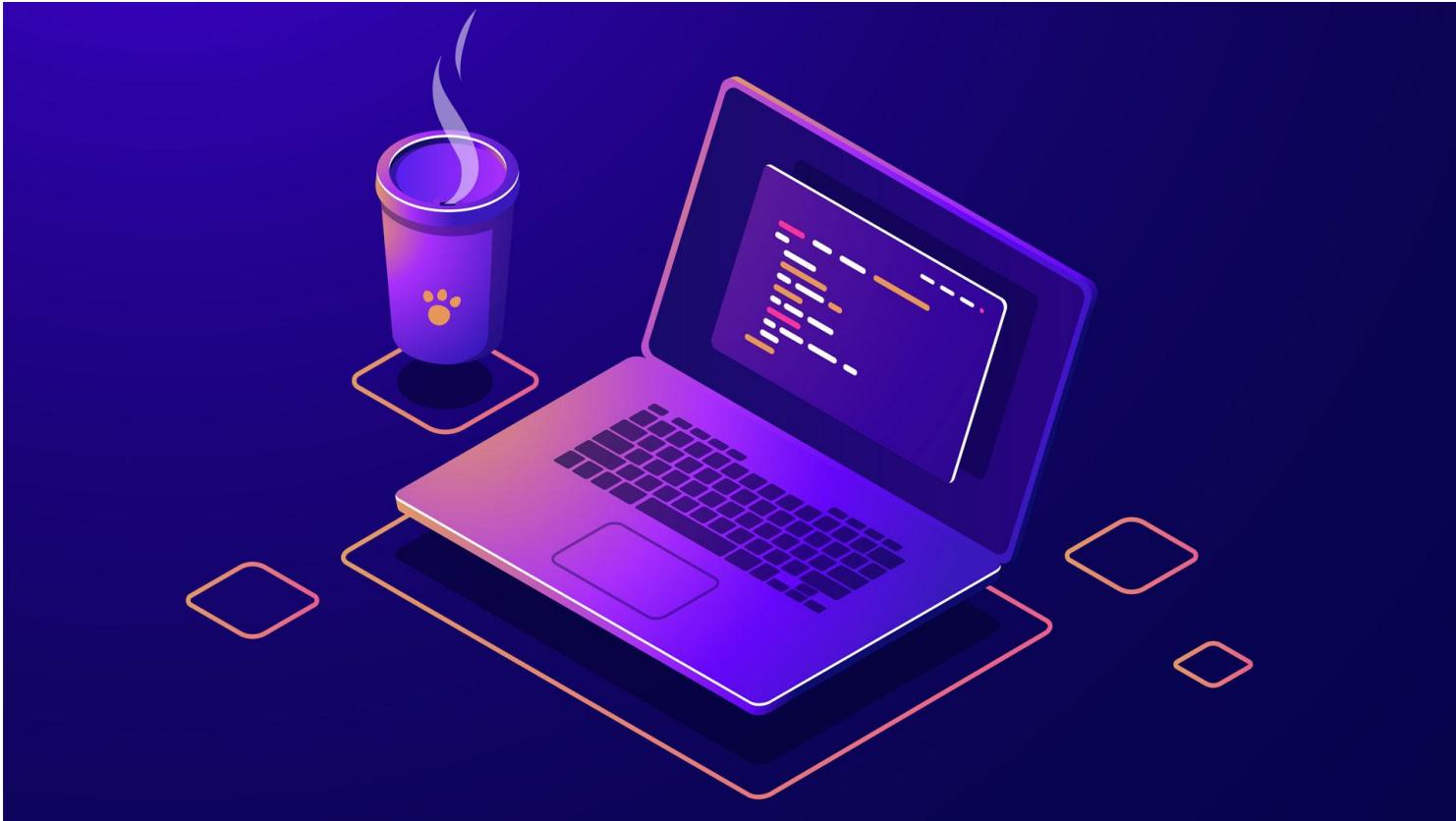
Instagram: lucaasbazilio



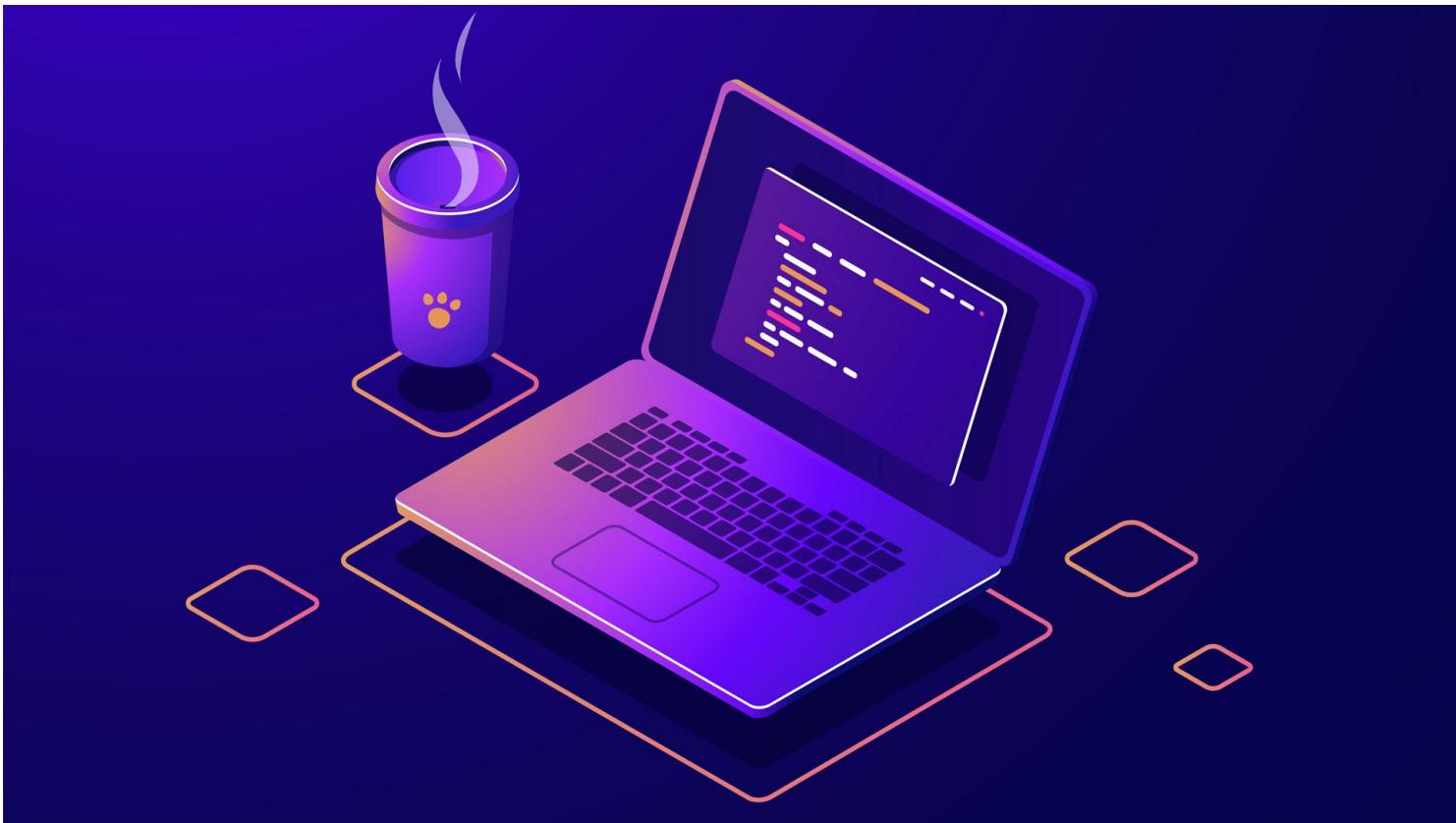
Twitter: lucasebazilio



OpenMP Problems



Problem 3





Problem 3

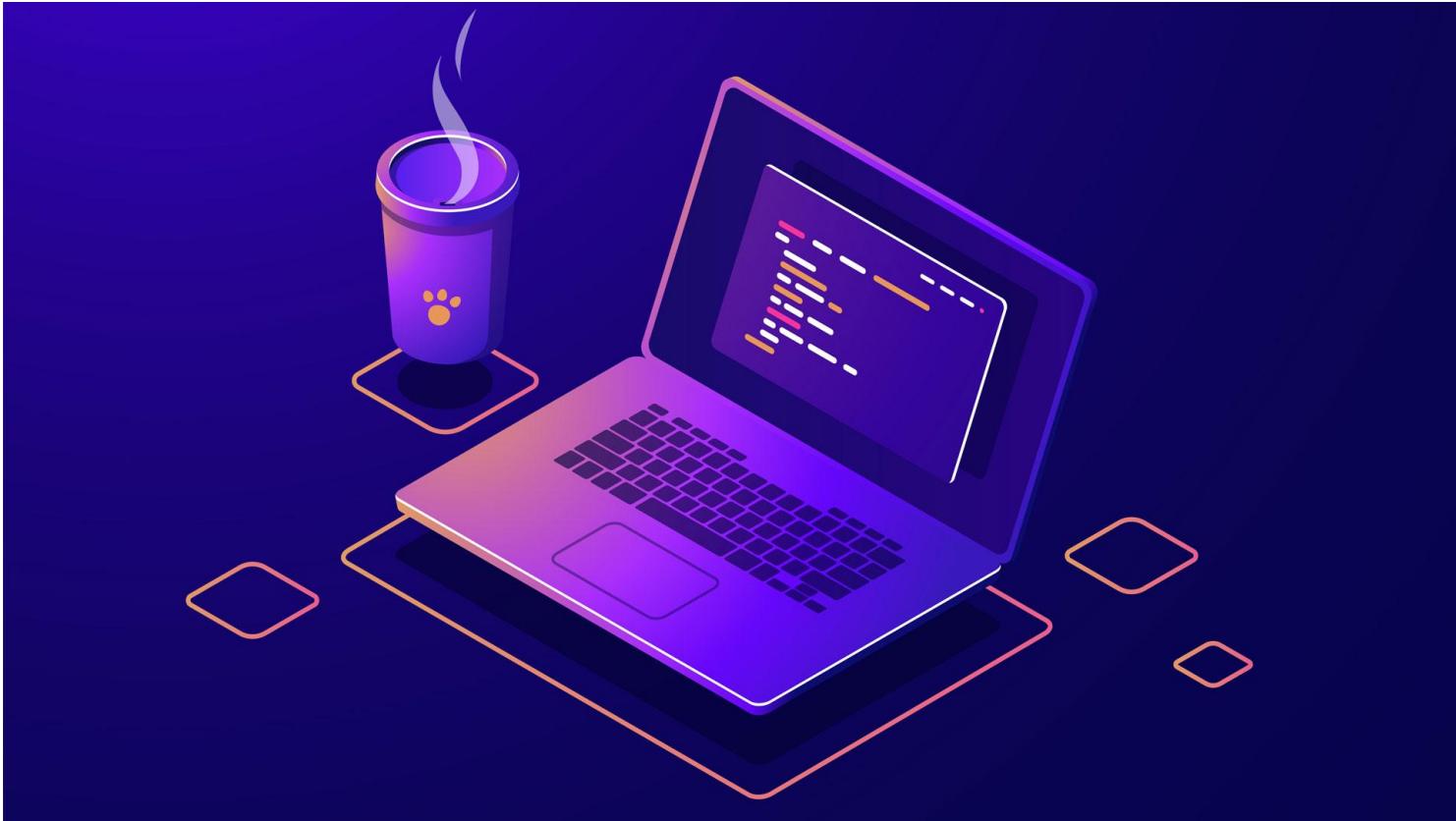
Parallelise the following sequential code using task dependencies in OpenMP, following a producer-consumer execution model. Producer and consumer code should be in two different tasks.

```
float sample[INPUT_SIZE+TAP1];
float coeff1[TAP1], coeff2[TAP2];
float data_out[INPUT_SIZE], final[INPUT_SIZE];

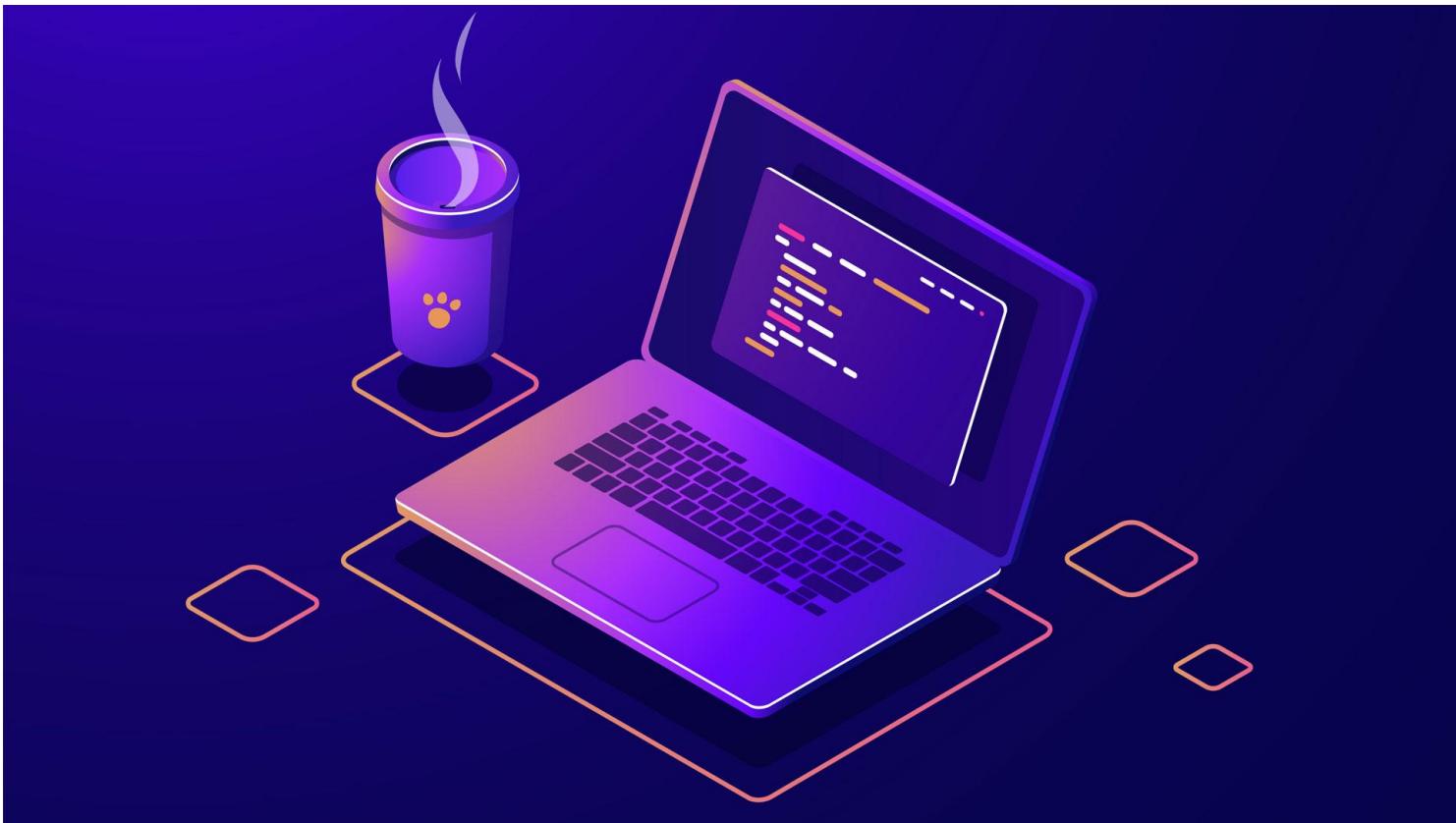
void main() {
    float sum;
    for (int i=0; i<INPUT_SIZE; i++) {
        // Producer: Finite Impulse Response (FIR) filter
        sum=0.0;
        for (int j=0; j<TAP1; j++)
            sum += sample[i+j] * coeff1[j];
        data_out[i] = sum;

        // Consumer: apply correction function
        for (int j=0; j<TAP2; j++)
            final[i] += correction(data_out[i], coeff2[j]);
    }
}
```

OpenMP Problems



Problem 4





Problem 4

SAXPY (*Single-precision A times X Plus Y*) is a combination of scalar multiplication and vector addition. It takes as input two vectors of 32-bit floats x and y with n elements each, and a scalar value a . It multiplies each element $x[i]$ by a and adds the result to $y[i]$, as shown below:

```
void saxpy(int n, float a, float *x, float *y) {  
    for (int i = 0; i < n; ++i) y[i] = a * x[i] + y[i];  
}
```



Problem 4

Given the following main program that makes use of saxpy function already defined. We want to achieve the asynchronous execution of the initialization loop, the two SAXPY calls and the two loops writing the result vectors to files.

Write a parallel version making use of the OpenMP task construct and task dependencies, creating the appropriate parallel context.



