4

Distributed Training

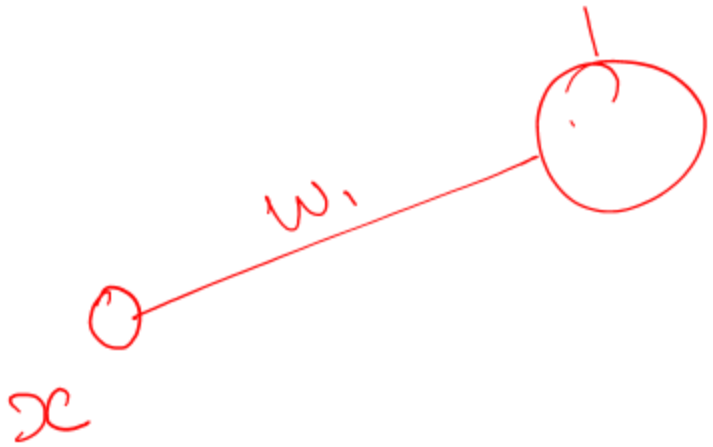
Contents

Model parallelism with OpenMP

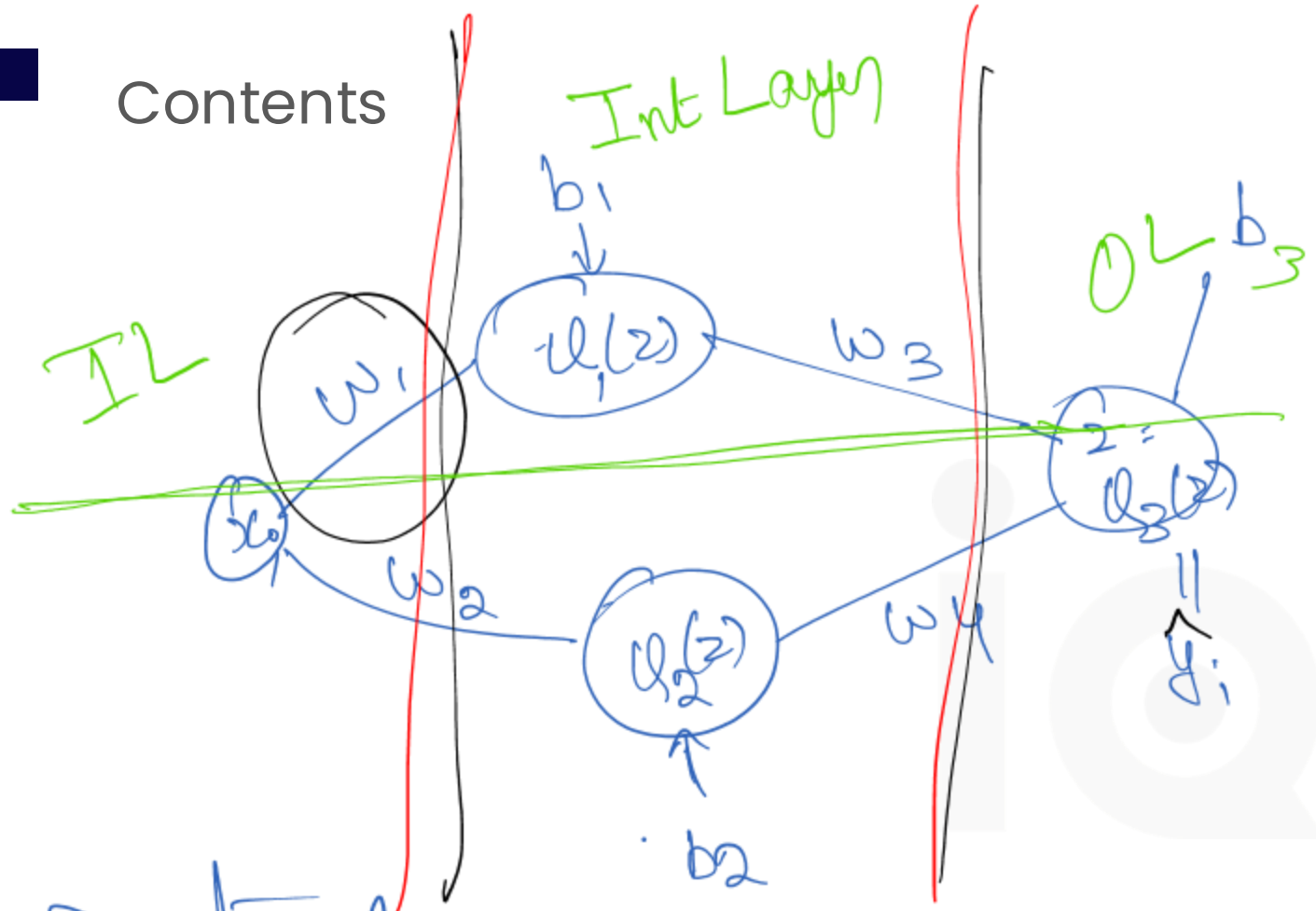
Data Parallelism with MPI

Summary

Contents

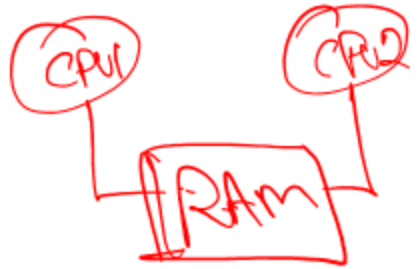


Contents



How to parallelize it?

I. SMP

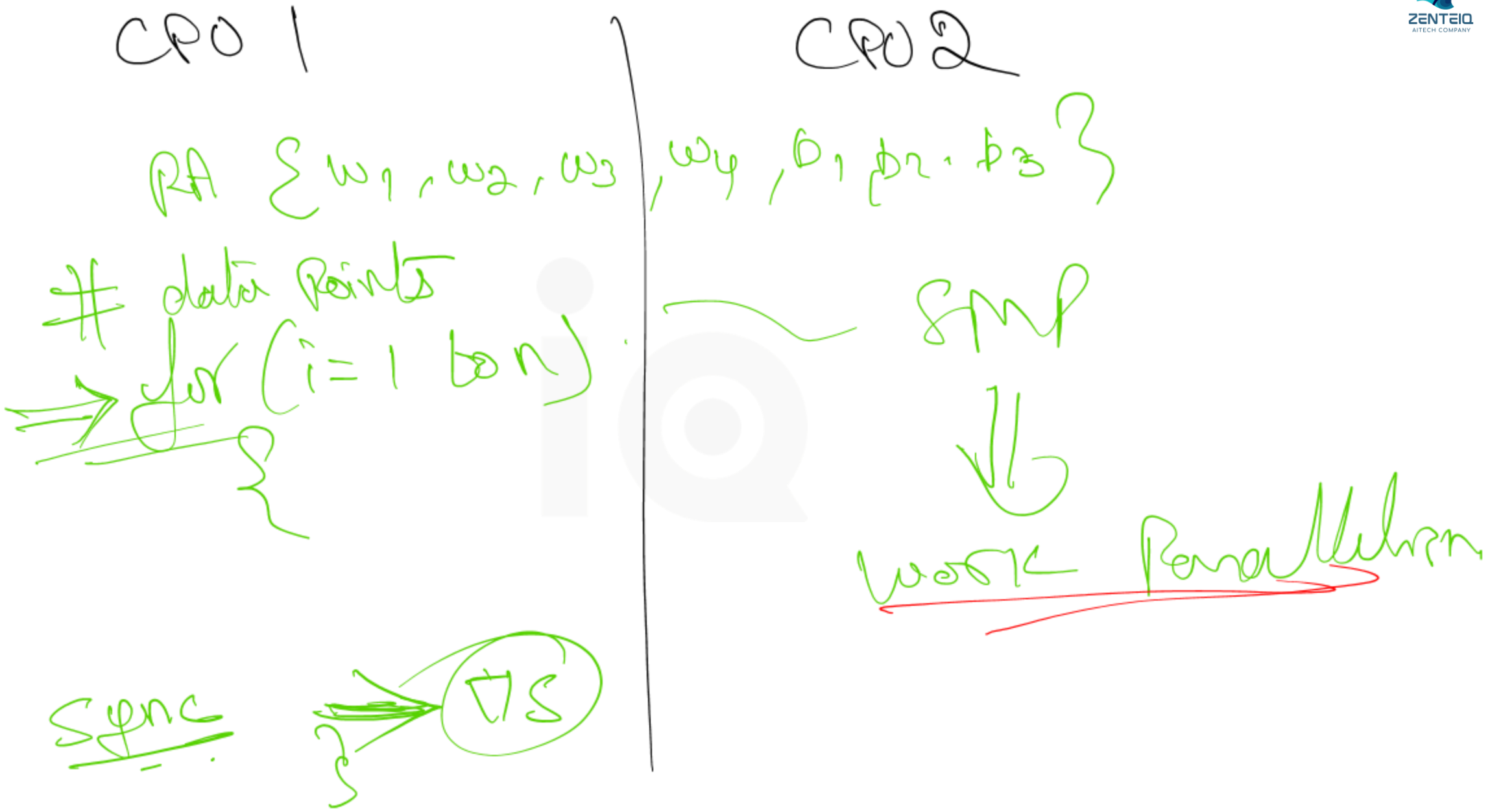


$$\min S(w,b) = \sum_{i=1}^{n} \| y_i - \hat{y} \|_0$$

$$w_1^{k+1} = w_1^{k} - \eta \frac{\partial S}{\partial w_1}$$

$$W^{k+1} = W^{k} - \eta \nabla_W S$$

Use Autodiff



Major Parallel Programming Models

- OpenMP for Model Parallelism
  - For shared memory parallelism (SMP)
  - To define the work decomposition No data decomposition
  - Synchronization is implicit (can also be user-defined)
- MPI (Message Passing Interface) for Data Parallelism
  - The user specifies how the work and data are distributed
  - The user specifies how and when communication has to be done by calling MPI communication library routines

Parallel Programming for GPUs

Feature	SYCL	HIP	OpenCL	CUDA	Kokkos
Ease of Learning	High	Medium	Medium	Medium	Low
Knowing C++	Essential	Essential	Essential	Helpful (C-like)	Advanced C++
Performance	High (portable)	Very High (AMD)	High	Very High (NVIDIA)	Very High
Robustness	High (growing ecosystem)	High (mature on AMD)	High (mature)	High (mature)	High (active development)
Portability	Excellent	Good (AMD focus)	Excellent	Limited (NVIDIA)	Excellent
Debugging	Improving	Good	Good	Good	Can be challenging
Community & Ecosystem	Growing rapidly	Strong (AMD support)	Large and mature	Very large and mature	Active and growing
Abstraction Level	High	Low	Low	Low	Medium

Introduction to Multi-Processing for Model Parallelism

SMP Programming Models

OpenMP for Model Parallelism

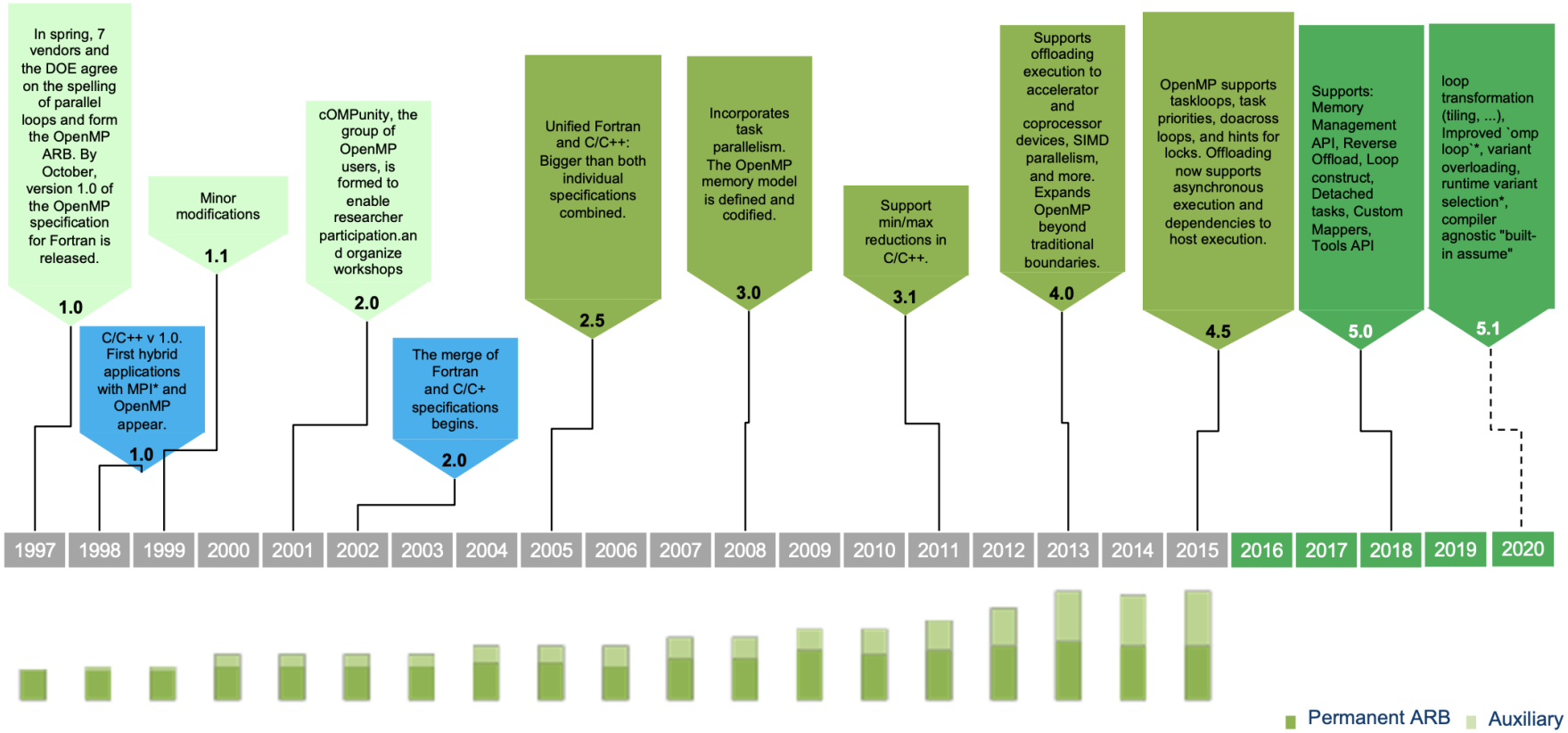
- an implementation of multithreading (not Multiprocessing)
- an API for writing multithreaded application on shared memory systems
- to define the work decomposition
- no data decomposition
- synchronisation is implicit (can also be user defined)

