

OpenMP is a portable, multi-platform, shared-memory parallel programming interface. It is designed to be easy to use and to be efficient. It is a standard that defines a set of directives and library routines that can be used to write parallel programs in C, C++, and Fortran. It is a standard that defines a set of directives and library routines that can be used to write parallel programs in C, C++, and Fortran.

## Introduction to OpenMP

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It enables developers to write parallel applications easily and effectively with a minimalist code clarity.

### History and Evolution

OpenMP was developed by a consortium of industry and academic researchers. It was first released in 1997 and has since then evolved through several versions. The latest version is OpenMP 5.0, which was released in 2018.

### Definition of OpenMP

OpenMP is a portable, multi-platform, shared-memory parallel programming interface. It is designed to be easy to use and to be efficient. It is a standard that defines a set of directives and library routines that can be used to write parallel programs in C, C++, and Fortran. It is a standard that defines a set of directives and library routines that can be used to write parallel programs in C, C++, and Fortran.

### Types of OpenMP

OpenMP is divided into two main categories: **Shared Memory** and **Distributed Memory**. Shared Memory OpenMP is used for programs that run on a single machine with multiple processors. Distributed Memory OpenMP is used for programs that run on a cluster of machines, each with its own memory.

## Key Features of OpenMP

OpenMP provides a set of powerful directives that facilitate parallel programming, enabling developers to optimize performance through efficient resource utilization and simplified code structures.

### Directives Overview

OpenMP provides a set of directives that can be used to parallelize code. These directives are used to specify the number of processors to be used, the scope of the parallel region, and the synchronization points. The directives are used to parallelize code in a portable way, so that the code can be run on different hardware configurations.

### Synchronization Mechanisms

OpenMP provides mechanisms for synchronizing parallel threads. These mechanisms include barriers, locks, and atomic operations. Barriers are used to ensure that all threads reach a certain point in the code before proceeding. Locks are used to ensure that only one thread can access a shared resource at a time. Atomic operations are used to ensure that a single operation is performed atomically.

### Parallel Regions

OpenMP provides a way to define parallel regions in code. A parallel region is a section of code that is executed in parallel by multiple threads. The parallel region is defined by the `#pragma omp parallel` directive. The threads within the parallel region can be synchronized using the `wait` directive.

### Work-sharing Constructs

OpenMP provides work-sharing constructs that allow a single thread to dynamically acquire more work as needed. These constructs include `for` loops, `do` loops, and `sections`. These constructs are used to parallelize loops and sections of code, allowing the work to be distributed among multiple threads.

### Follow-up

OpenMP is a powerful tool for parallel programming. It provides a set of directives and mechanisms that make it easy to write parallel code. OpenMP is supported by many compilers and is widely used in scientific and engineering applications.

# Understanding OpenMP Directives

## THANK YOU

## ANY QUESTIONS?

### Conclusion

OpenMP is a powerful tool for parallel programming. It provides a set of directives and mechanisms that make it easy to write parallel code. OpenMP is supported by many compilers and is widely used in scientific and engineering applications.

## A Comprehensive Guide to Parallel Programming in C/C++

## Basic Working of OpenMP

### Thread Management

OpenMP provides a way to manage threads in a parallel program. The `omp_set_num_threads` function is used to set the number of threads to be used. The `omp_get_num_threads` function is used to get the number of threads that are currently being used. The `omp_get_thread_num` function is used to get the thread number of the current thread.

### Memory Management

OpenMP provides mechanisms for managing memory in a parallel program. The `omp_setenv` function is used to set environment variables. The `omp_getenv` function is used to get environment variables. The `omp_malloc` function is used to allocate memory that is shared among all threads.

### Compilation and Execution

OpenMP programs are compiled using a compiler that supports OpenMP. The compiler uses the OpenMP directives to generate code that can be executed in parallel. The execution of the program is controlled by the OpenMP runtime library.

### Working Example

```

#include <omp.h>
#include <stdio.h>

int main() {
    omp_set_num_threads(4);
    printf("Number of threads: %d\n", omp_get_num_threads());
    return 0;
}
    
```

### Portability

OpenMP is a portable standard that can be used on a wide variety of hardware and software configurations. It is supported by many compilers and is widely used in scientific and engineering applications.