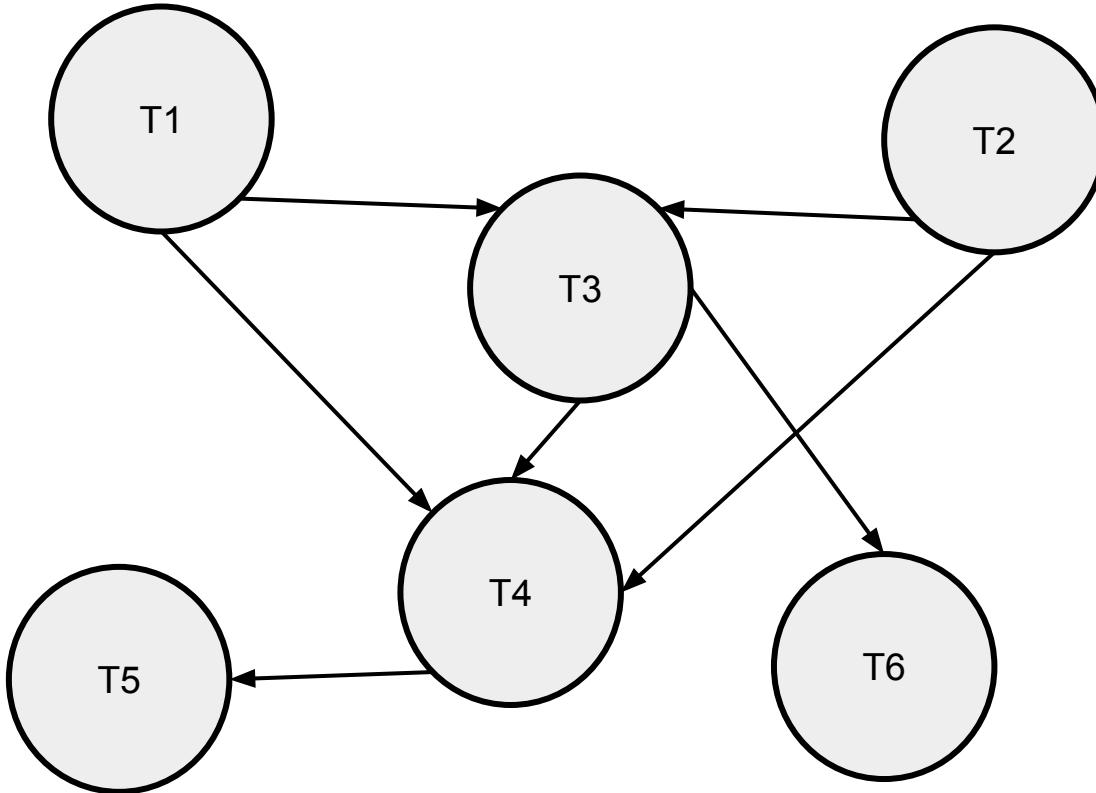
Problem 4

```
int main() {
    int N = 1 << 20;      /* 1 million floats */
    float *fx = (float *) malloc(N * sizeof(float));
    float *fy = (float *) malloc(N * sizeof(float));
    FILE *fpx = fopen("fx.out", "w");
    FILE *fpy = fopen("fy.out", "w");
    /* simple initialization just for testing */
    for (int k = 0; k < N; ++k)
        fx[k] = 2.0f + (float) k;
    for (int k = 0; k < N; ++k)
        fy[k] = 1.0f + (float) k;
    /* Run SAXPY TWICE */
    saxpy(N, 3.0f, fx, fy);
    saxpy(N, 5.0f, fy, fx);

    /* Save results */
    for (int k = 0; k < N; ++k)
        fprintf(fpx, " %f ", fx[k]);
    for (int k = 0; k < N; ++k)
        fprintf(fpy, " %f ", fy[k]);

    free(fx); fclose(fpx);
    free(fy); fclose(fpy);
}
```

Problem 4



Instructor Social Media

Youtube: Lucas Science



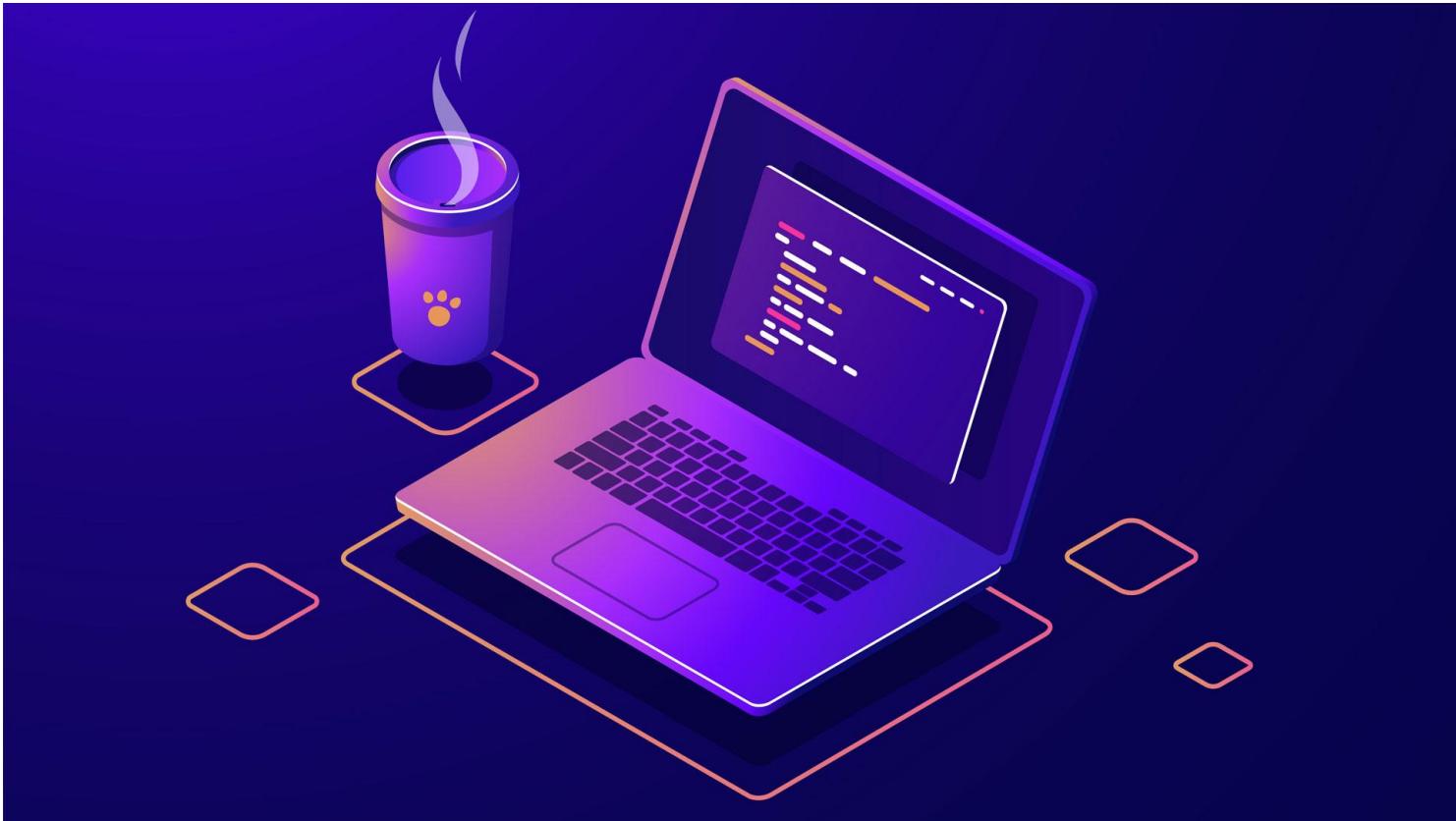
Instagram: lucaasbazilio



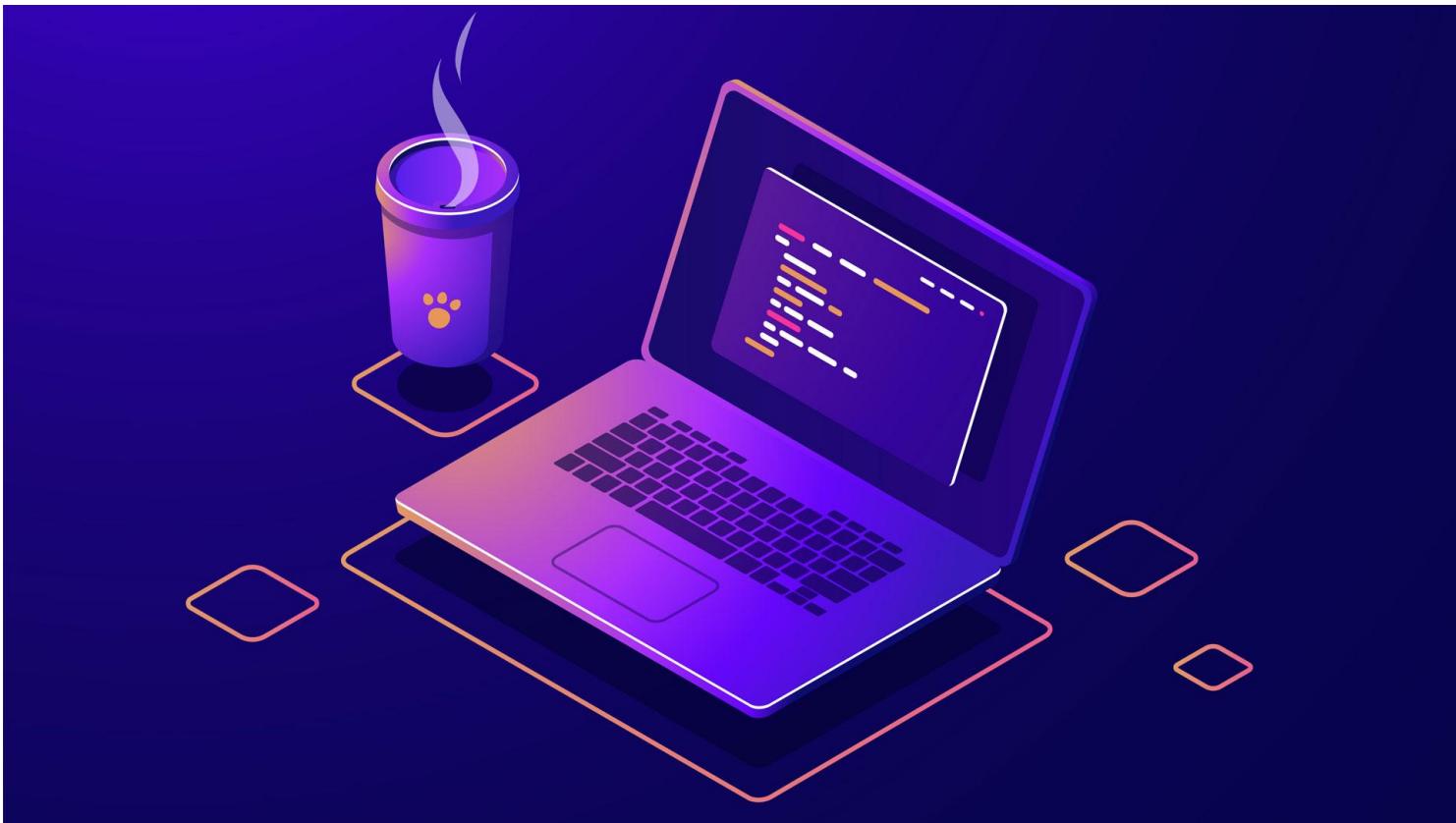
Twitter: lucasebazilio



OpenMP Problems



Problem 5



Problem 5



We ask to parallelize the computation of the histogram of values appearing on a vector. The histogram is another vector in which each position counts the number of elements in the input vector that are in a certain value range. The following program shows a possible sequential implementation for computing the histogram (vector **frequency**) of the input vector **numbers**:

```
#define MAX_ELEM 1024*1024
#define HIST_SIZE 250
unsigned int numbers[MAX_ELEM];
unsigned int frequency[HIST_SIZE];

void ReadNumbers (int * input, int * size);
void FindBounds(int * input, int size, int * min, int * max) {
    for (int i=0; i<size; i++)
        if (input[i]>(*max)) (*max)=input[i];

    for (int i=0; i<size; i++)
        if (input[i]<(*min)) (*min)=input[i];
}

void FindFrequency(int * input, int size , int * histogram, int min, int max) {
    int tmp;
    for (int i=0; i<size; i++) {
        tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));
        histogram[tmp]++;
    }
}
```