History

- Introduced in 1997
- Latest 5.2, (Nov 2021)
- https://www.openmp.org/specifications/
- https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

SMP Programming Models

History of OpenMP: 1997 - 2020



OpenMP

OpenMP Technical Report 12: Version 6.0 Preview 2

2024

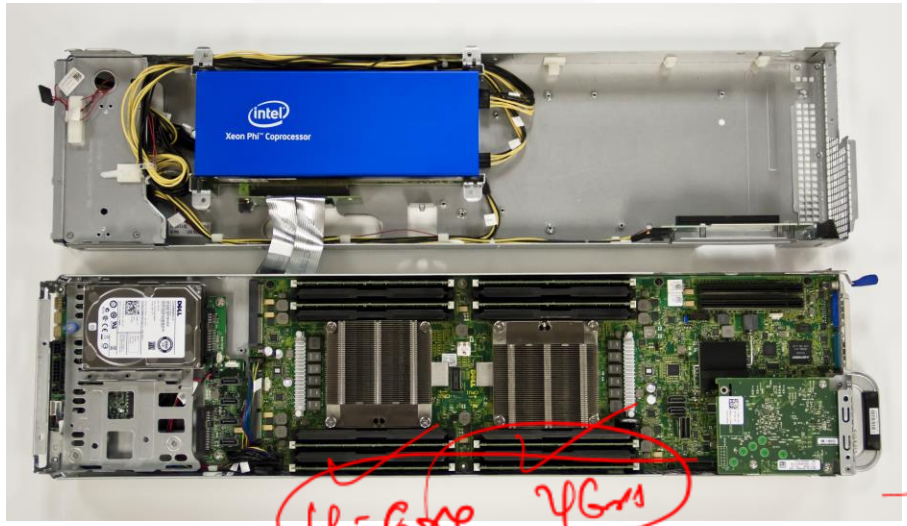
National Nuclear Security Administration | U.S. DEPARTMENT OF ENERGY | Office of Science | ECP

Source: https://ecpannualmeeting.com/assets/overview/sessions/ff2020%20ECP-Tutorial-with-ECP-template.pdf

SMP Programming Models

OpenMP targets:

- A node with one or more sockets
- each socket with many CPU cores
- each core is an independent processing unit, with access to all the memory on the node.

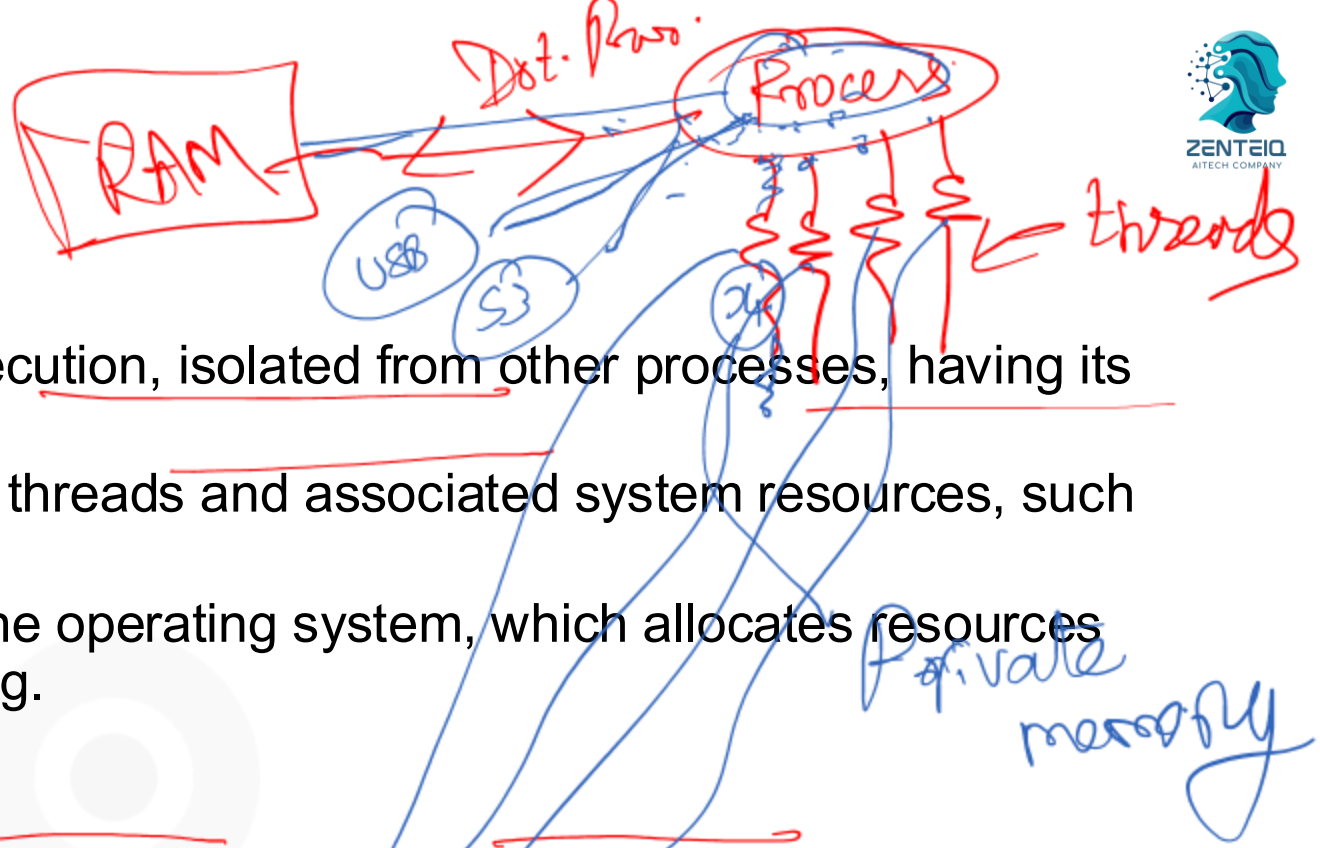


A node with two sockets and a co-processor

Thread-based parallelism in Python

https://docs.python.org/3/library/threading.html

OpenMP for Model Parallelism



Process

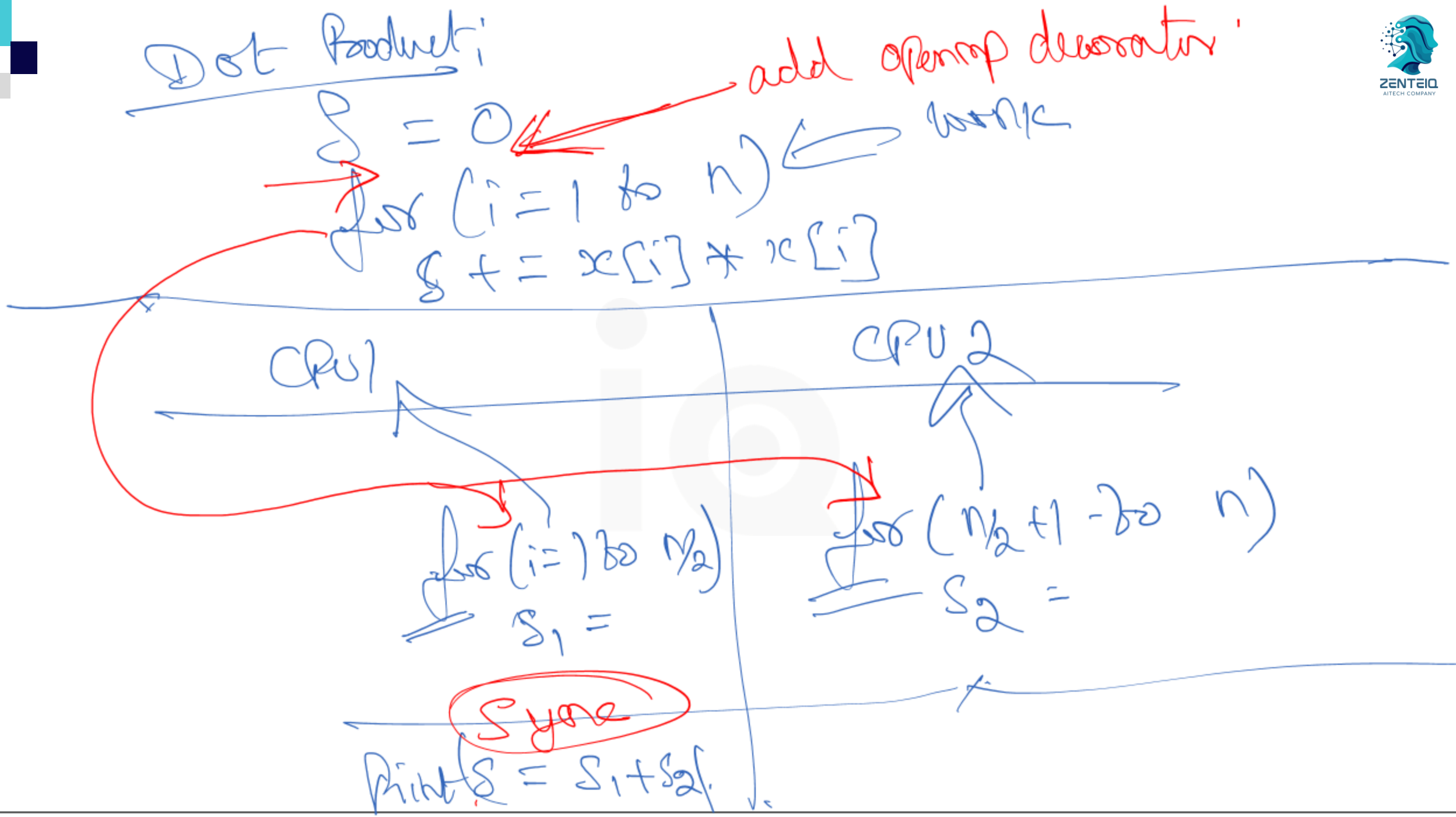
- an independent program in execution, isolated from other processes, having its own address space.
- It is a collection of one or more threads and associated system resources, such as file handles and memory.
- Each process is managed by the operating system, which allocates resources and handles process scheduling.