### Ease of Use

OpenMP is designed to be easy to use. It provides a set of directives and mechanisms that make it easy to write parallel code. OpenMP is supported by many compilers and is widely used in scientific and engineering applications.

### Limitations

OpenMP has some limitations. It is not suitable for programs that require fine-grained control over the execution of parallel threads. It is also not suitable for programs that require a high degree of portability.

### Advantages and Disadvantages of OpenMP

Forming the benefits and limitations of OpenMP reveals its practical applications and challenges in the realm of parallel programming.

### Comparison with Other Parallel Programming Models

OpenMP is compared with other parallel programming models, such as MPI and CUDA. OpenMP is found to be more suitable for shared-memory systems, while MPI is more suitable for distributed-memory systems.

### Performance Evaluation

OpenMP is evaluated using various performance metrics, such as speedup and efficiency. It is found that OpenMP performs well on a wide range of benchmarks.

# Team members:

---

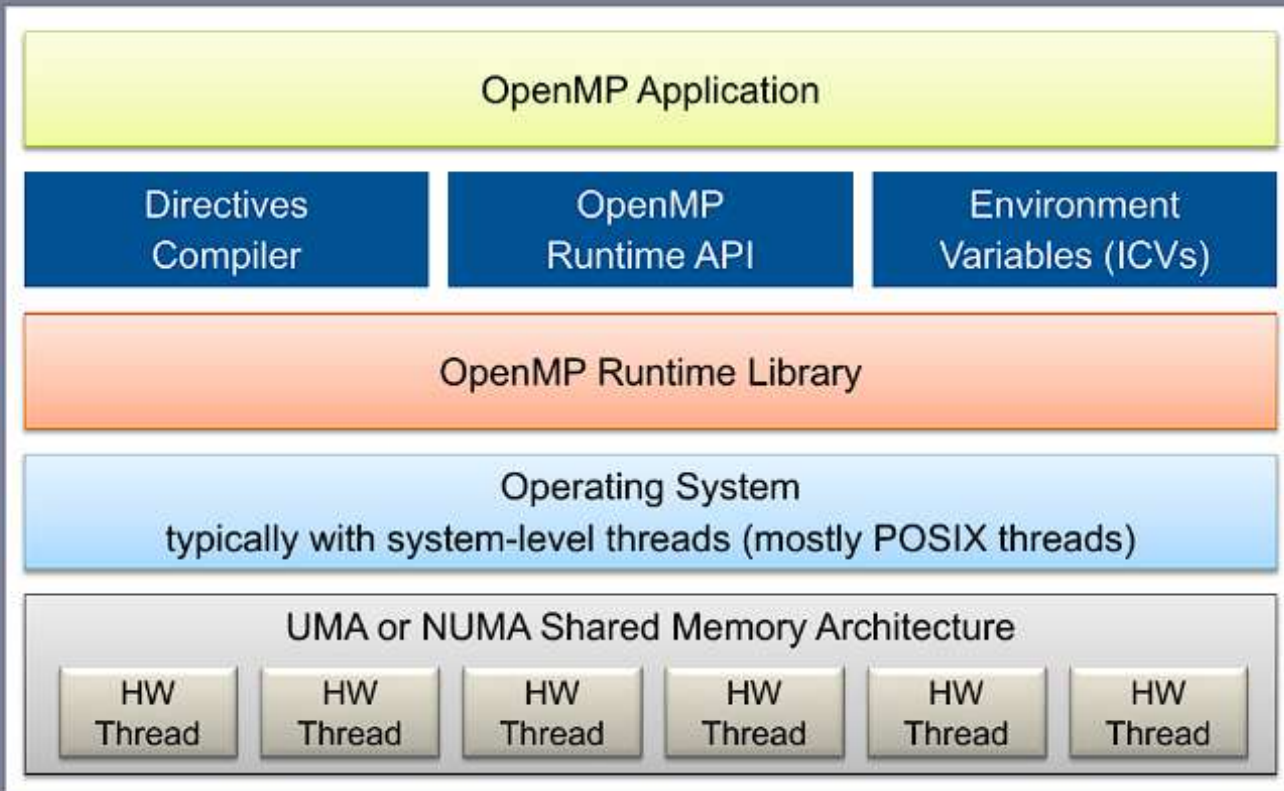
- 1.Kandula Rupa Srilekha – 21MIC0044
- 2.Mallidi Sasi Kiran Reddy – 21MIC0046
- 3.Garikapati Hema Sandeep – 21MIC0062
- 4.Beegala Malyadri Pavan Ganesh – 21MIC0065

# Introduction to



OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It enables developers to write parallel applications easily and effectively while maintaining code clarity.