Problem 5

```
void DrawHistogram(int * histogram, int minimum, int maximum);
void main() {
    int num_elem, max, min;

    ReadNumbers(numbers, &num_elem); // read input numbers
    max=min=numbers[0];
    FindBounds(numbers, num_elem, &min, &max); // returns the upper and lower
                                                // values for the histogram
    FindFrequency(numbers, num_elem, frequency, min, max); // compute histogram
    DrawHistogram(frequency, min, max); // print the histogram
}
```



Problem 5

- a) Write a parallel version with OpenMP for function `FindBounds` using an iterative task decomposition, in which you only generate as many tasks as threads in the parallel region, as you minimize the possible synchronization overheads.

- b) Write a parallel implementation for the function `FindFrequency` using OpenMP.



Problem 5 - Point A - Solution

```
void FindBounds(int * input, int size, int * min, int * max) {
    int chunk_size = (size + omp_get_max_threads() - 1) / omp_get_max_threads();
    int local_min = input[0], local_max = input[0];

#pragma omp parallel num_threads(omp_get_max_threads())
{
    int tid = omp_get_thread_num();
    int start = tid * chunk_size;
    int end = (tid + 1) * chunk_size;
    if (end > size) end = size;

    for (int i = start; i < end; i++) {
        if (input[i] > local_max) local_max = input[i];
        if (input[i] < local_min) local_min = input[i];
    }

#pragma omp critical
{
        if (local_max > *max) *max = local_max;
        if (local_min < *min) *min = local_min;
    }
}
}
```

Problem 5 - Point A - Solution



In this parallel version of `FindBounds`, we first determine the chunk size based on the input size and the number of threads in the parallel region. Then, we create a parallel region using `#pragma omp parallel` with the number of threads specified by `omp_get_max_threads()`. Inside the parallel region, each thread computes the minimum and maximum values of a subset of the input vector, as determined by its thread ID (`omp_get_thread_num()`) and the chunk size. To avoid race conditions when updating the global minimum and maximum values, we use a critical section (`#pragma omp critical`) to ensure mutual exclusion.



Mutual Exclusion

Mutual exclusion: mechanism to ensure that only one task at a time executes the code within a region.