Thread

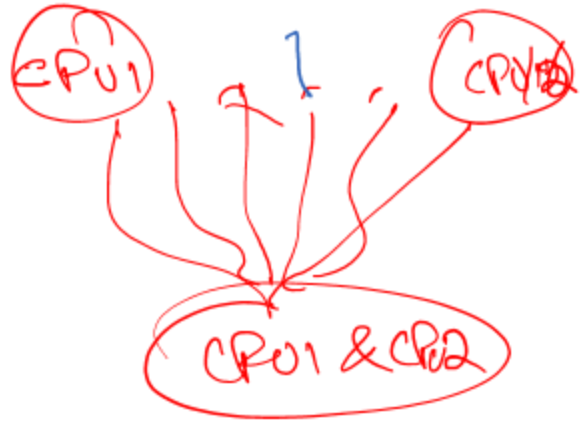
- A thread, on the other hand, is the smallest unit of execution within a process
- Shares the process's resources but executes independently.
- Multiple threads within a process share the same address space, allowing them to communicate and share information more easily than separate processes

OpenMP Thread

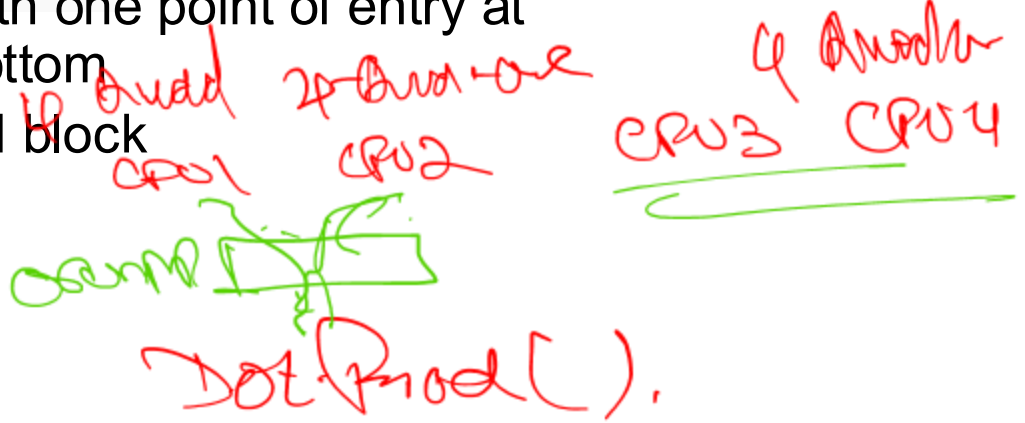
- A thread that is managed by the OpenMP implementation.
- OpenMP is NOT an automatic parallel programming model



OpenMP Core Syntax



- Most of the constructs in OpenMP are compiler directives.
  - #pragma omp construct [clause [clause]…]
  - #pragma omp parallel num_threads(4)
- Header file: #include <omp.h>
  - Contains function prototypes and types
- Most OpenMP constructs apply to a “structured block”:
  - a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - Can have exit(0) within the structured block



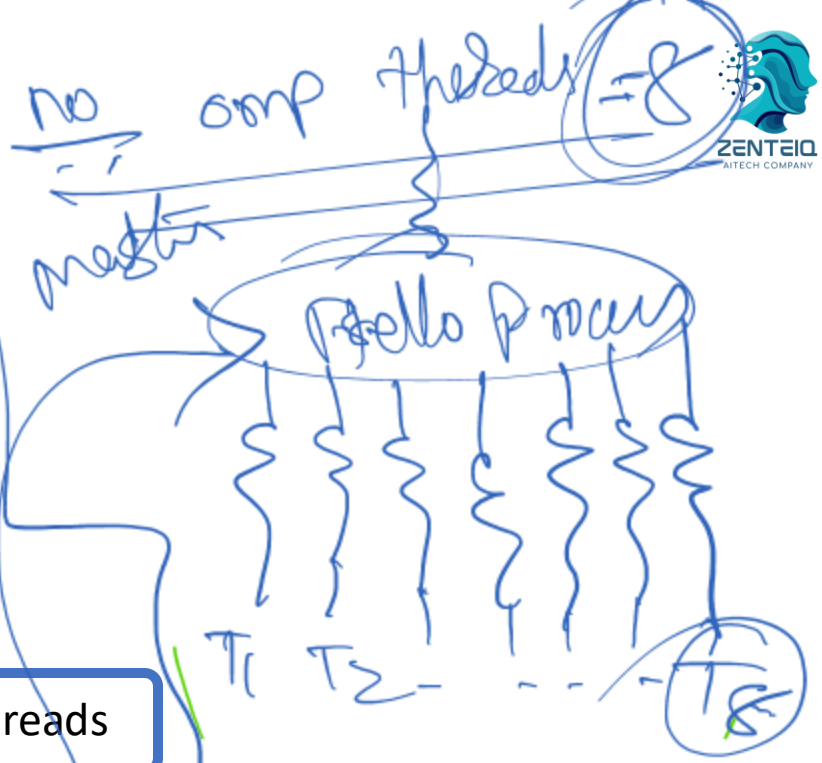
OpenMP: Hello World!

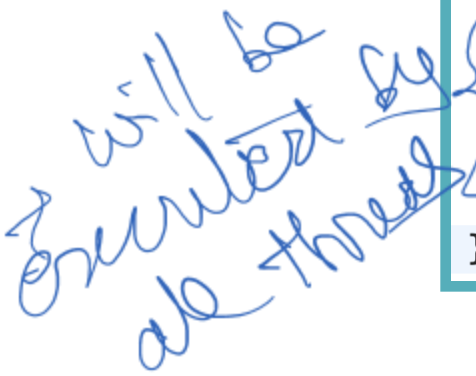
```cpp
#include <iostream>

int main() {
    std::cout << "Hello World!"<<std::endl;
    return 0;
}
```

parallel region with default number of threads

```cpp
int main (int argc, char *argv[])
{
  #pragma omp parallel
  {
    printf ("Hello world! I'm thread %d out of %d threads.\n",
            omp_get_thread_num(), omp_get_num_threads());
  }
  return 0;
}
```

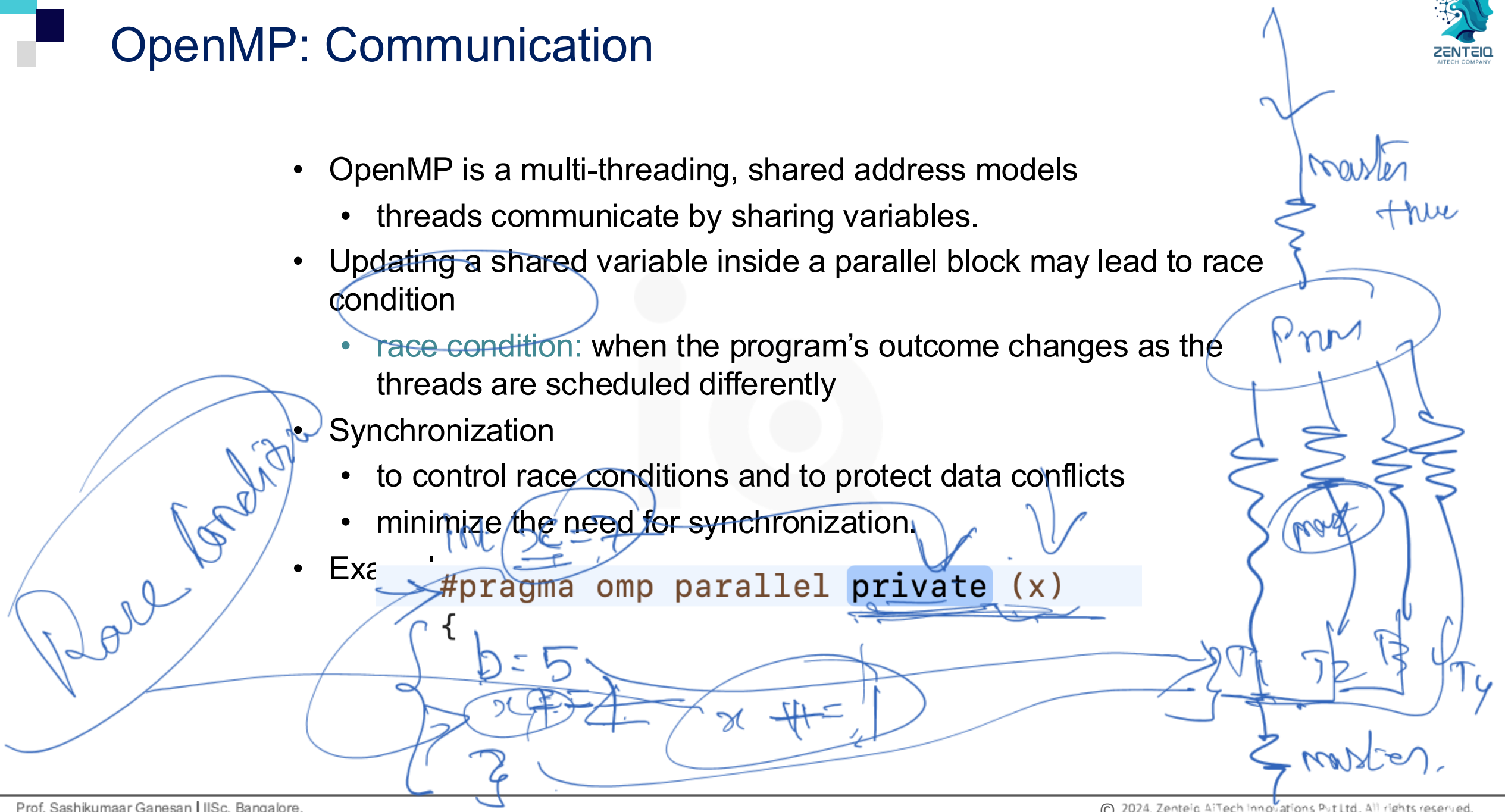




OpenMP: Communication

- OpenMP is a multi-threading, shared address models
  - threads communicate by sharing variables.
- Updating a shared variable inside a parallel block may lead to race condition
  - race condition: when the program's outcome changes as the threads are scheduled differently
- Synchronization
  - to control race conditions and to protect data conflicts
  - minimize the need for synchronization.
- Exa

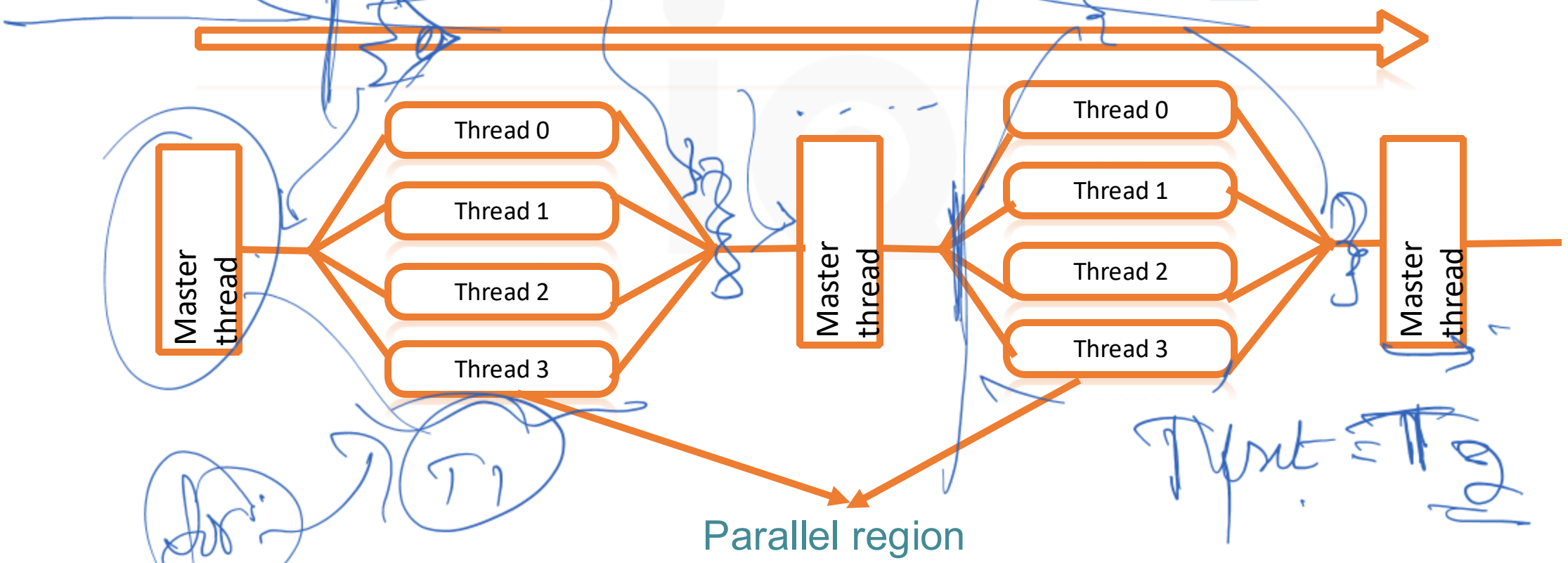
```
#pragma omp parallel private (x)
{
```



OpenMP: Creating Threads

Fork-join parallelism

- Master thread spans a team of threads as and when needed



OpenMP: Creating Threads

Fork-Join parallelism

- All OpenMP Programm starts with one thread: Master thread
- Fork: Master thread spans a team of threads as and when parallel region constructor is encountered
- Parallel Region: Master thread creates a team of parallel threads
- Statements within the parallel region are executed in parallel among the various team threads
- Join: Team of threads completes the statements in the parallel region, synchronise and terminate, leaving only the master thread

OpenMP: Terminology

Components

- Compiler directives
- Runtime routines, e.g., omp_set_num_threads()
- Environment variables, e.g., OMP_NUM_THREADS

OpenMP: Terminology

Directives

- In C/C++ the pragma mechanism is used: annotations for the benefit of the compiler that are otherwise not part of the language
- directives have the format

```
#pragma omp somedirective clause(value, othervalue)
{
  parallel statement 1;
  parallel statement 2;
}
```

- #pragma omp key to indicate that an OpenMP directive is coming
- somedirective such as parallel and possibly clauses with values.
- After the directive comes either a single statement or a block in curly braces to indicate the parallel region