# Definition of OpenMP



OpenMP, or Open Multi-Processing, is an application programming interface (API) that provides a portable, scalable model for shared memory parallel programming across various platforms. It allows developers to write parallel code more intuitively using compiler directives, library routines, and environment variables.



# Follow up

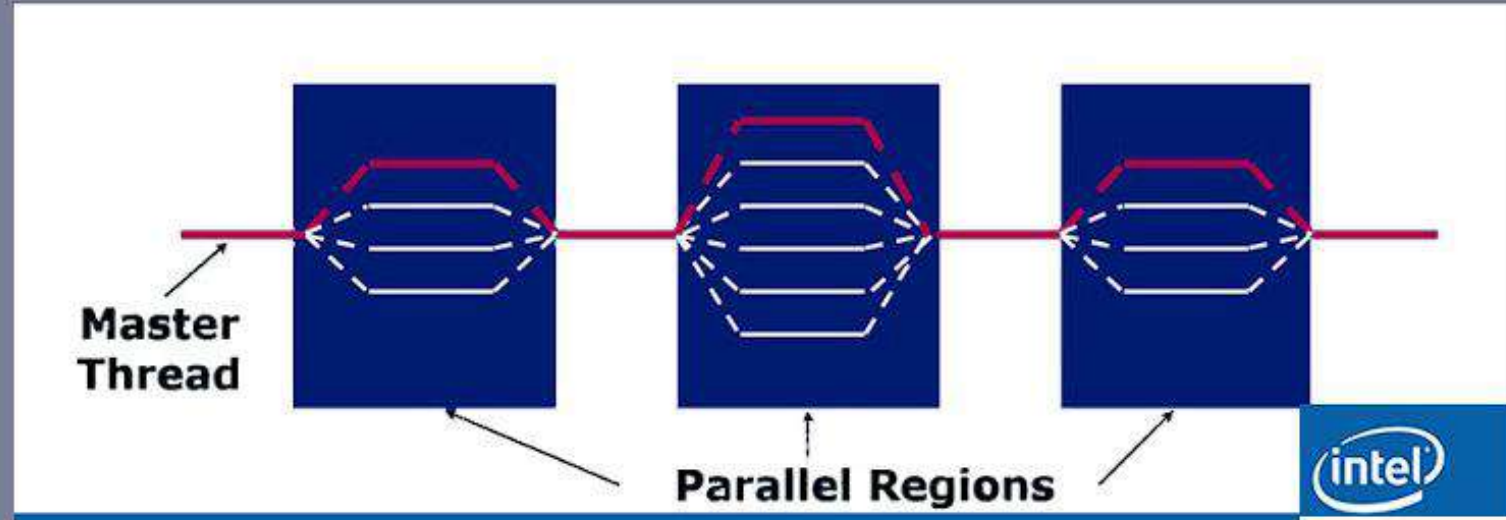
The primary purpose of OpenMP is to simplify the parallel programming process by allowing developers to easily add parallelism to existing serial code. It is designed to support incremental parallel programming, enabling programs to be optimized step by step for performance enhancement.

It uses compiler directives, library routines, and environment variables to manage parallelism efficiently.

OpenMP supports key constructs like parallel, for, sections, and critical for flexible control.

This makes it ideal for shared memory systems and multi-core processors.