Instructor Social Media

Youtube: Lucas Science



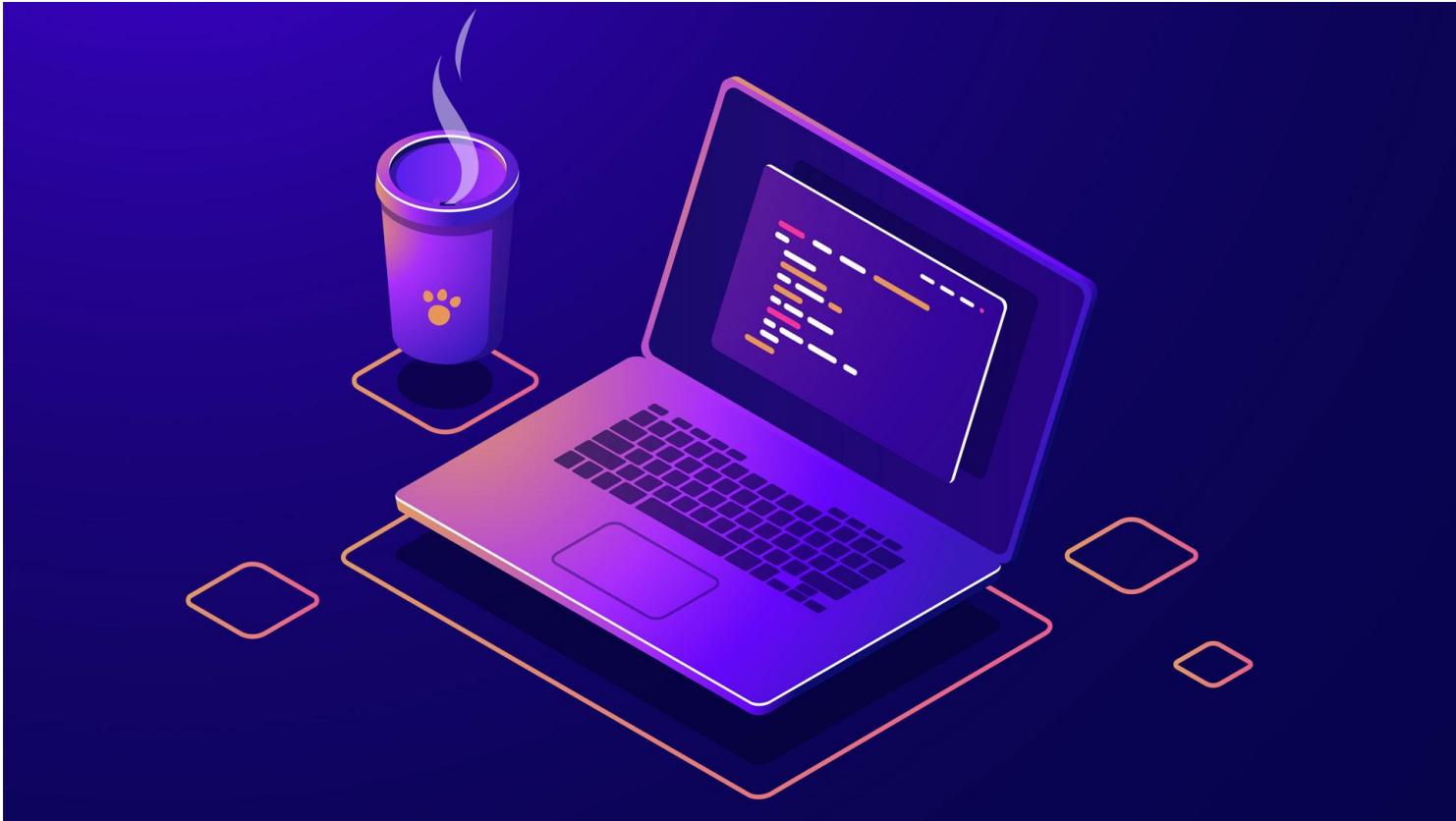
Instagram: lucaasbazilio



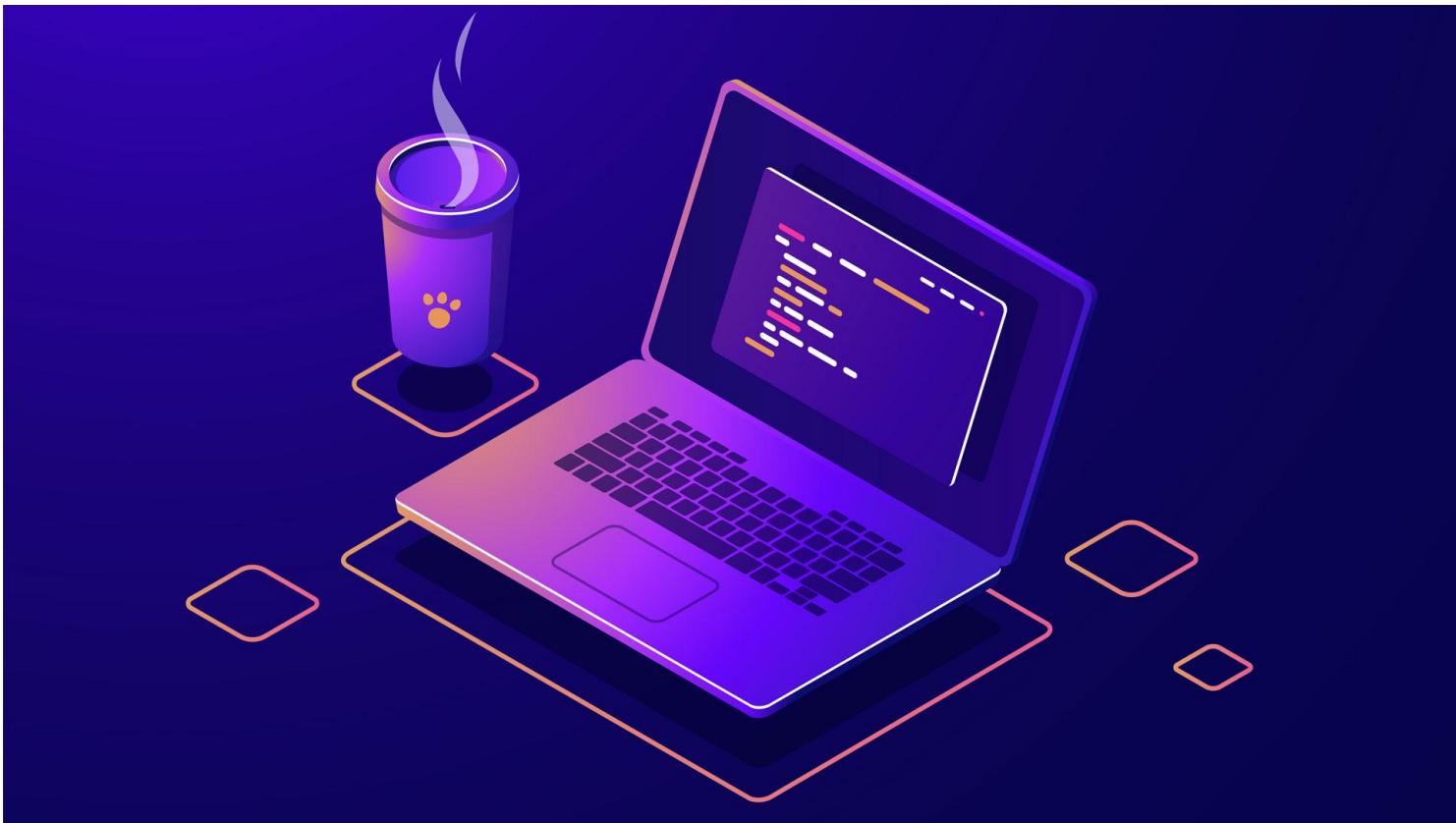
Twitter: lucasebazilio



OpenMP Problems



Problem 5



Problem 5



We ask to parallelize the computation of the histogram of values appearing on a vector. The histogram is another vector in which each position counts the number of elements in the input vector that are in a certain value range. The following program shows a possible sequential implementation for computing the histogram (vector **frequency**) of the input vector **numbers**:

```
#define MAX_ELEM 1024*1024
#define HIST_SIZE 250
unsigned int numbers[MAX_ELEM];
unsigned int frequency[HIST_SIZE];

void ReadNumbers (int * input, int * size);
void FindBounds(int * input, int size, int * min, int * max) {
    for (int i=0; i<size; i++)
        if (input[i]>(*max)) (*max)=input[i];

    for (int i=0; i<size; i++)
        if (input[i]<(*min)) (*min)=input[i];
}

void FindFrequency(int * input, int size , int * histogram, int min, int max) {
    int tmp;
    for (int i=0; i<size; i++) {
        tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));
        histogram[tmp]++;
    }
}
```



Problem 5

```
void DrawHistogram(int * histogram, int minimum, int maximum);
void main() {
    int num_elem, max, min;

    ReadNumbers(numbers, &num_elem); // read input numbers
    max=min=numbers[0];
    FindBounds(numbers, num_elem, &min, &max); // returns the upper and lower
                                                // values for the histogram
    FindFrequency(numbers, num_elem, frequency, min, max); // compute histogram
    DrawHistogram(frequency, min, max); // print the histogram
}
```



Problem 5

- a) Write a parallel version with OpenMP for function `FindBounds` using an iterative task decomposition, in which you only generate as many tasks as threads in the parallel region, as you minimize the possible synchronization overheads.

- b) Write a parallel implementation for the function `FindFrequency` using OpenMP.

Problem 5 - Point B - Solution



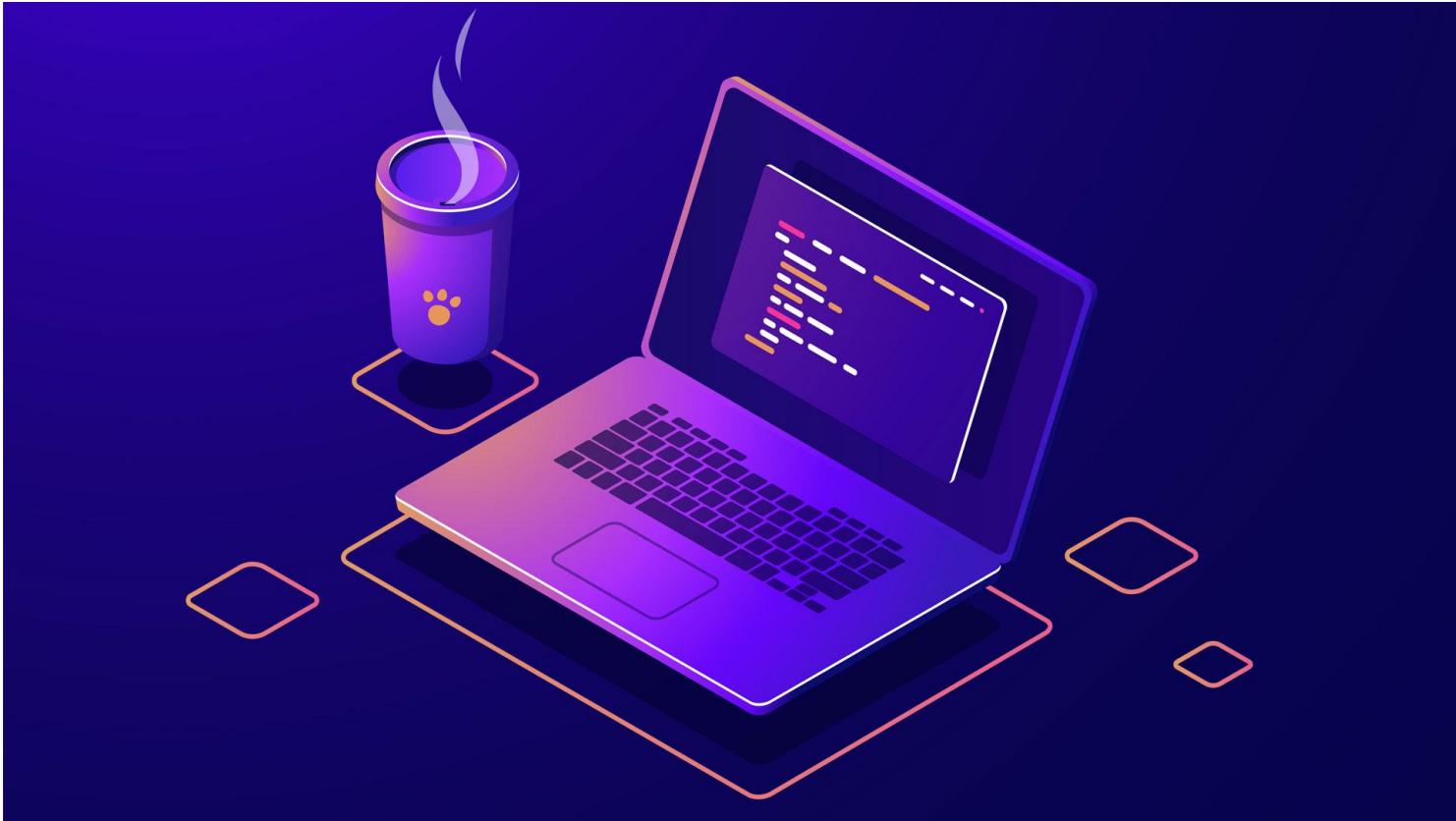
```
void FindFrequency(int * input, int size , int * histogram, int min, int max) {  
    int chunk_size = (size + omp_get_max_threads() - 1) / omp_get_max_threads();  
  
    #pragma omp parallel num_threads(omp_get_max_threads())  
    {  
        int tid = omp_get_thread_num();  
        int start = tid * chunk_size;  
        int end = (tid + 1) * chunk_size;  
        if (end > size) end = size;  
  
        int tmp;  
        for (int i = start; i < end; i++) {  
            tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));  
            #pragma omp atomic  
            histogram[tmp]++;
        }
    }
}
```