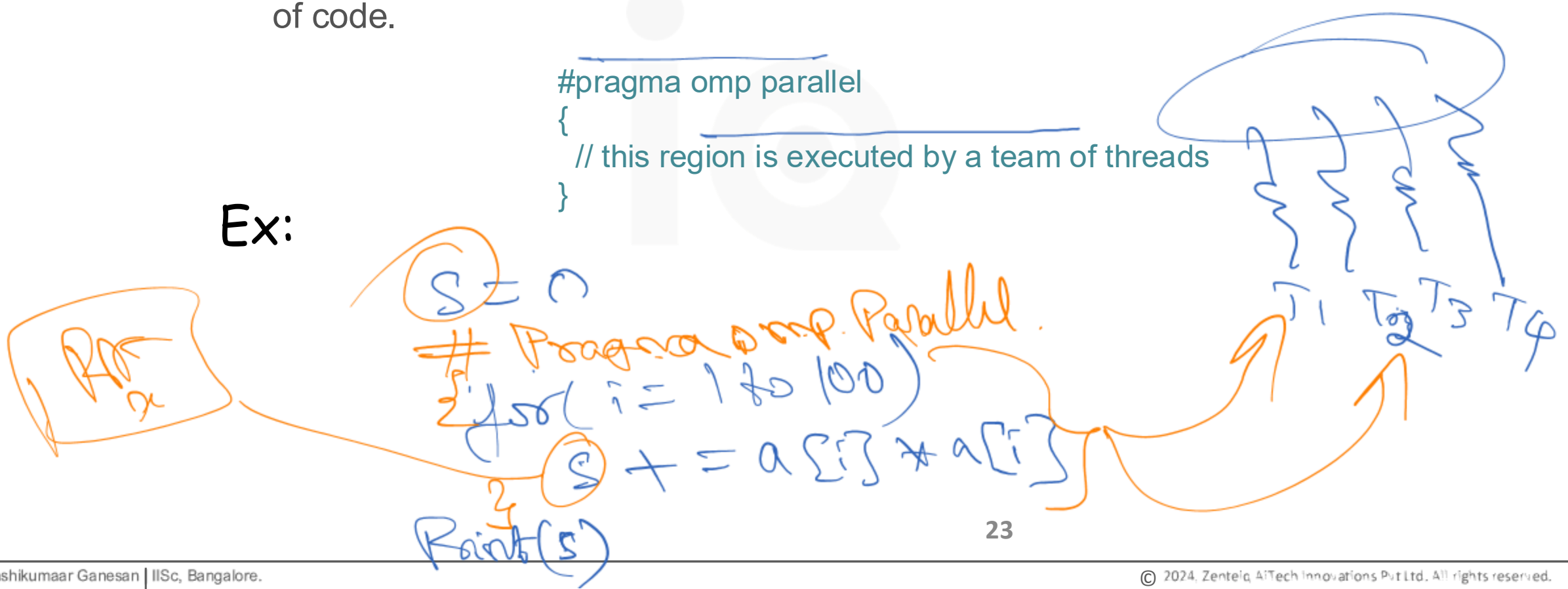
OpenMP: Parallel Region

Parallel region

- instance of the SPMD model: all threads execute the same segment of code.

```
#pragma omp parallel
{
  // this region is executed by a team of threads
}
```

Ex:



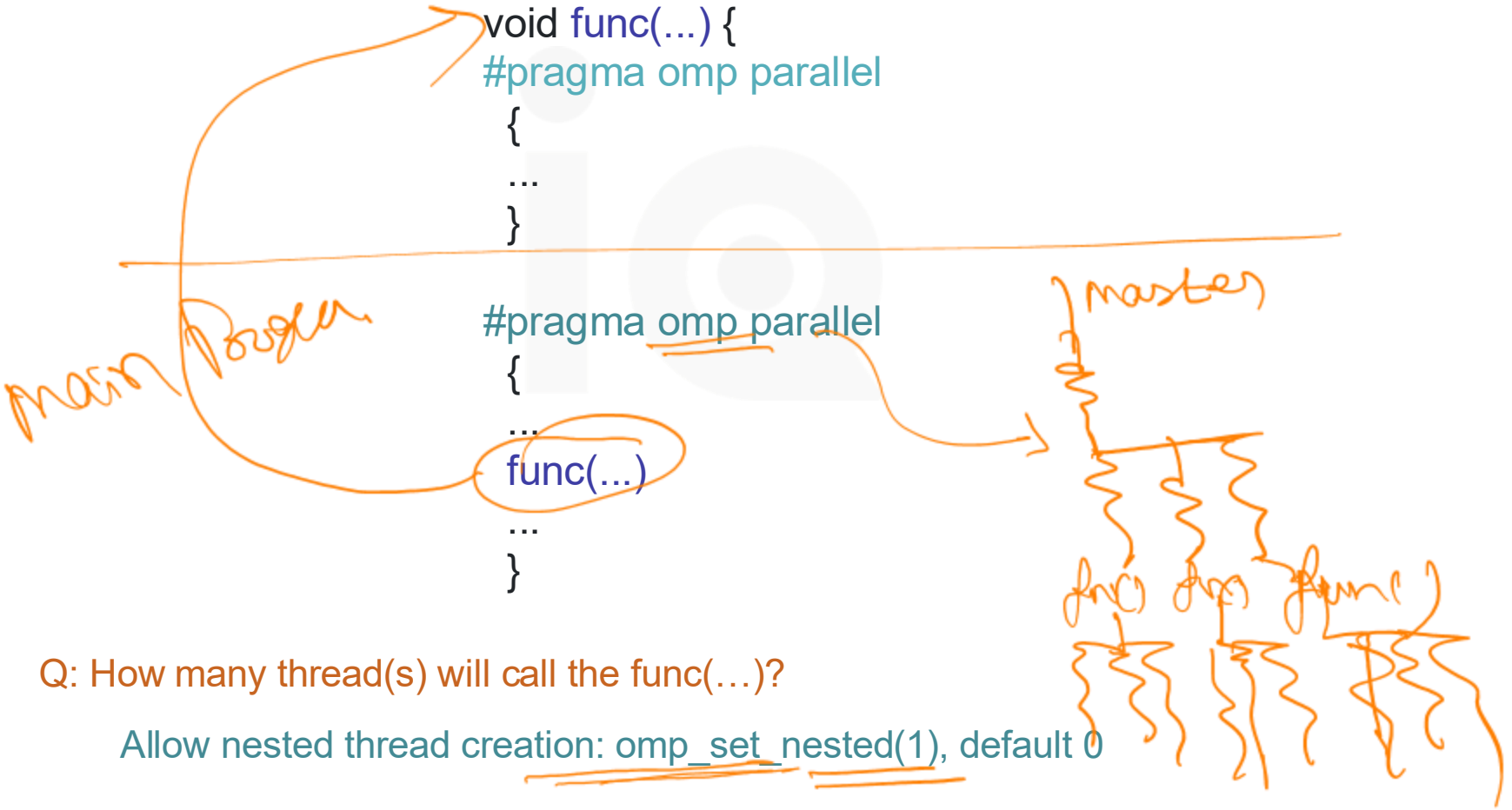
OpenMP: Parallel Region

Nested Parallel

- Calling a function from inside a parallel region, and that function itself contains a parallel region

```
void func(...) {
#pragma omp parallel
  {
  ...
  }

#pragma omp parallel
  {
  ...
  func(...)
  ...
  }
```



Q: How many thread(s) will call the func(…)?

Allow nested thread creation: omp_set_nested(1), default 0

OpenMP: Work Sharing

Work sharing

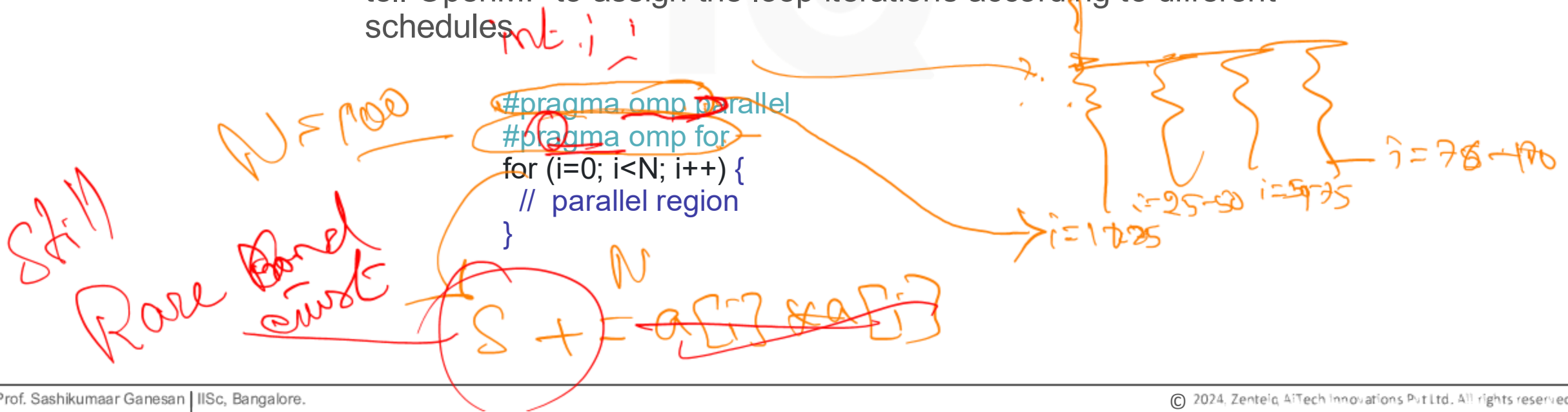
- worksharing constructs: constructs that take an amount of work and distribute it over the available threads in a parallel region
- for (in C) or do (in Fortran): Threads divide up the loop iterations among themselves
- sections: Threads divide a fixed number of sections between themselves
- single: The section is executed by a single thread
- task Will be discussed in the next lecture

OpenMP: Parallel Region

Loop Parallelism

- worksharing constructs:
  - constructs that take an amount of work and distribute it over the available threads in a parallel region
- a number of different ways possible:
  - No need to calculate the loop bounds for the threads, but can also tell OpenMP to assign the loop iterations according to different schedules