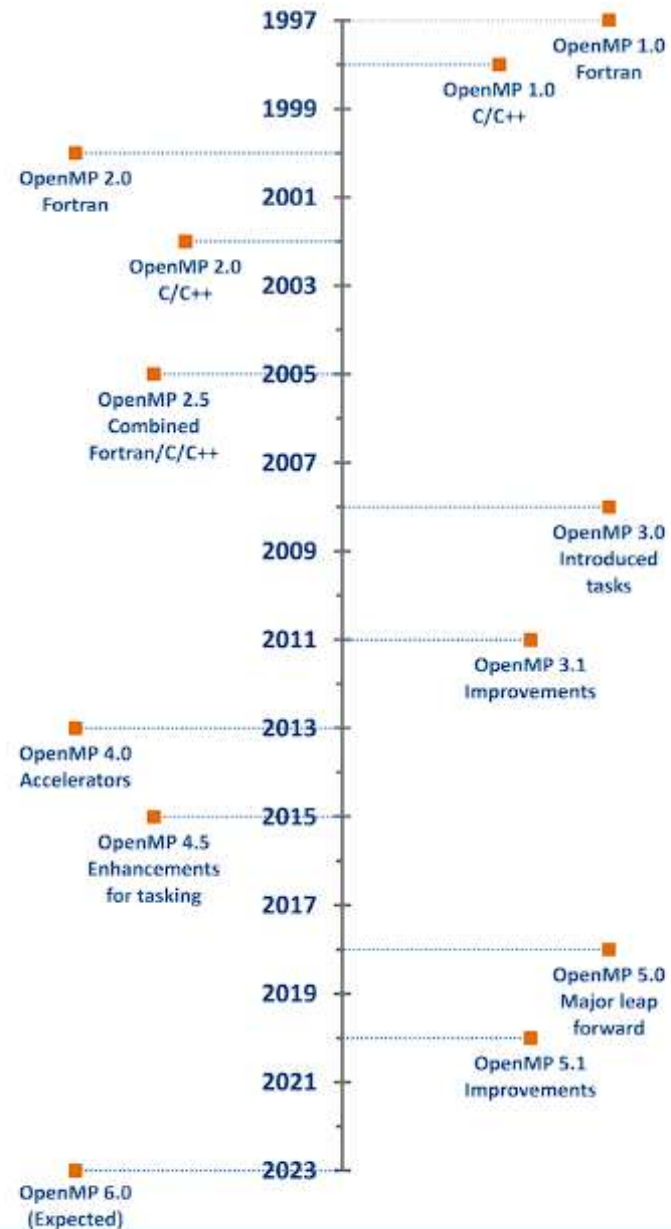
Overall, OpenMP enhances performance while keeping development simple and scalable.



# History and Evolution

OpenMP was first introduced in 1997 and has undergone significant evolution with subsequent versions enhancing its capabilities. Over the years, it has become a standard for shared memory parallel programming in the computing industry by enabling better performance on multi-core processors.

OpenMP Version Timeline



# Follow up

OpenMP is applied in various computational fields, including scientific computing, data analysis, and machine learning. Its compatibility with different programming languages allows it to be used across diverse systems, facilitating performance improvements in applications that require extensive calculations.