Problem 5 - Point B - Solution

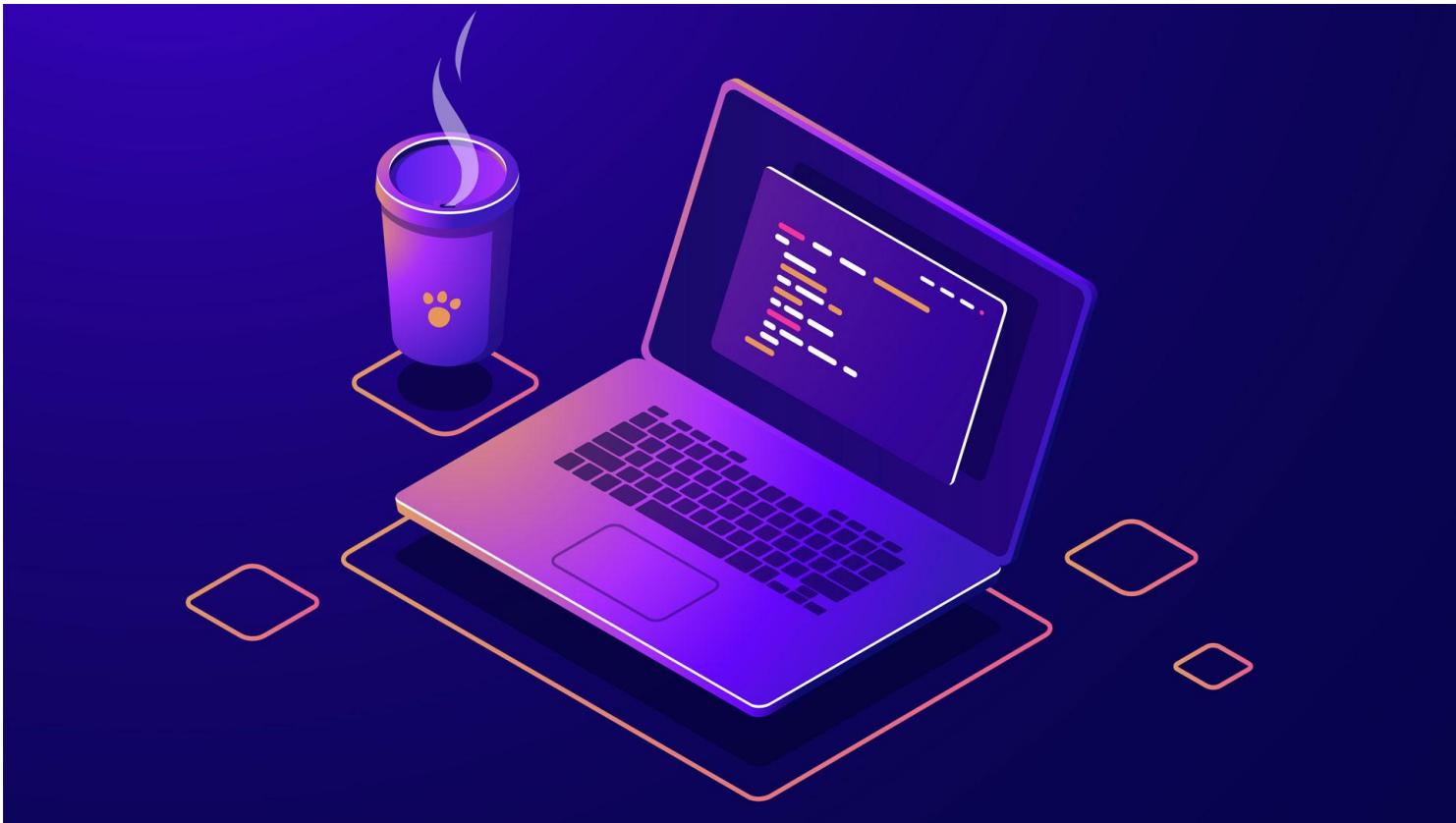


In this parallel version of FindFrequency, we first determine the chunk size based on the input size and the number of threads in the parallel region. Then, we create a parallel region using `#pragma omp parallel` with the number of threads specified by `omp_get_max_threads()`. Inside the parallel region, each thread computes the histogram of a subset of the input vector, as determined by its thread ID (`omp_get_thread_num()`) and the chunk size. To avoid race conditions when updating the histogram, we use an atomic operation (`#pragma omp atomic`) to increment the appropriate histogram bin.

OpenMP Problems



Problem 6



Problem 6



Given the following sequential code to calculate the sum of all the elements of a vector:

```
int v[N];
int sum_vector(int *X, int n) {
    int sum = 0;
    for (int i=0; i< n; i++) sum += X[i];
    return sum;
}
void main() {
    int sum = sum_vector(v, N);
}
```



Problem 6

- a) Write a parallel version in OpenMP implementing an iterative task decomposition parallelisation strategy making use of the OpenMP tasking model, with taskloop.

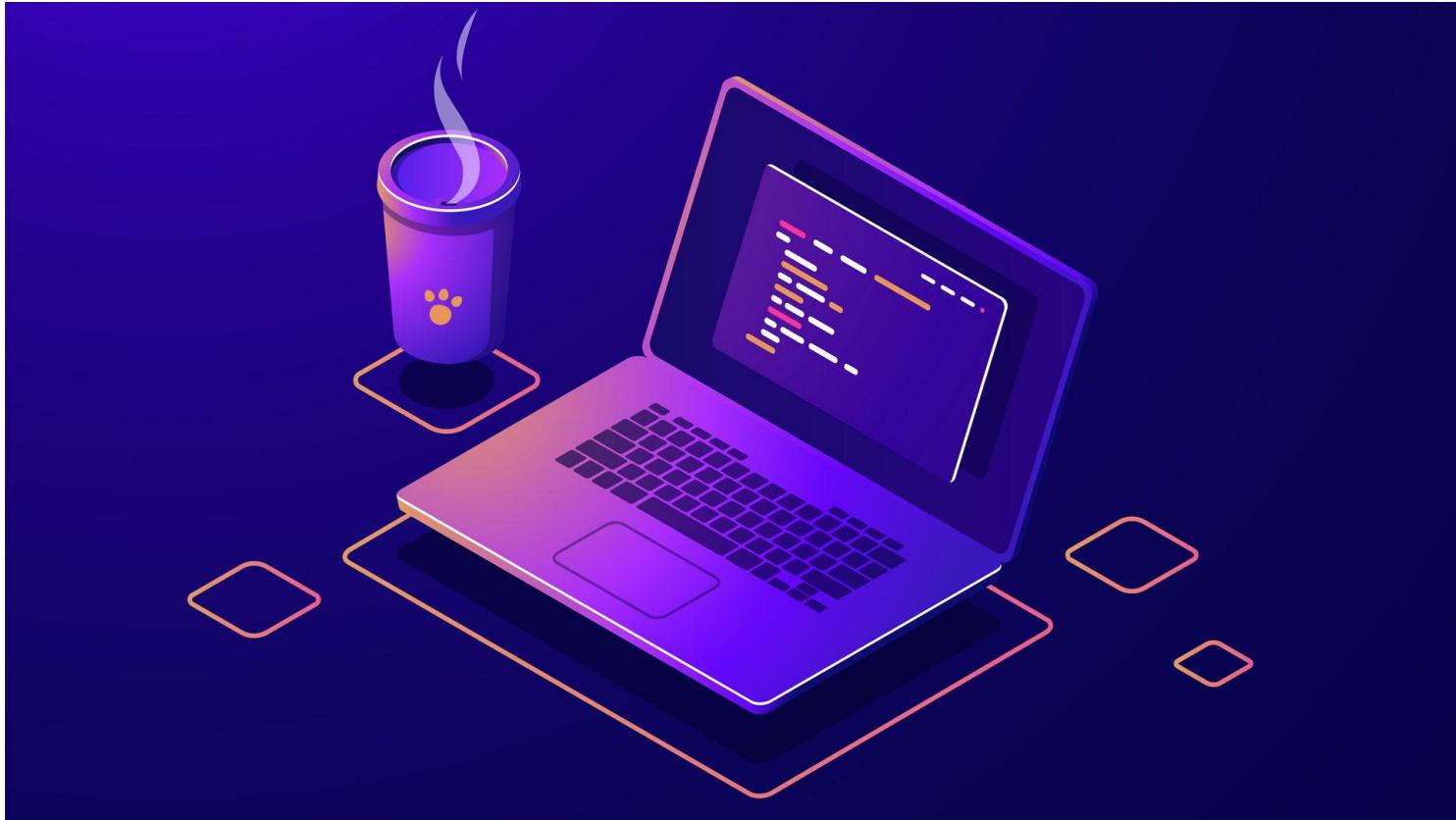
- b) Write a new parallel version in OpenMP of the previous sum_vector function (now called recursive_sum_vector) that implements a divide and conquer recursive strategy over vector v.