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  //  parallel region
}
```

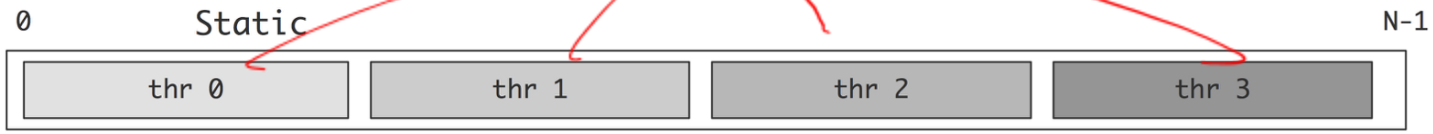


OpenMP: Parallel Region

Loop Schedules

- Usually many iterations in a loop than there are threads
- OpenMP lets you specify this with the schedule clause

#pragma omp for schedule(static)



#pragma omp for schedule(static, [chunk])

- Deal-out blocks of iterations of size “chunk” to each thread.
- Done at compile time

OpenMP: Parallel Region

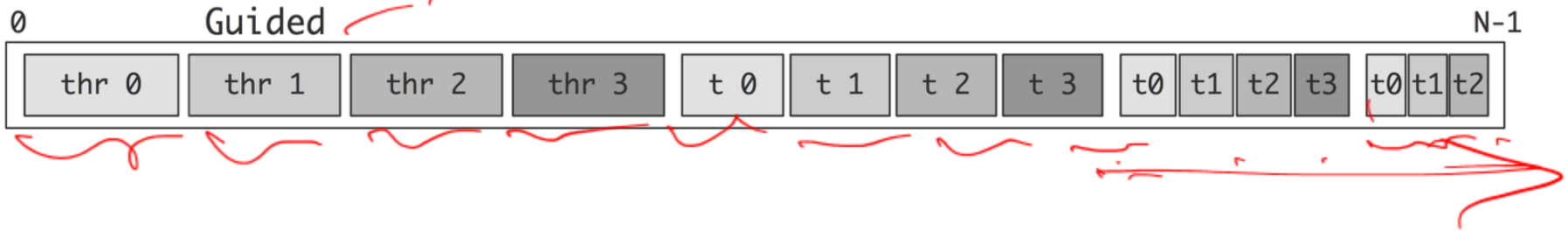
Loop Schedules

#pragma omp for schedule(dynamic, [chunk])

- Each thread grabs “chunk” iterations off a queue until all iterations have been handled
- Scheduling logic used at run-time

#pragma omp for schedule(guided, [chunk])

- Iterations dynamically assigned, with reducing chunk size, till all iterations have been handled

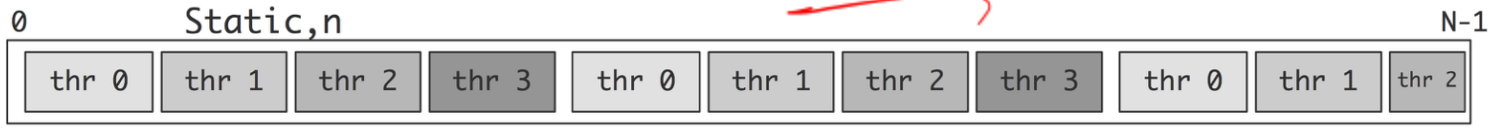


OpenMP: Parallel Region

Loop Schedules

#pragma omp for schedule(runtime)

- Each thread grabs “chunk” iterations off a queue until all iterations have been handled
- The “chunk” size is taken from the OMP_SCHEDULE



#pragma omp for schedule(auto)

- The schedule is left up to the runtime
- Need not be any of the above type

OpenMP: Parallel Region

Which Schedule clause to use

#pragma omp for schedule(static)

- Knowing that the workload of each iterator is same (uniform compute time)
- Pre-determined and predictable by the programmer
- Least work at runtime, scheduling done at compile time

#pragma omp for schedule(dynamic)

- the workload of each iterator is unknown/unpredictable (nonuniform compute time)
- More work at runtime, complex scheduling logic used at run time

OpenMP: Parallel Region

EX: Write a simple OpenMP parallel code for

1. Matrix addition
2. Matrix vector multiplication

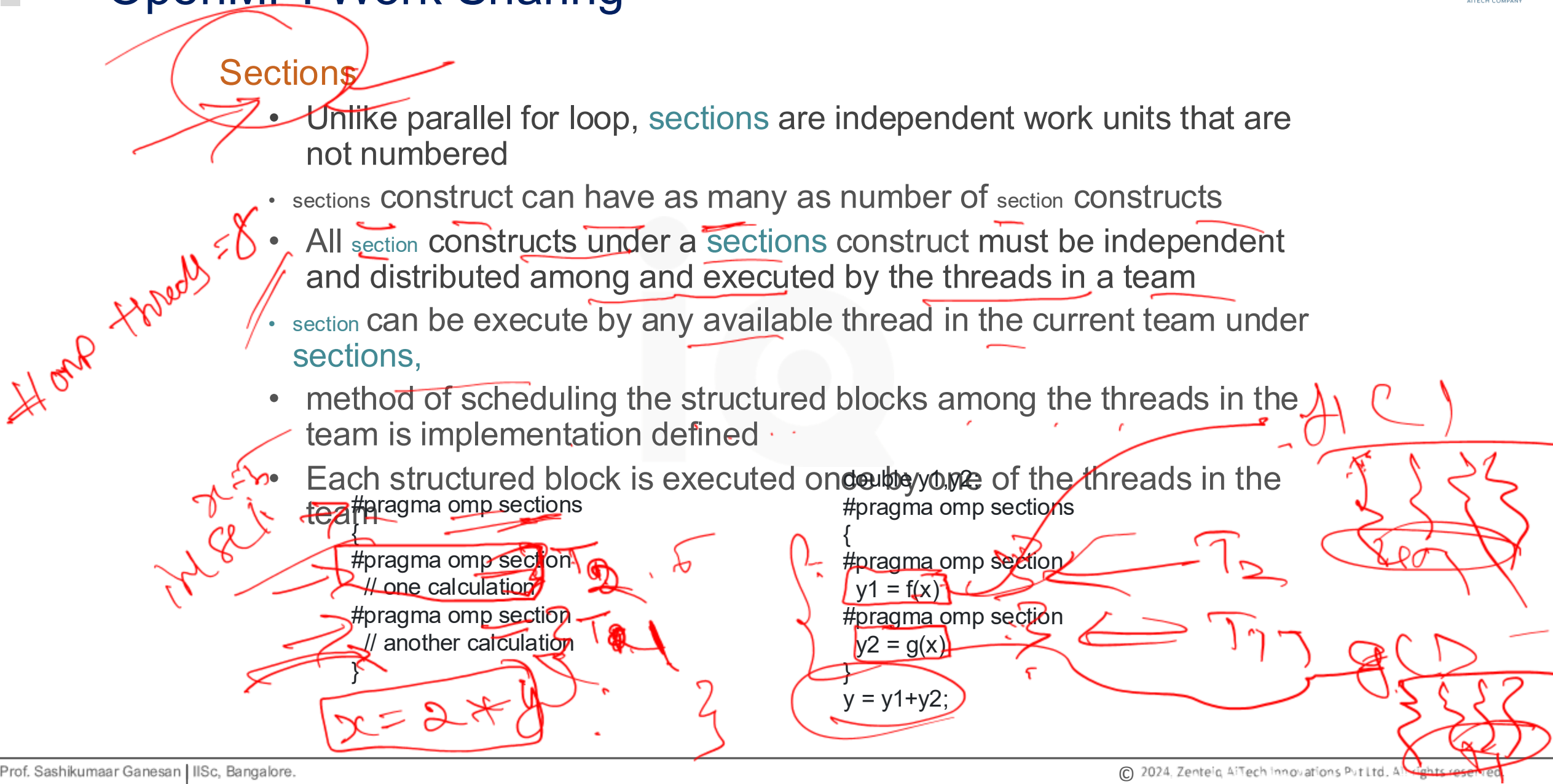
OpenMP: Work Sharing

Sections

- Unlike parallel for loop, sections are independent work units that are not numbered
- sections construct can have as many as number of section constructs
- All section constructs under a sections construct must be independent and distributed among and executed by the threads in a team
- section can be execute by any available thread in the current team under sections,
  - method of scheduling the structured blocks among the threads in the team is implementation defined
  - Each structured block is executed once by one of the threads in the team

```
#pragma omp sections
{
#pragma omp section
 // one calculation
#pragma omp section
 // another calculation
}
```

```
double y1,y2;
#pragma omp sections
{
#pragma omp section
 y1 = f(x)
#pragma omp section
 y2 = g(x)
}
y = y1+y2;
```



OpenMP: Work Sharing

Single/Master

- it limits the execution of a block to a single threads
- can be used for IO operations
- can be used to initialize


```
#pragma omp parallel
{
#pragma omp single
  printf("We are starting this section!\n");
  // parallel stuff
}
```

OpenMP: Controlling thread data

Storage attributes

- Shared
- Private
- Firstprivate
- Lastprivate

OpenMP: Controlling thread data

Shared Data

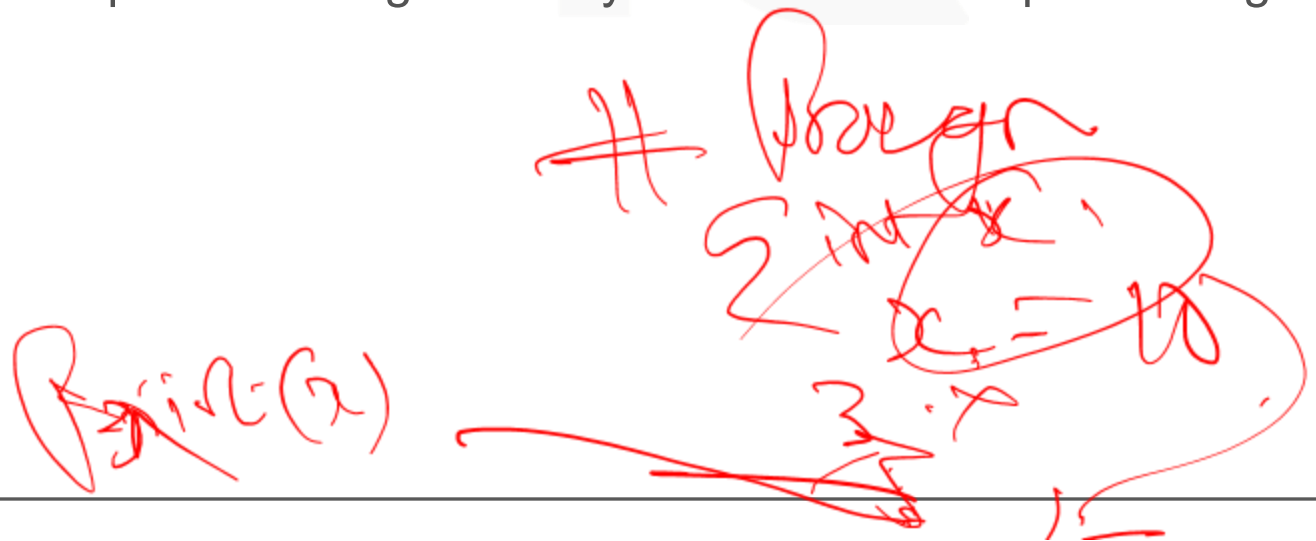
- any data declared outside the parallel region is shared
- All threads use the same memory location of a shared variable

```
int x = 5;
#pragma omp parallel
 {
   x = x+1;
   printf("shared: x is %d\n",x);
 }
```

OpenMP: Controlling thread data

Private Data

- it is possible to declare variables inside a lexical scope; inside curly braces
- any variable declared in a parallel block following an OpenMP directive will be local to the executing thread
- computed value goes away at the end of the parallel region



OpenMP: Controlling thread data

Data In Dynamic Scope

- Any variables locally defined to the function are private
- static variables in C and save variables in Fortran are shared
- function arguments inherit their status from the calling environment

Temporary Variables In A Loop

- a variable that is set and used in each loop iteration

OpenMP: Controlling thread data

Default

- Any variables locally defined to the function are private
- Loop variables in an omp for are private;
- Local variables in the parallel region are private
- Statically allocated data can either be shared or private
- Dynamically allocated data can only be shared

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

OpenMP: Controlling thread data

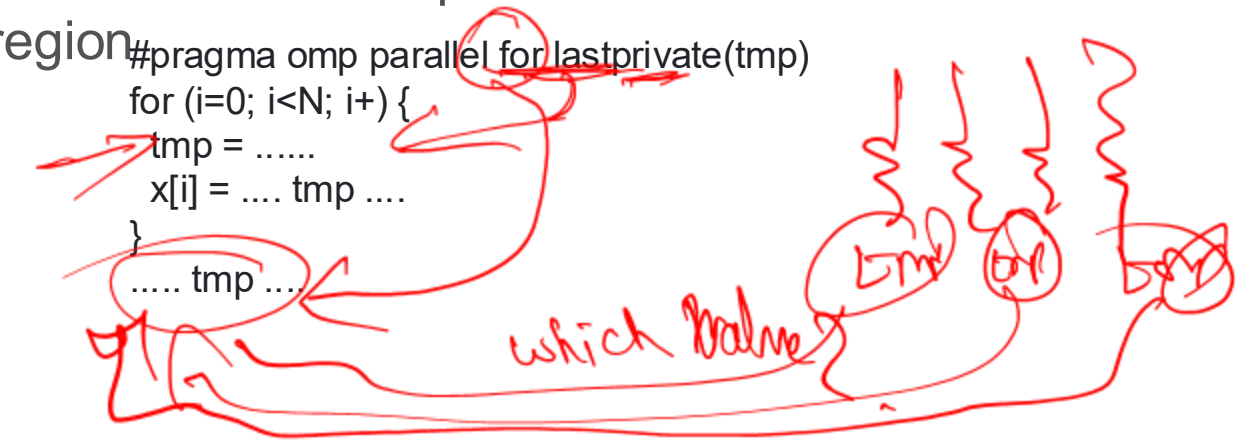
First And Last Private

- private variables are created with an undefined value
- can force their initialization with

```
int t=2;
#pragma omp parallel firstprivate(t)
{
  t += f( omp_get_thread_num() );
  g(t);
}
```

- private value to be preserved to the environment outside the parallel region

```
#pragma omp parallel for lastprivate(tmp)
  for (i=0; i<N; i++) {
    tmp = ......
    x[i] = .... tmp ....
  }
..... tmp ....
```

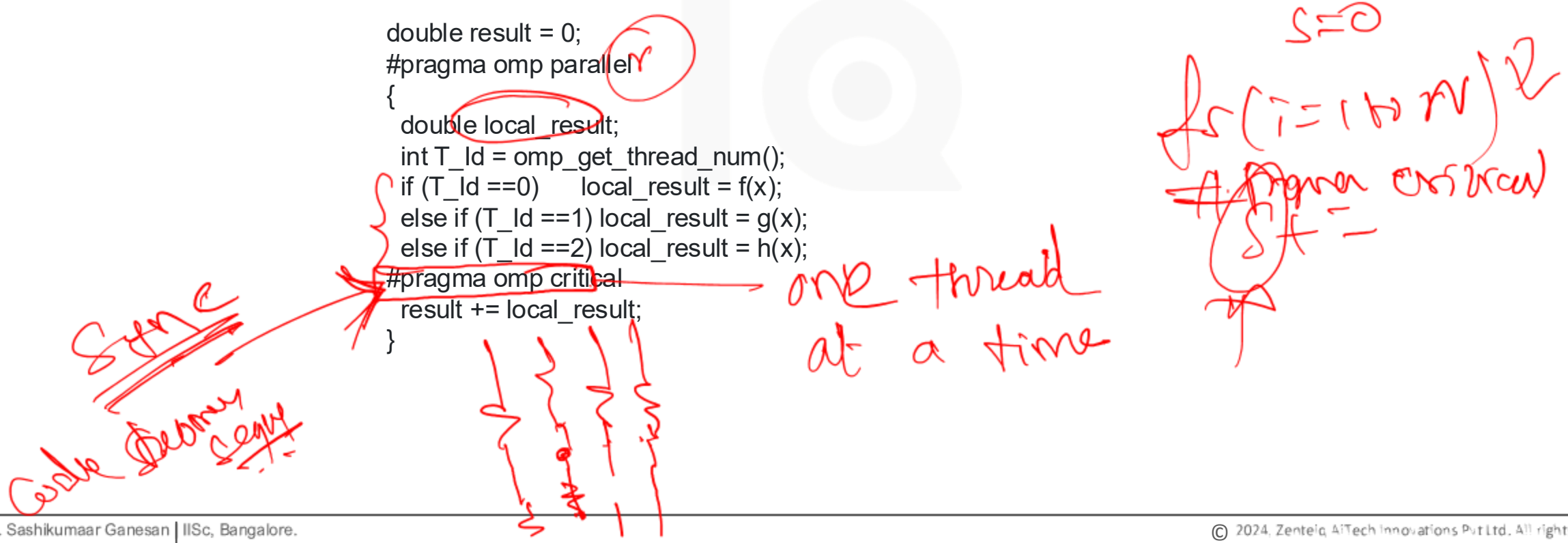


OpenMP: Reductions

Handling Race condition

- parallel tasks often produce some quantity that needs to be summed or otherwise combined
- How to avoid race condition?

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  int T_Id = omp_get_thread_num();
  if (T_Id ==0)      local_result = f(x);
  else if (T_Id ==1) local_result = g(x);
  else if (T_Id ==2) local_result = h(x);
#pragma omp critical
  result += local_result;
}
```