OpenMP Version Timeline



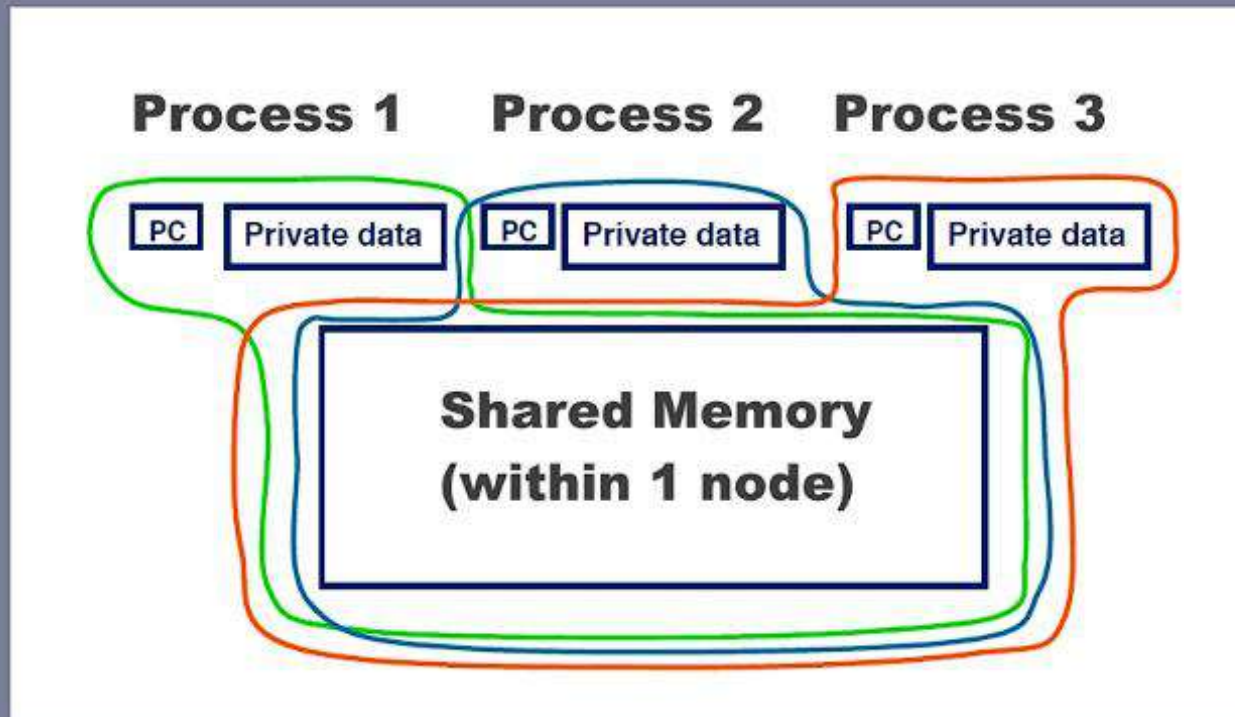


# Key Features of OpenMP

OpenMP provides a set of powerful directives that facilitate parallel programming, enabling developers to optimize performance through efficient resource utilization and simplified code structures.

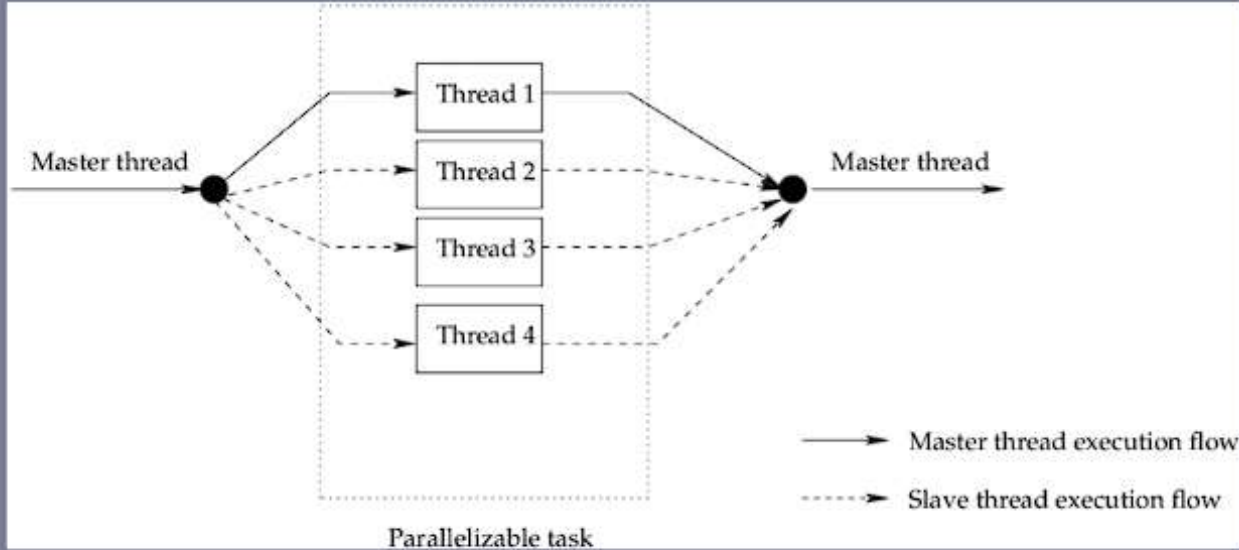


# Directives Overview



OpenMP employs compiler directives to instruct the compiler to parallelize specific sections of code. This approach allows developers to add parallelism incrementally without rewriting entire applications.