Problem 6 - Point A - Solution

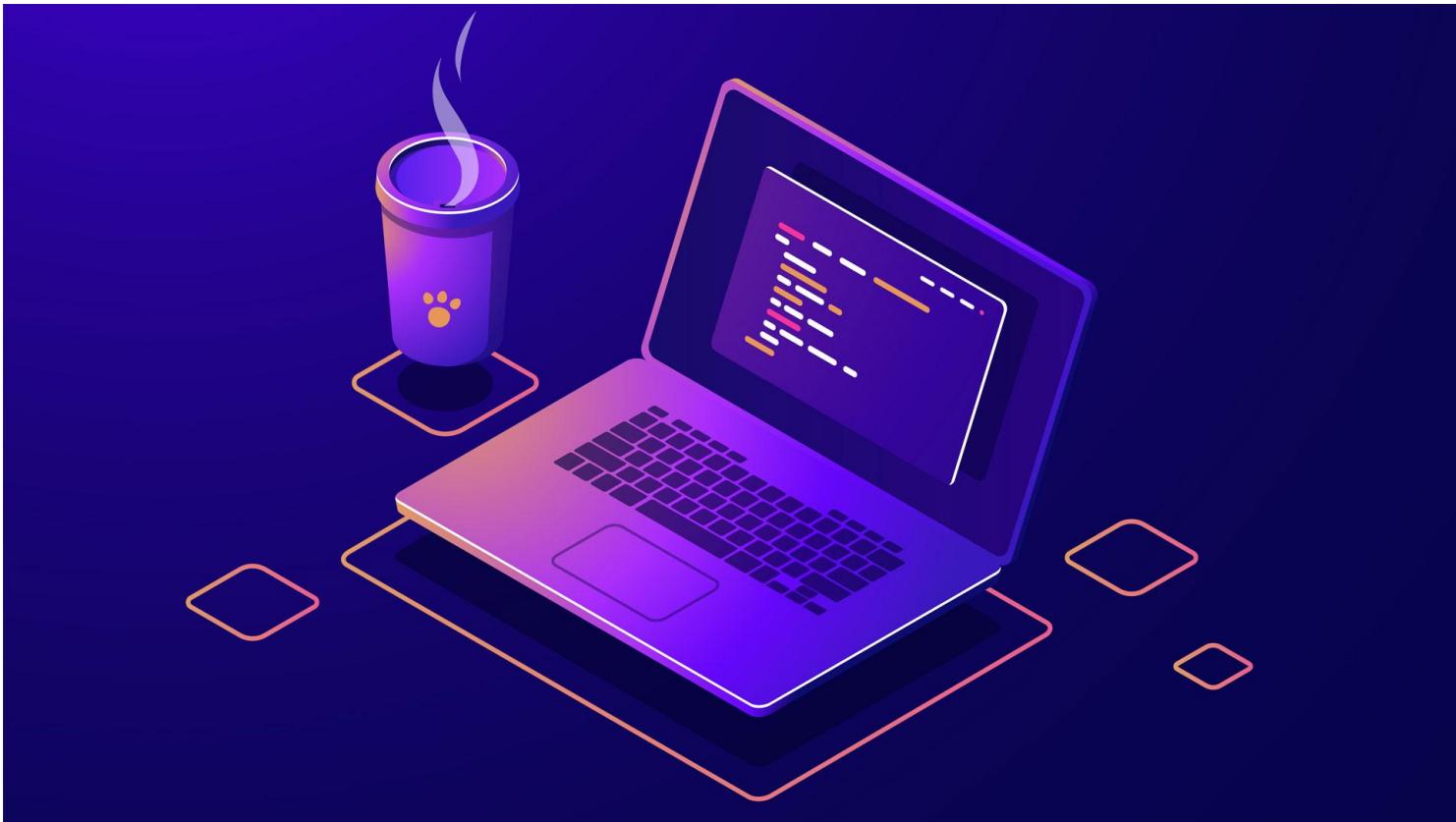


```
int sum_vector(int *x, int n) {
    int sum = 0;
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp taskloop grainsize(1)
            for (int i = 0; i < n; i++) {
                #pragma omp task shared(sum)
                {
                    #pragma omp atomic
                    sum += x[i];
                }
            }
        }
    }
    return sum;
}
```

OpenMP Problems



Problem 6



Problem 6



Given the following sequential code to calculate the sum of all the elements of a vector:

```
int v[N];
int sum_vector(int *X, int n) {
    int sum = 0;
    for (int i=0; i< n; i++) sum += X[i];
    return sum;
}
void main() {
    int sum = sum_vector(v, N);
}
```



Problem 6

- a) Write a parallel version in OpenMP implementing an iterative task decomposition parallelisation strategy making use of the OpenMP tasking model, with taskloop.

- b) Write a new parallel version in OpenMP of the previous sum_vector function (now called recursive_sum_vector) that implements a divide and conquer recursive strategy over vector v.

Problem 6 - Point B - Solution



```
int recursive_sum_vector(int *X, int n) {
    int sum = 0;
    if (n == 1) { return X[0]; } // Base Case
    else {
        #pragma omp parallel
        {
            #pragma omp single
            {
                #pragma omp task shared(sum)
                {
                    int half = n / 2;
                    sum = recursive_sum_vector(X, half) +
                        recursive_sum_vector(X + half, n-half);
                }
            }
        }
    }
    return sum;
}
```

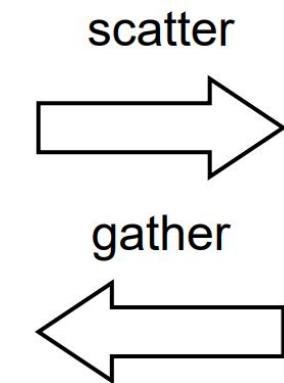
Gather and Scatter Operations



Gather and Scatter Operations

↓ processes

A ₀	A ₁	A ₂	A ₃	A ₄	A ₅



A ₀					
A ₁					
A ₂					
A ₃					
A ₄					
A ₅					

MPI_Gather

- `int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`

MPI_Gather

- Each process (root process included) sends the contents of its send buffer to the root process.
- The root process receives the messages and stores them in rank order.
- The outcome is *as if* each of the n processes in the group including the root process had executed a call to MPI_Send and the root had executed n calls to MPI_Recv.



MPI_Gather Simple Example

```
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank(MPI_COMM_WORLD,myrank);
if (myrank == root) {
    MPI_Comm_size(MPI_COMM_WORLD,&gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray,100,MPI_Int,rbuf,100,MPI_Int,root,
MPI_COMM_WORLD);
```

MPI_Scatter

- `int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);`

MPI_Scatter

- It is the inverse operation to MPI_Gather.
- The outcome is *as if* the root executed n send operations MPI_Send, and each process executed a receive MPI_Receive.



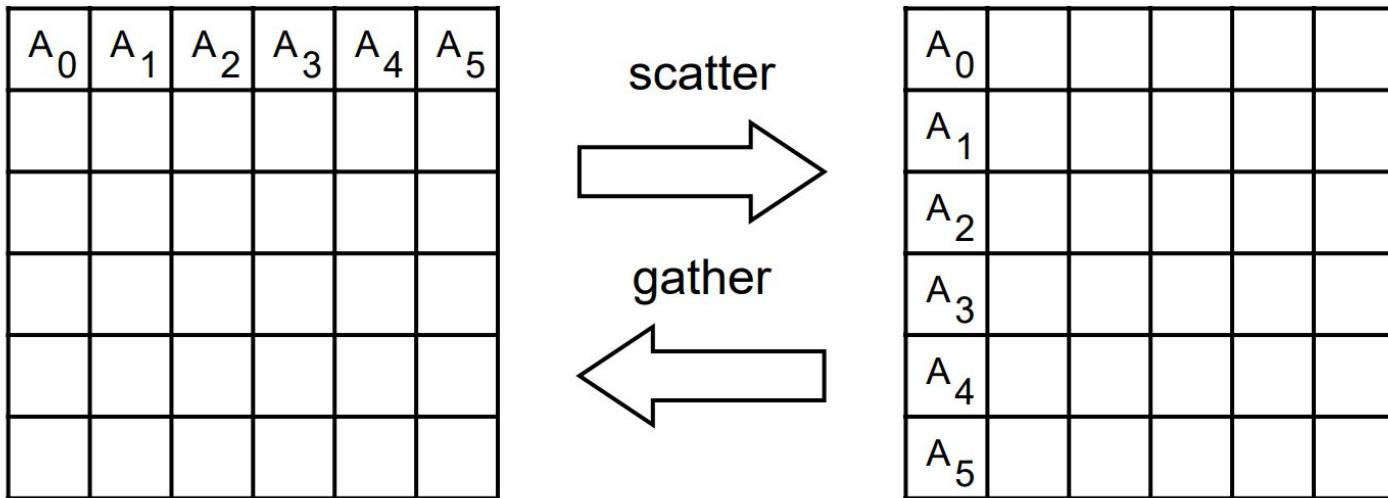
MPI_Scatter Simple Example

```
int gsize, *sendbuf; // Size of sendbuf is 100
int root, rbuf[20]; // Assuming we have 5 processes
MPI_Comm_size(MPI_COMM_WORLD, &gsize);
sendbuf = (int *)malloc(100 * sizeof(int));

// Assuming we have 5 processes
int sendcount = 20;

MPI_Scatter(sendbuf, sendcount, MPI_INT, rbuf, sendcount,
MPI_INT, root, MPI_COMM_WORLD);
```

Gather and Scatter Operations



Parallel Sum Calculation



Parallel Sum Calculation

Consider an array of integers with a size of N , where N is divisible by the number of processes P . The goal is to calculate the sum of all elements in the array using parallel processing with MPI. Each process is responsible for computing the sum of a portion of the array, and the final result is obtained by aggregating the partial sums.

Parallel Sum Calculation

Consider an array of integers with a size of N , where N is divisible by the number of processes P . The goal is to calculate the sum of all elements in the array using parallel processing with MPI. Each process is responsible for computing the sum of a portion of the array, and the final result is obtained by aggregating the partial sums.

You need to use the functions `MPI_Gather` and `MPI_Scatter`.

For this problem you can assume $N = 100$ and $P = 4$.

Parallel Sum Calculation

Also assume for this problem that the array contains all the integers from 1 to 100 , thus:

1,2,3, ..., 97,98,99,100

Arithmetic Series Formula

$$S_n = n \left(\frac{a_1 + a_n}{2} \right)$$

n = the number of terms being added

a_1 = the first term

a_n = the nth term (last term)

Arithmetico Series Formula

$$S_n = n \left(\frac{a_1 + a_n}{2} \right)$$

$$100 \times (101 / 2) \rightarrow 5050$$

MPI_Allgather



MPI_Allgather

- `int MPI_Allgather(const void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);`

MPI_Allgather

- It is used for gathering data from all processes in a communicator and distributing it to all processes in the same communicator. The data is gathered from each process and then broadcast to all other processes.

MPI_Allgather

- It is used for gathering data from all processes in a communicator and distributing it to all processes in the same communicator. The data is gathered from each process and then broadcast to all other processes.
- **MPI_Allgather** can be thought of as **MPI_Gather** except all processes receive the result instead of just the root.

MPI_Allgather

- The i th block of data sent from each process is received by every process and placed in the i th block of the receive buffer.
- The outcome of a call to **MPI_Allgather** is *as if* all processes executed n calls to **MPI_Gather** for $\text{root} = 0, \dots, n-1$.