OpenMP: Reductions

Built-In Reduction

- Arithmetic reductions: +,∗,−,max,min
- Logical operator reductions in C: & && | || ^

```
double avg, A[MAX];
#pragma omp parallel for reduction(+:avg)
  for (int i=0; i<MAX; i++)
    avg += A[i];

avg /= MAX;
```

```
  x = init_x
#pragma omp parallel for reduction(min:x)
  for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

OpenMP: Reductions

Reduction operands/initial-values

Operator	*Initial value*
+	*0*
-	*0*
***	*1*
min	*Largest possible value*
max	*Most negative number*

OpenMP: Synchronization

Explicit Barrier

- barrier defines a point in the code where all active threads will stop until all threads have arrived at that point

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  #pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```

Implicit Barrier

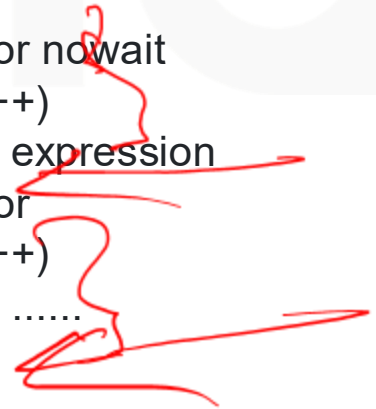
```
#pragma omp parallel
{
#pragma omp for
  for (int mytid=0; mytid<number_of_threads; mytid++)
    x[mytid] = some_calculation();
#pragma omp for
  for (int mytid=0; mytid<number_of_threads-1; mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

OpenMP: Synchronization

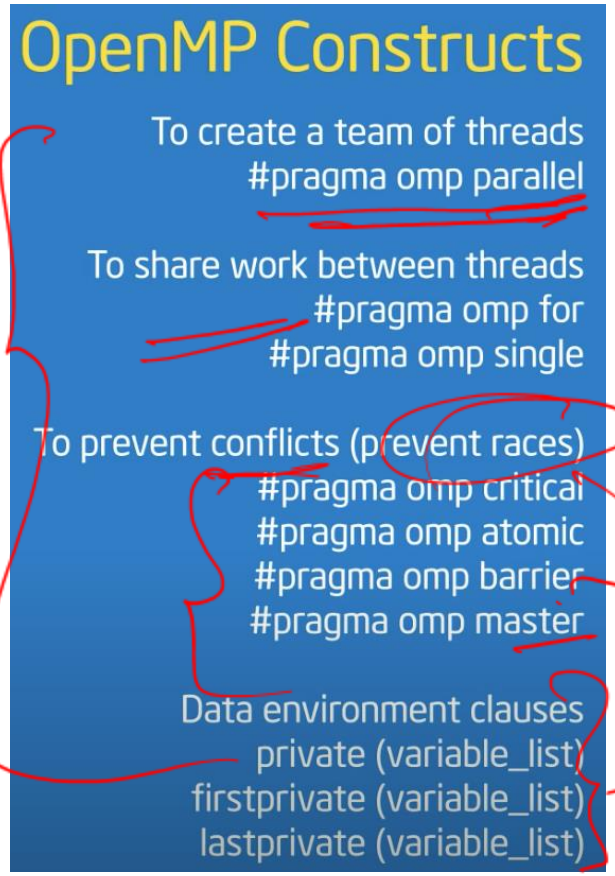
Implicit Barrier

- Implicit barrier behaviour can be cancelled with the clause nowait
- nowait clause implies that threads can start on the second loop while other threads are still working on the first