# Parallel Regions



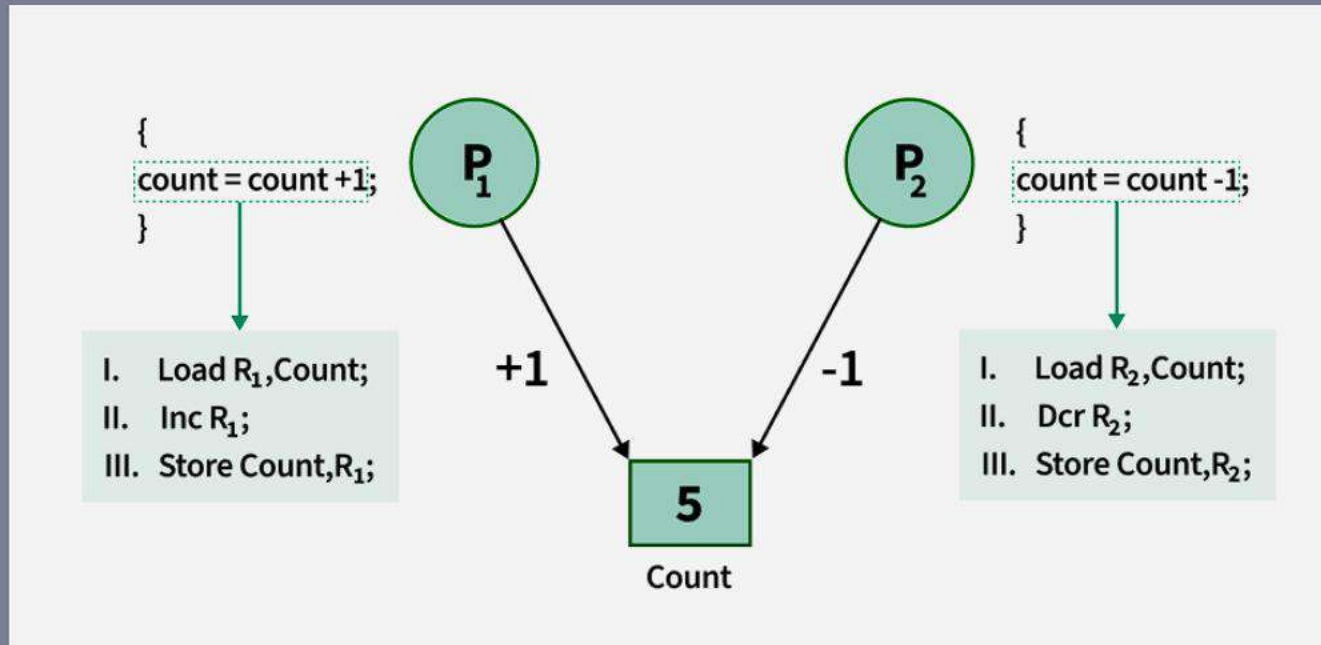
Parallel regions in OpenMP define code segments that can be executed by multiple threads simultaneously. The directive `#pragma omp parallel` initiates these sections, significantly enhancing performance in multi-core environments.

# Work-sharing Constructs

Work-sharing constructs like ``for``, ``sections``, and ``single`` divide tasks among threads efficiently. These constructs optimize workload distribution, ensuring that work is balanced to minimize idle time among threads.