Parallel Maximum Element Search



Parallel Maximum Element Search

Consider a scenario where you have an array of integers distributed across multiple processes, and you want to find the maximum element from the entire array using MPI. Each process initially holds a portion of the array, and you need to efficiently find the global maximum across all processes.

Parallel Maximum Element Search

Assume the global array size is 50 and you have 5 processes. Each process will initialize its local array with $50/5 = 10$ elements. Now, this initialization will be done using a random number generator. Its seed will depend on the rank of the process. Namely,

$$\text{seed} = \text{rank} + 1$$

Once this has been initialized, each process will compute its local maximum and then you need to use the function **MPI_Allgather** to gather these local maximums in order to compute the global maximum.

Parallel Maximum Element Search



Parallel Maximum Element Search

Consider a scenario where you have an array of integers distributed across multiple processes, and you want to find the maximum element from the entire array using MPI. Each process initially holds a portion of the array, and you need to efficiently find the global maximum across all processes.

Parallel Maximum Element Search

Assume the global array size is 50 and you have 5 processes. Each process will initialize its local array with $50/5 = 10$ elements. Now, this initialization will be done using a random number generator. Its seed will depend on the rank of the process. Namely,

$$\text{seed} = \text{rank} + 1$$

Once this has been initialized, each process will compute its local maximum and then you need to use the function **MPI_Allgather** to gather these local maximums in order to compute the global maximum.

MPI_Reduce Operations



MPI_Reduce Operations



On this lecture we are going to see the operations we can perform with MPI_Reduce.

MPI_Reduce Operations



- `int MPI_Reduce(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`

MPI_Reduce Operations

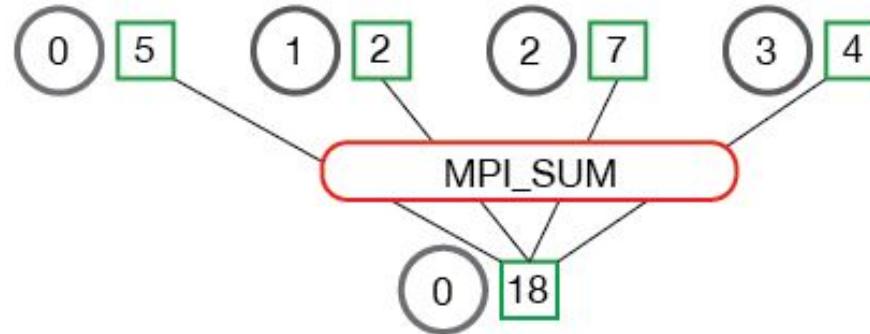


Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

MPI_Reduce Operations



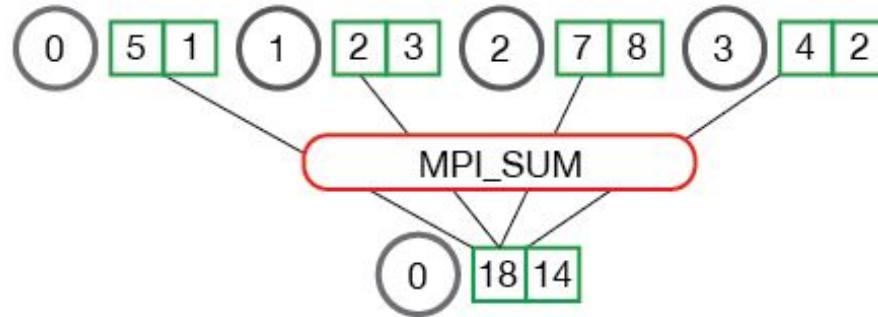
MPI_Reduce



MPI_Reduce Operations



MPI_Reduce



Difference between Bitwise AND - Logical AND



MPI_BAND

$$\begin{array}{r} 6 \quad 10 \\ 1010 \\ 0110 \\ \hline 0010 \end{array}$$

MPI LAND

$$\begin{array}{r} 6 \quad 10 \\ 1 \quad 88 \quad 1 \\ \sim \\ 1 \end{array}$$

Solved Problems - Collective Communications



Problem 1



Problem 1



Make a program to compute the average of values with MPI_Reduce. Each process will have a local array with elements in the range [0,1]. These local arrays must be created at random. The program receives only one input argument, which is the size of each local array.

Once these local arrays have been initialized, each process will compute its local sum and local average. Then, you need to reduce all these local sums to compute the global sum. Finally, you need to compute the global average. You should print all the important information in the program.

Solved Problems - Collective Communications



Problem 2



Problem 2



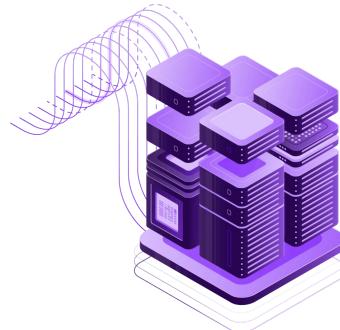
Consider a scenario where you have a large dataset that needs to undergo a specific transformation, and the goal is to efficiently parallelize this transformation process using MPI. The dataset is initially distributed among MPI processes, and each process is responsible for transforming its local portion of the data. After the transformation, the results should be gathered to construct the final transformed dataset.



Problem 2



In this problem, you can consider the DATA_SIZE to be 100 and the number of processes to be 5. Also you can assume that the DATA_SIZE is a multiple of the number of processes so that we can evenly distribute the work for each process.



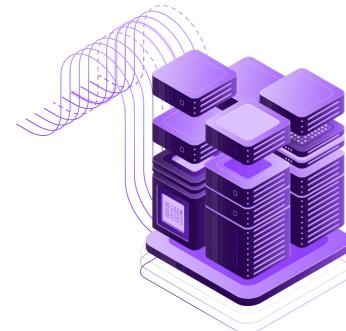


Problem 2

You can suppose that the dataset contains integer numbers in the range `[1,DATA_SIZE]`.

The transformation is the square function, so that:

Given an element x , the transformed element is $x \bullet x$



Problem 2



Hint: This is a problem that involves using **`MPI_Scatter`** to scatter the data, and then **`MPI_Gather`** to gather the results.



Sharing Data



Sharing Data



- `int MPI_Pack(const void* inbuf, int incount, MPI_Datatype datatype,
void* outbuf, int outsize, int* position, MPI_Comm comm);`
- `int MPI_Unpack(const void* inbuf, int insize, int* position,
void* outbuf, int outcount, MPI_Datatype datatype,
MPI_Comm comm);`



MPI_Pack

- It is used for packing data into a contiguous buffer. The purpose of packing is to prepare data for transmission, typically when we want to send non-contiguous data in a single message. This function is often used in conjunction with MPI_Unpack on the receiving side.



MPI_Unpack

- This function is used for unpacking data that was previously packed using MPI_Pack.

Contiguous Buffer



- A contiguous buffer refers to a block of memory where the elements are stored consecutively without any gaps.

```
int array[5] = {1, 2, 3, 4, 5};
```

Problem Statement



Create a program that reads an integer and a double-precision value from standard input (from process 0).

Then, it communicates this to all of the other processes with an MPI_Bcast call. Use MPI_Pack to pack the data into a buffer.



Problem Statement



The program will continuously ask for the two input values.
At each iteration, the evolution needs to be shown.
Finally, if the user introduces a negative value input, all the processes will terminate.





MPI_Unpack

- This function is used for unpacking data that was previously packed using MPI_Pack.

Broadcast



- **MPI_Bcast** distributes data from one process (the root) to all others in a communicator.

```
int array[100];
int root = 0;
...
MPI_Bcast(array,100,MPI_INT,root,MPI_COMM_WORLD);
```

Introduction to Non-blocking Operations



Non-blocking Operations

- We can improve performance on many systems by overlapping communication and computation.
- **Non-blocking** operations in MPI refer to communication routines that allow a program to continue its execution without waiting for the completion of the communication operation.

Blocking Operations Review



- `int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`

Non-blocking Operations

- In contrast to blocking operations, where the program halts until the communication is finished, non-blocking operations enable overlap of computation and communication.
- The program can continue with other computations while the communication is in progress.

Non-blocking Operations

- They are **asynchronous**: Non-blocking operations initiate communication and then return control to the program immediately without waiting for the communication to complete.

Non-blocking Operations

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`

MPI_Isend

- Initiates the sending of a message but does not wait for the completion of the communication. It allows the program to continue with other computations or communication operations while the data is being sent in the background.

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`

MPI_Irecv

- Initiates the receiving of a message but does not wait for the completion of the communication. It allows the program to continue with other computations or communication operations while waiting for incoming data in the background.
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`

Non-blocking Operations Application



Non-blocking Operations Application

Create a program with two processes. Process 0 will initialize the following array **message** of 8 positions:

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Next, it will send this array to the Process 1 using a non-blocking operation. Then, it will compute the sum of the array **message**.

Non-blocking Operations Application

Process 1 will receive the **message** array using a non-blocking operation. It will then compute the sum of this array.

In the program show all the summations results for both processes. In addition, tell the user when the Process 0 has sent the array and when the Process 1 has received it.

What are the results? How can we perform this task by ensuring the synchronization between the processes and compute always the correct sum of the **message** array in the Process 1?

Non-blocking Operations

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`

MPI_Isend

- Initiates the sending of a message but does not wait for the completion of the communication. It allows the program to continue with other computations or communication operations while the data is being sent in the background.

- `int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request);`

MPI_Irecv

- Initiates the receiving of a message but does not wait for the completion of the communication. It allows the program to continue with other computations or communication operations while waiting for incoming data in the background.
- `int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request);`