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
    b[i] = ..... a[i] .....
```



OpenMP

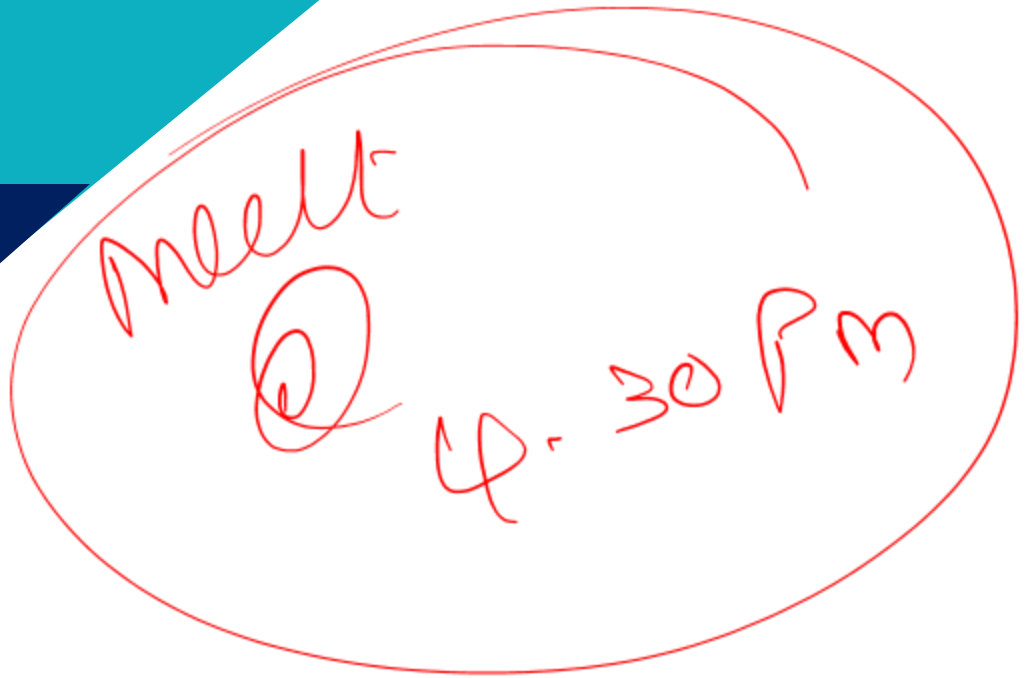


Parallel Processing and Multiprocessing in Python

https://wiki.python.org/moin/ParallelProcessing

Message Passing Interface (MPI)

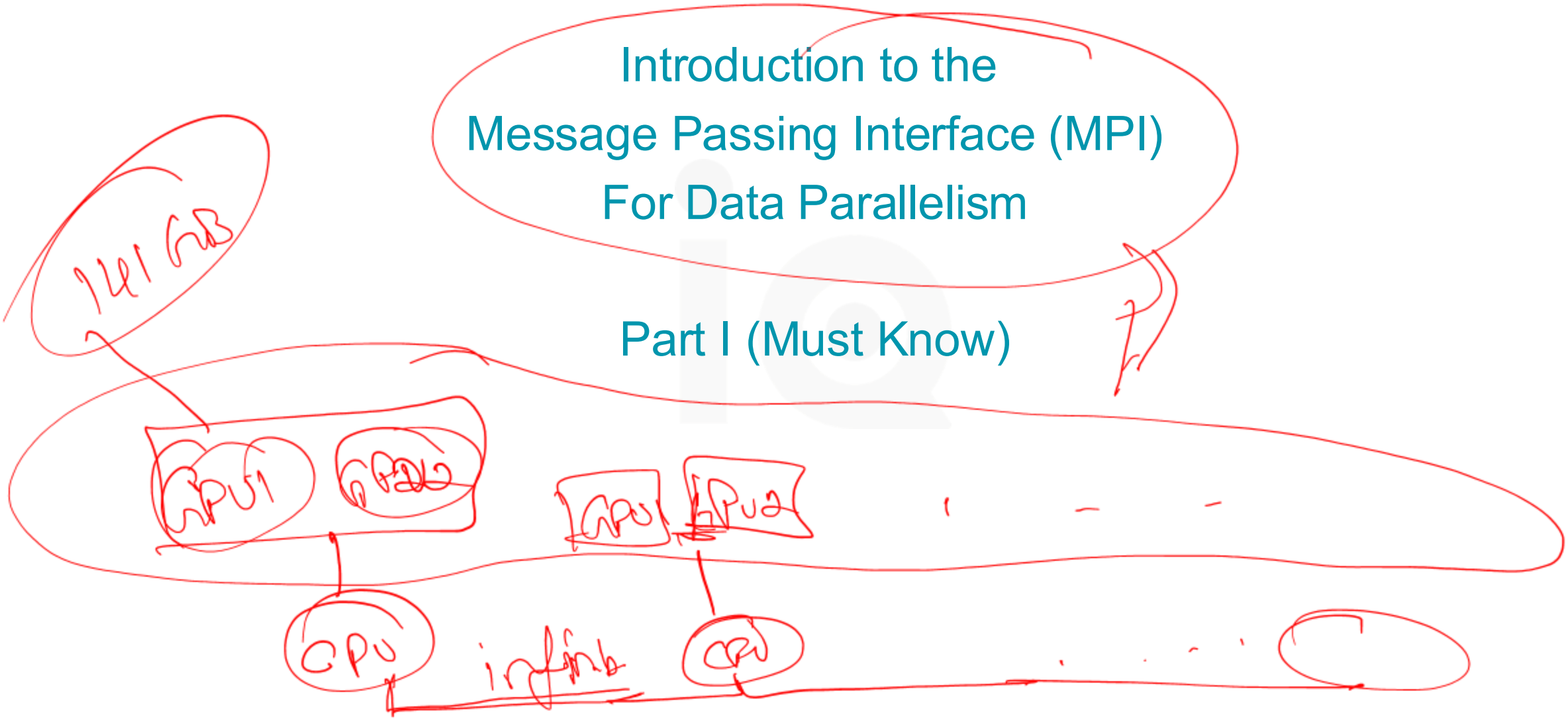
For Data Parallelism



MPI Implementations

Introduction to the
Message Passing Interface (MPI)
For Data Parallelism

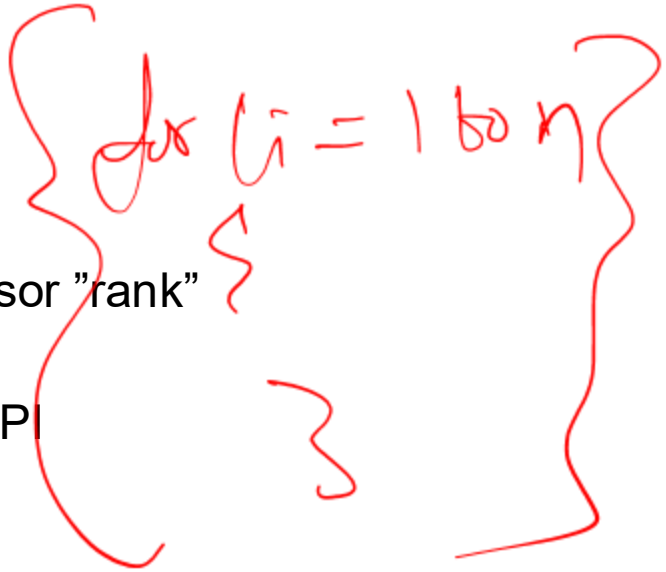
Part I (Must Know)

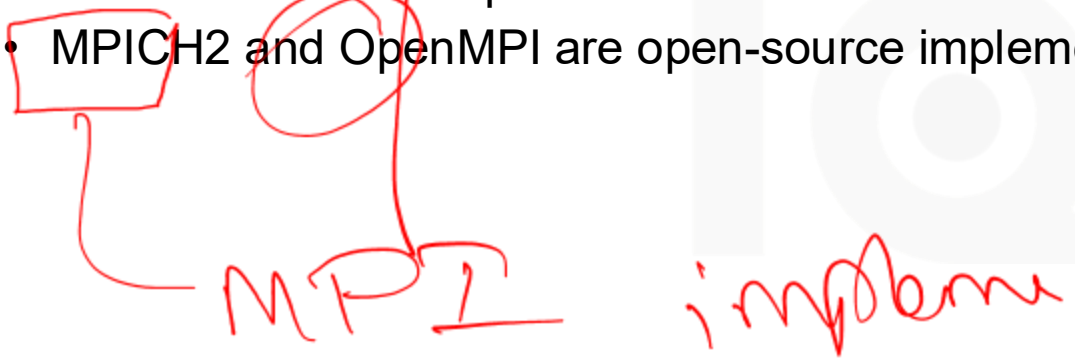


MPI Implementations

Overview

- each processor in a program runs the same/sub-program
  - written in a conventional sequential language
  - typically, the same on each processor (SPMD)
  - all work and data distribution are based on the value of processor ”rank”
  - communication via special send & receive routines
  - MPICH2 and OpenMPI are open-source implementations of MPI

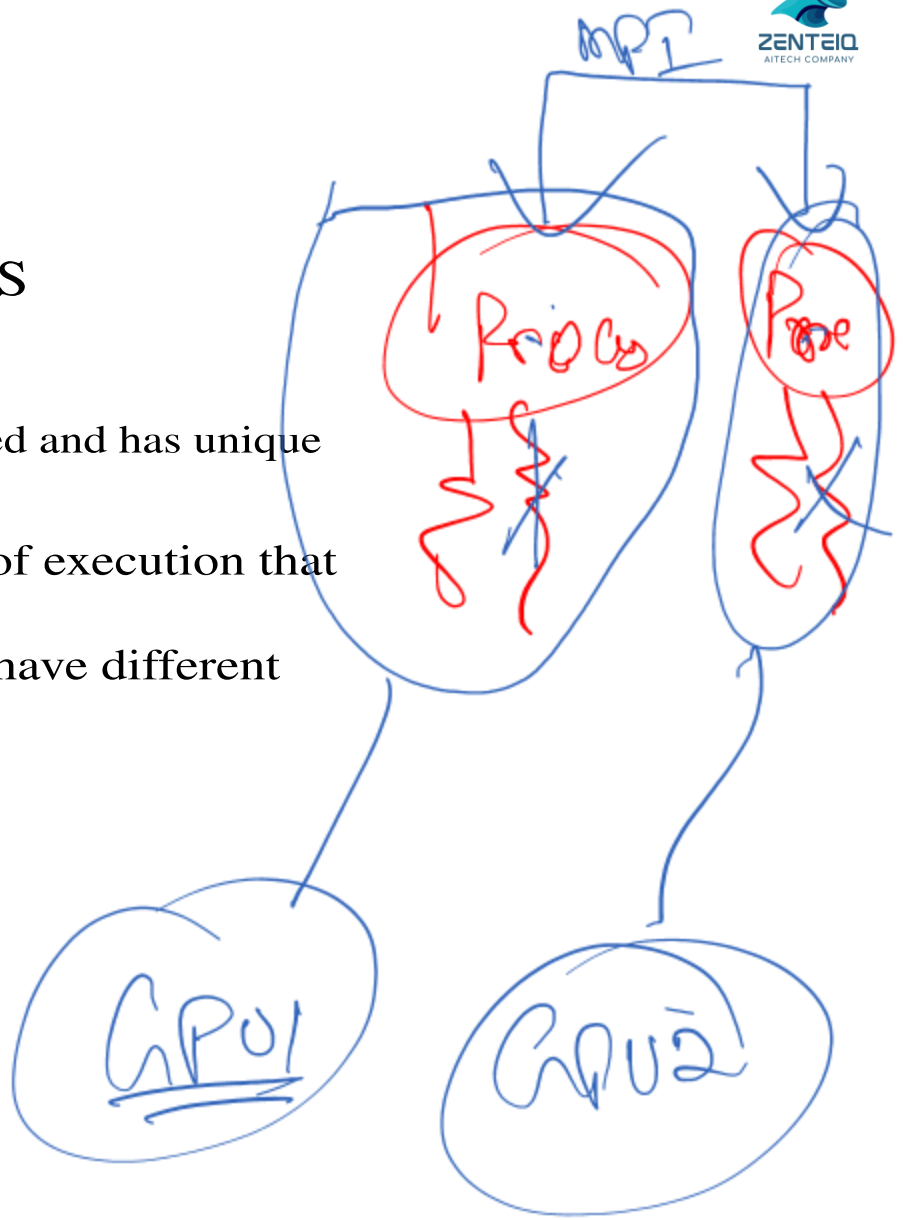




MPI Implementations

Message Passing Between Processes

- A process is an instance of a program that is being executed and has unique address space
  - a process may be made up of multiple threads of execution that shares a single address space
  - MPI communicates between processes, which have different address spaces
- Inter-process (MPI) communication consists of
  - synchronization of data
  - send and receive data between processes



MPI Implementations

MPI - How much do we need to know?

Functions in MPI

Fundamental functions in MPI

- MPI is small (6 functions)
- Simple parallel programs can be written with these functions

Advanced functions in MPI

- MPI is large (128 functions)
- Don't have to panic, as no need to master all to write an advance parallel program

Fundamental functions

MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv, MPI_Finalize

MPI Implementations

FUNDAMENTAL FUNCTIONS IN MPI

MPI_Init

initialize MPI processes

MPI_Comm_size

find the number of the processes in the MPI communicator

MPI_Comm_rank

the rank number within the pool of MPI communicator processes

MPI_Send

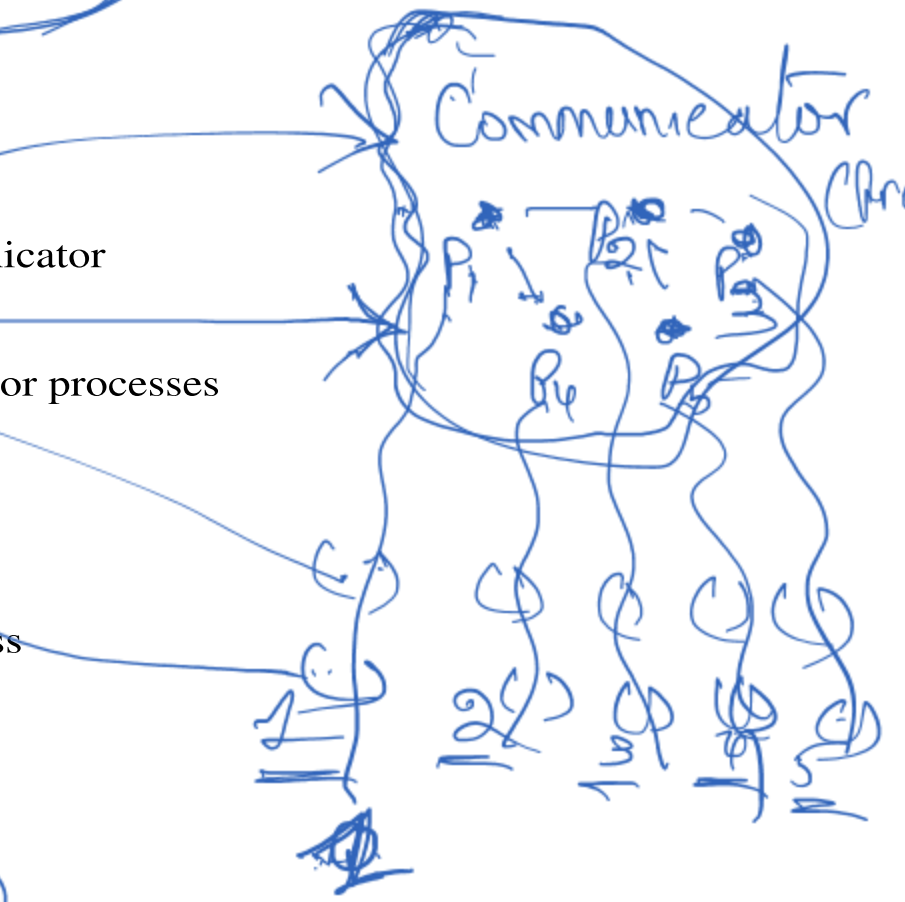
send a message from a process to another process

MPI_Recv

receive a message from another process to my process

MPI_Finalize

close down all MPI processes and prepare for exit



MPI Implementations

Hello World I

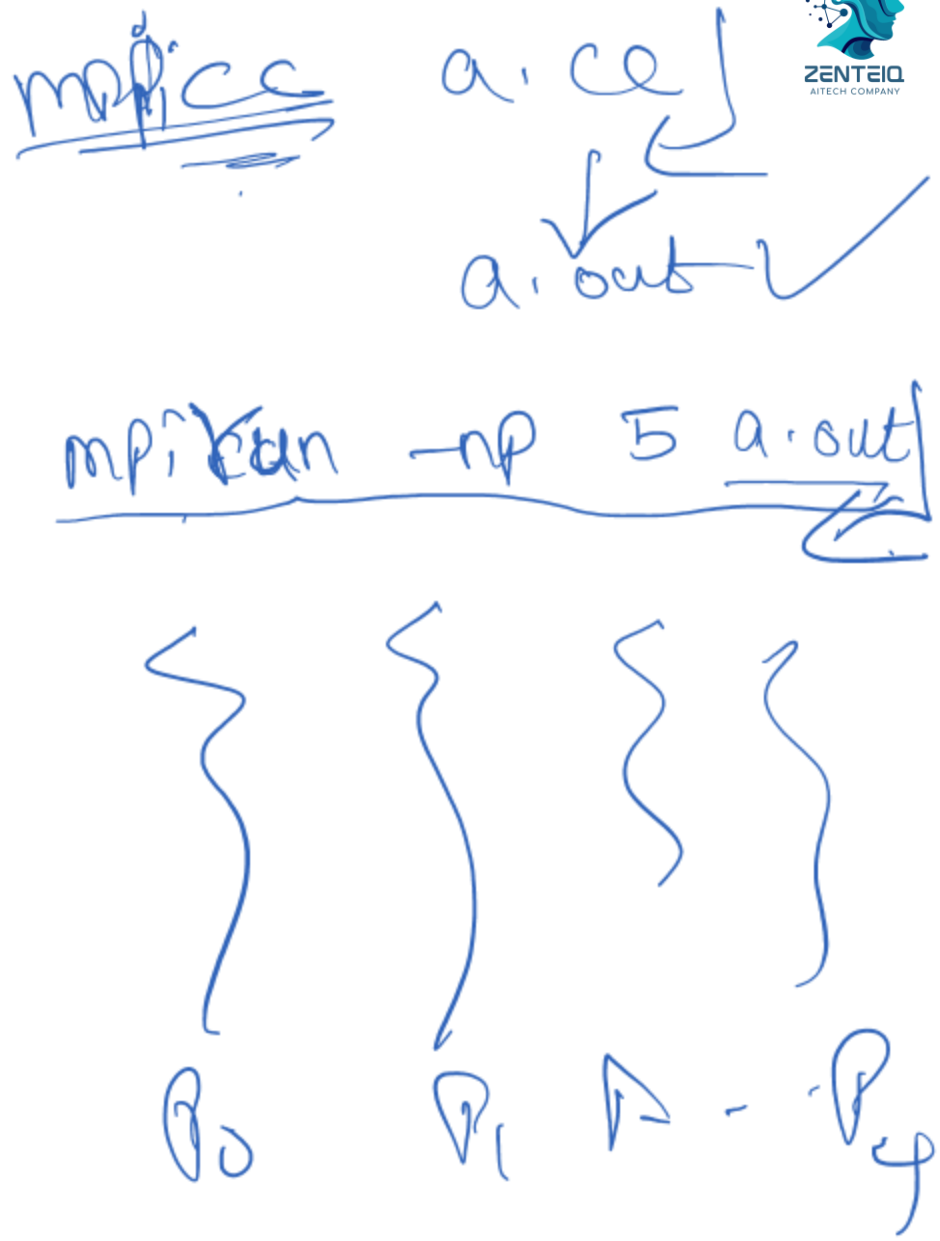
a.cc

```
#include <stdio.h>

int main(int argc, char ** argv)
{

printf("Hello World!\n");