# Synchronization Mechanisms

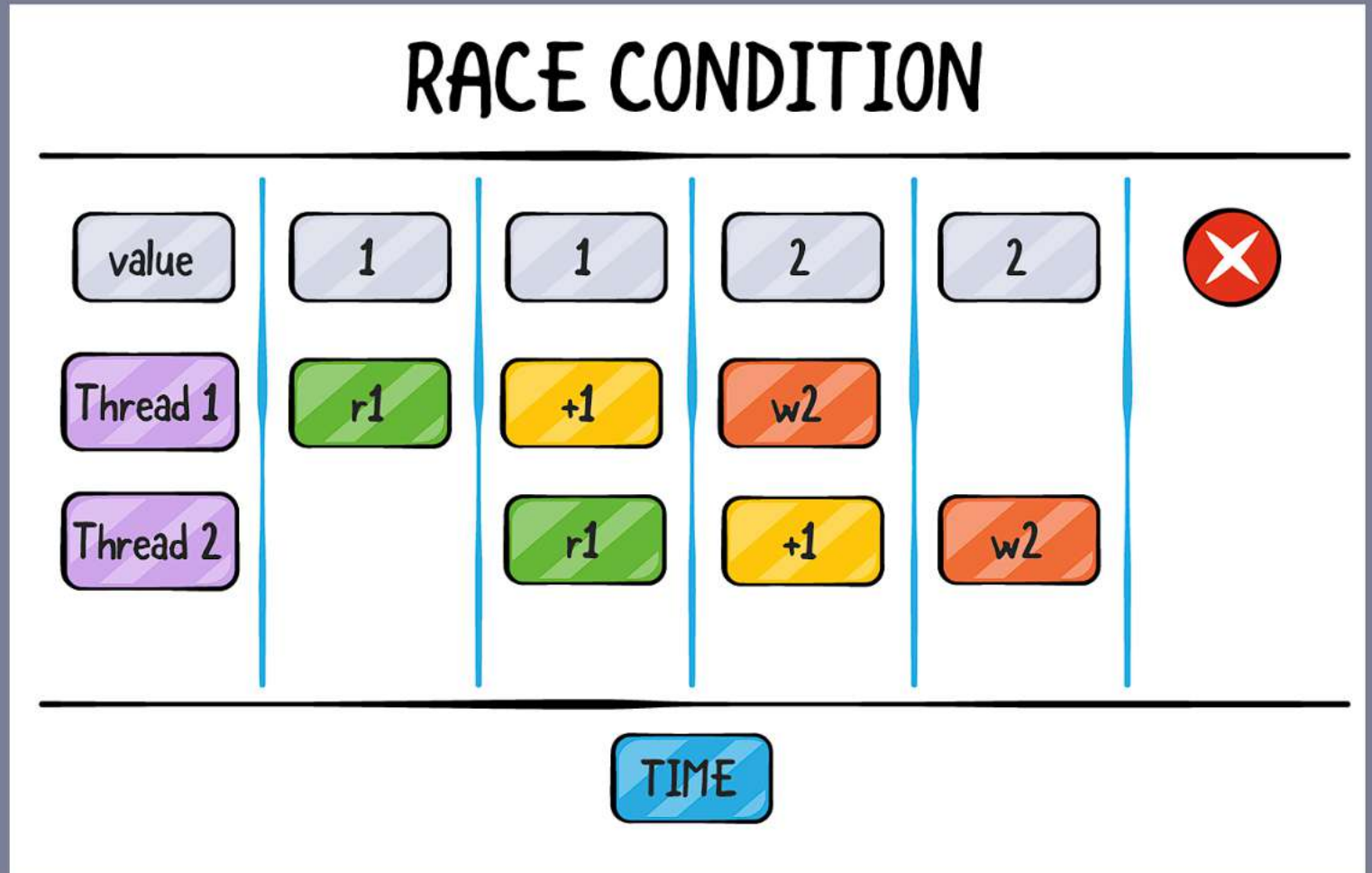
OpenMP provides synchronization mechanisms such as `critical` and `atomic` to manage access to shared resources. These mechanisms ensure data integrity and prevent race conditions when threads interact.





# Follow up

OpenMP utilizes environment variables to control runtime behavior. Variables such as `OMP\_NUM\_THREADS` can specify the number of threads, while `OMP\_SCHEDULE` defines scheduling policies for dynamic work assignment.