}/* end main */
```



MPI Implementations

Hello World II

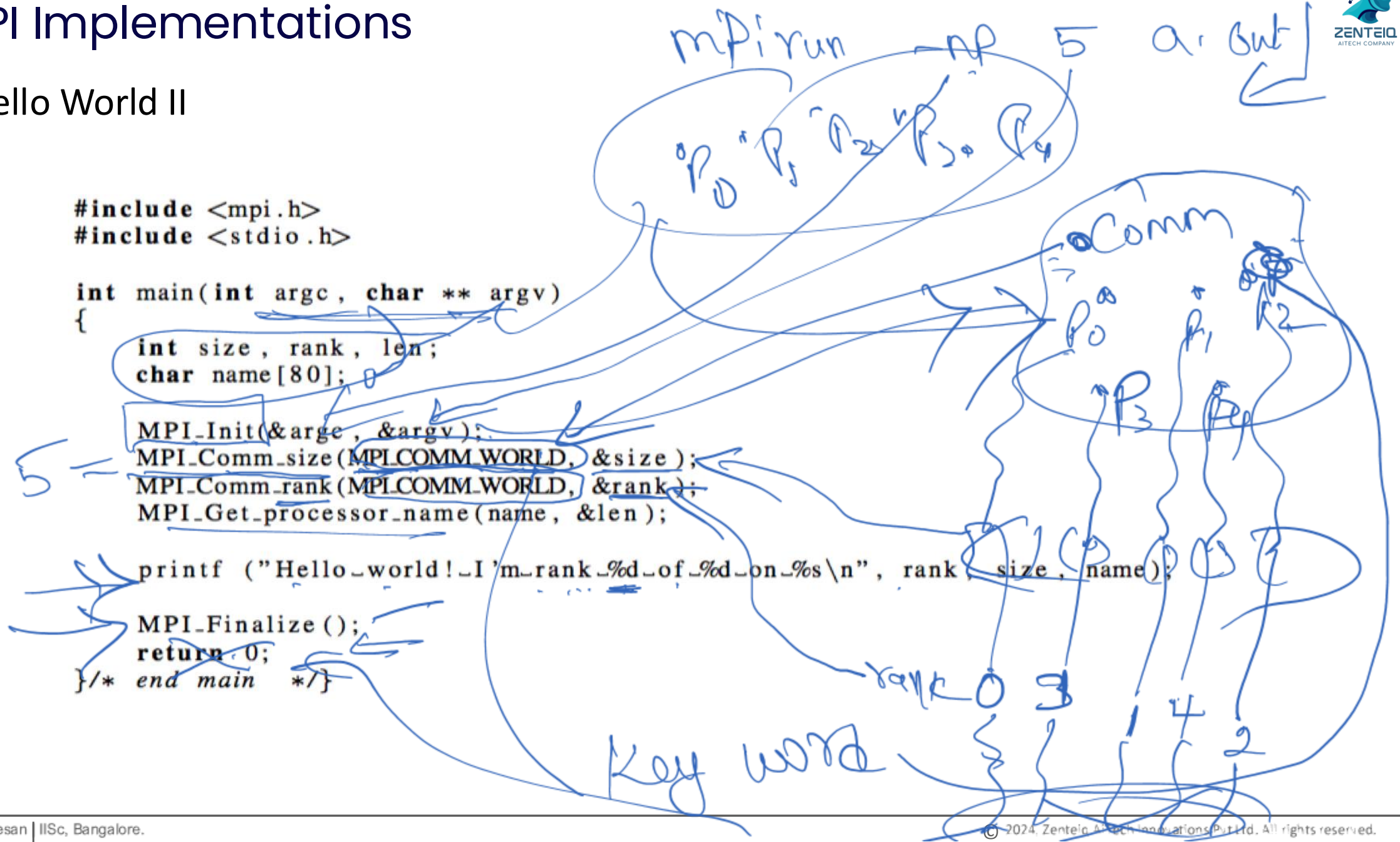
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int size, rank, len;
    char name[80];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);

    printf ("Hello world! I'm rank %d of %d on %s\n", rank, size, name);

    MPI_Finalize();
    return 0;
}/* end main */
```



MPI Implementations

MPI Communication

Hello World - III

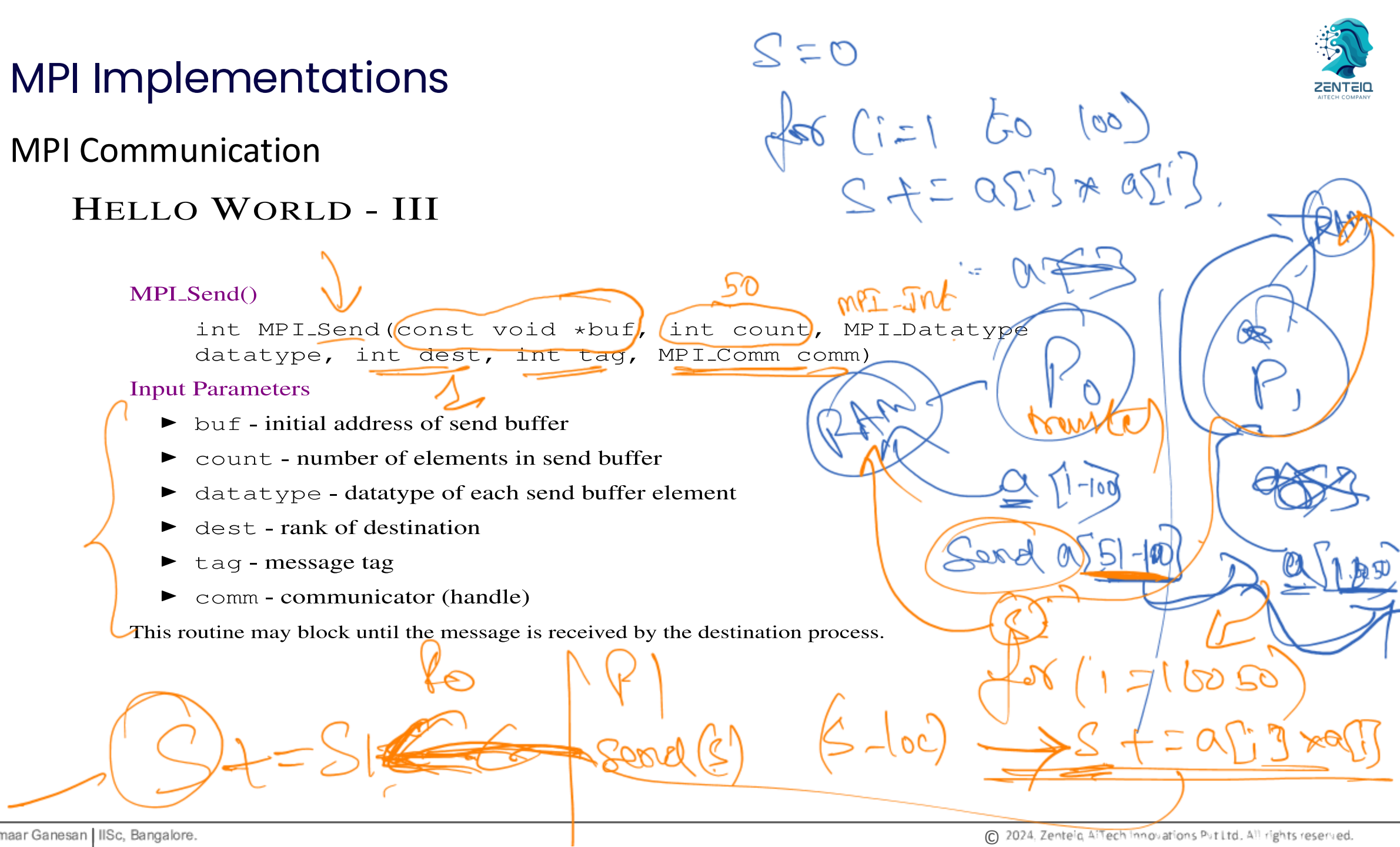
MPI_Send()

```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

Input Parameters

- `buf` - initial address of send buffer
- `count` - number of elements in send buffer
- `datatype` - datatype of each send buffer element
- `dest` - rank of destination
- `tag` - message tag
- `comm` - communicator (handle)

This routine may block until the message is received by the destination process.



MPI Implementations

MPI Communication

Hello World - III

MPI_Recv()

```
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Status *status)
```

Parameters

- `buf` - initial address of receive buffer
- `count` - maximum number of elements in receive buffer
- `datatype` - datatype of each receive buffer element
- `source` - rank of source
- `tag` - message tag
- `comm` - communicator (handle)

MPI Implementations

Hello World III

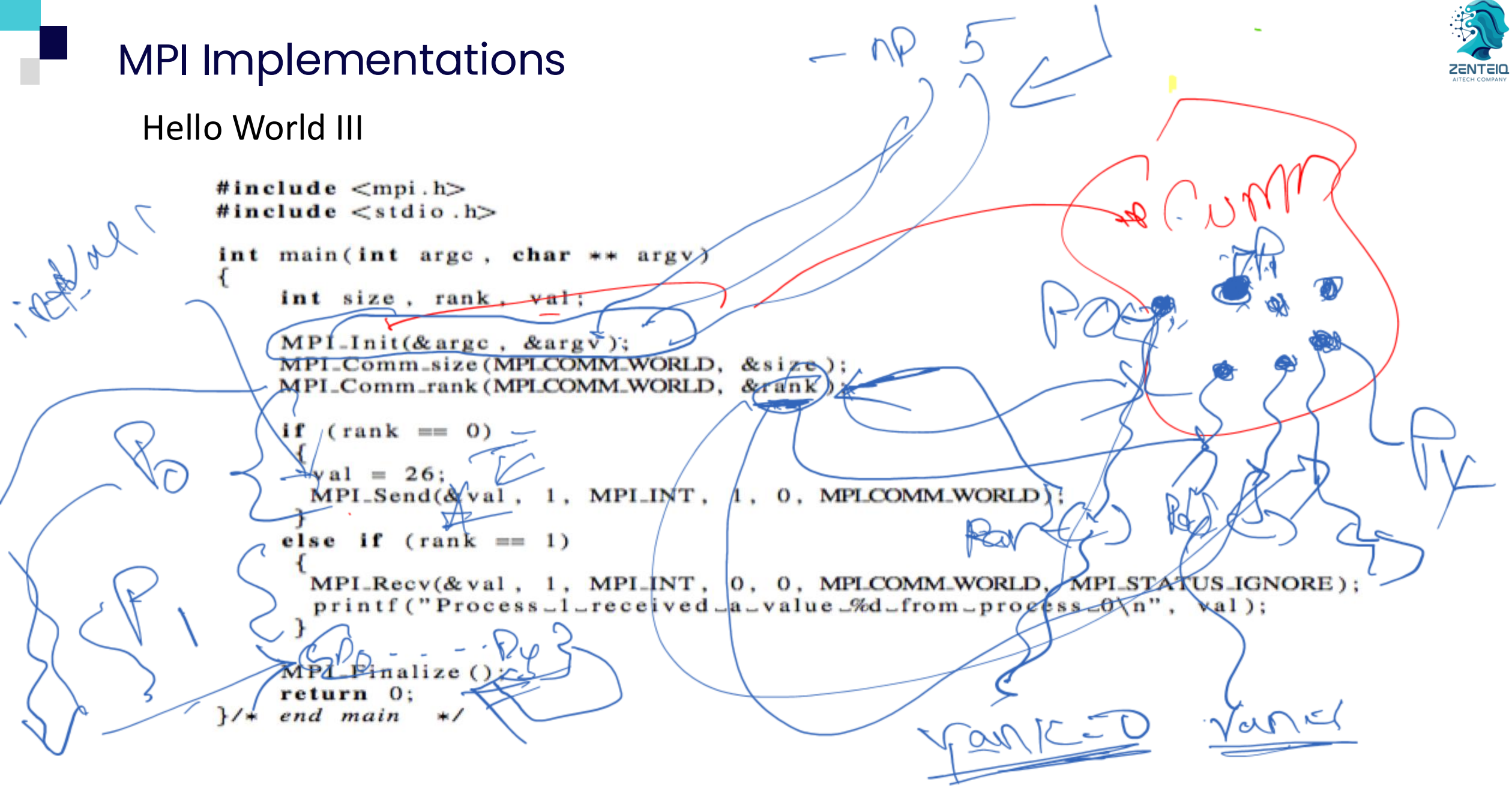
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int size, rank, val;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
    {
      val = 26;
      MPI_Send(&val, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == 1)
    {
      MPI_Recv(&val, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process 1 received a value %d from process 0\n", val);
    }

    MPI_Finalize();
    return 0;
}/* end main */
```



MPI Implementations

Hello World in Python

```
# install mpi4py
!pip install mpi4py
```

```
%%writefile helloworld.py
from mpi4py import MPI # Import mpi4py package
# Define a function
def main():
    '''A function to print the size and rank'''
    # creating the communicator
    comm = MPI.COMM_WORLD
    # Index of the process in the communicator
    rank = comm.Get_rank()
    # total number of processes in the communicator
    size = comm.Get_size()
    # Displaying the rank and size of a communicator
    print("Hello World: My rank is {} in the communicator of size {}".format(rank,size))

#call the function
main()
```

MPI Implementations

True/False

1. Each MPI process will have their own private memory
2. MPI programs can be executed on SMP machines
3. The order of MPI execution is based on the rank order (ID of MPI process)

Summary

Scalability in ML

- Scalability in ML refers to the ability to efficiently process increasing amounts of data and more complex algorithms without significant performance loss.

Challenges in Scaling ML Algorithms

- Major challenges include handling high-dimensional data, algorithm complexity, computational resource limitations, and maintaining performance efficiency with larger datasets

ML Libraries and its Parallel Capabilities

- ML libraries like TensorFlow and PyTorch are designed with parallel processing in mind to harness the full power of modern multi-core CPUs and GPUs

NUMBA and Dask

- Numba is used to optimize Python code for performance, while Dask provides advanced parallelization capabilities, particularly for scaling NumPy, pandas, and scikit-learn.

Effective scalability in ML is crucial for advancing the field, and despite challenges, progress is being facilitated by libraries that offer sophisticated parallel processing and optimization techniques.

Summary

Multiprocessing with OpenMP

- OpenMP provides a simple and flexible interface for developing parallel applications in C, C++, and Fortran on shared memory systems.
- It enables the parallel execution of code blocks by dividing tasks among multiple processors with the goal of reducing execution time.

Distributed Computing with MPI

- MPI stands for Message Passing Interface and is used for programming parallel computers on distributed memory systems.
- It allows various processors to communicate with each other by sending and receiving messages, which is essential for solving large-scale problems.

Summary

- Both OpenMP and MPI are vital in the realm of high-performance computing (HPC).
- OpenMP is suited for multi-threading in shared memory architectures, while MPI excels in scaling across multiple nodes in a distributed memory system.
- Together, they enable complex computational tasks to be performed more efficiently and are widely used in scientific computing, simulations, and ML model training at scale.