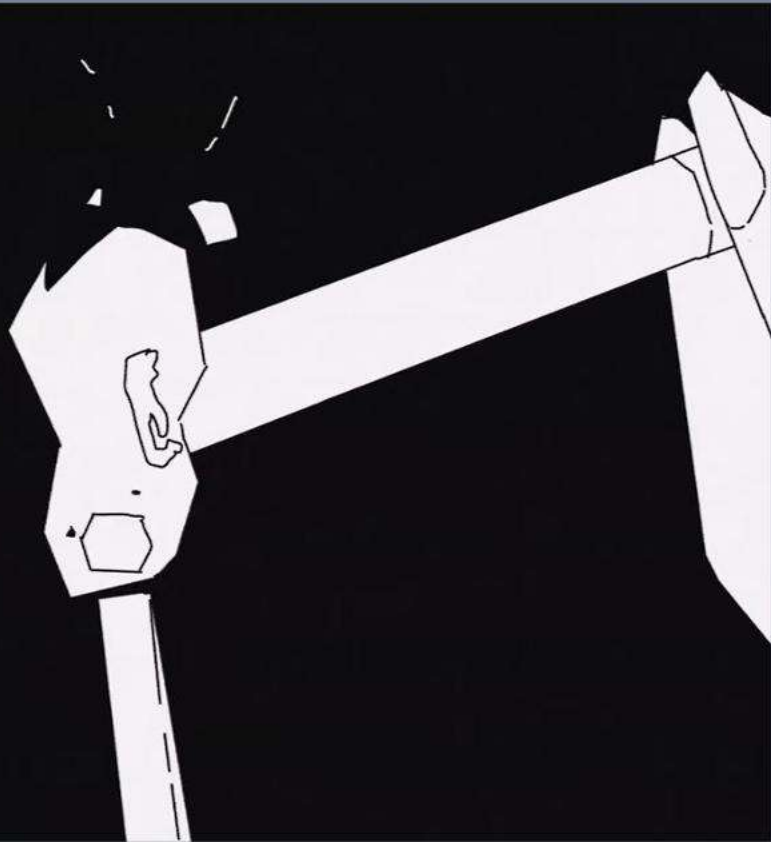


# Advantages and Disadvantages of OpenMP

Examining the benefits and limitations of OpenMP reveals its practical applications and challenges in the realm of parallel programming.

# Ease of Use

OpenMP's straightforward syntax allows developers to parallelize existing code by simply adding directives. This simplicity accelerates the learning curve for users familiar with shared memory programming without a deep dive into complex APIs.



# Portability

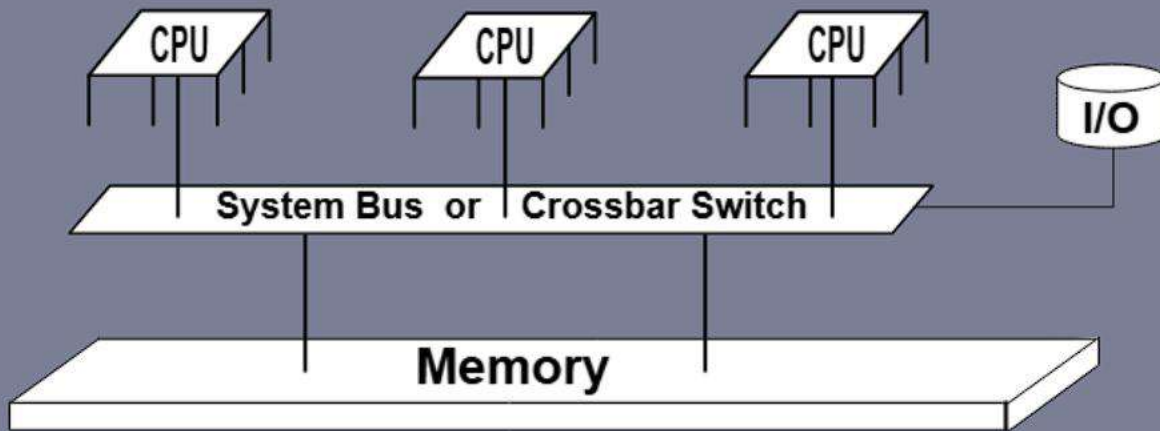
OpenMP is designed to be platform-independent, providing a consistent model across various operating systems and compilers. This allows code to be developed on one system and easily executed on different architectures without significant modifications.





# Follow up

OpenMP supports incremental parallelism, enabling developers to gradually convert sequential code into parallel code. This approach facilitates the testing and optimization of parallel sections without a complete rewrite, ensuring stability during transitions.

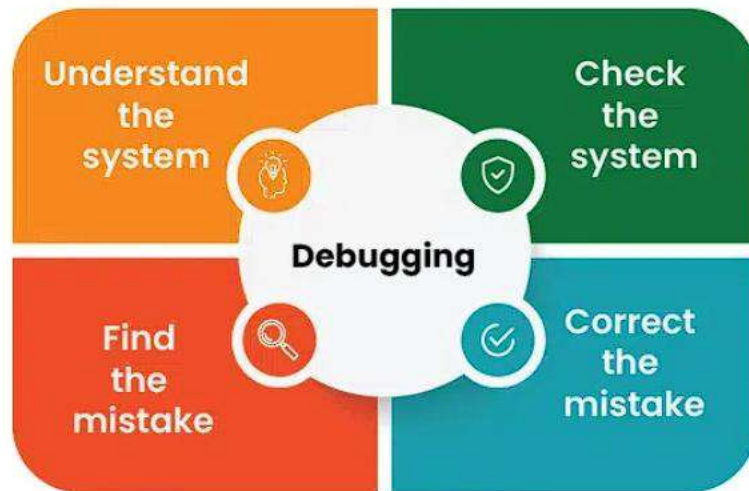


## Limited to Shared Memory

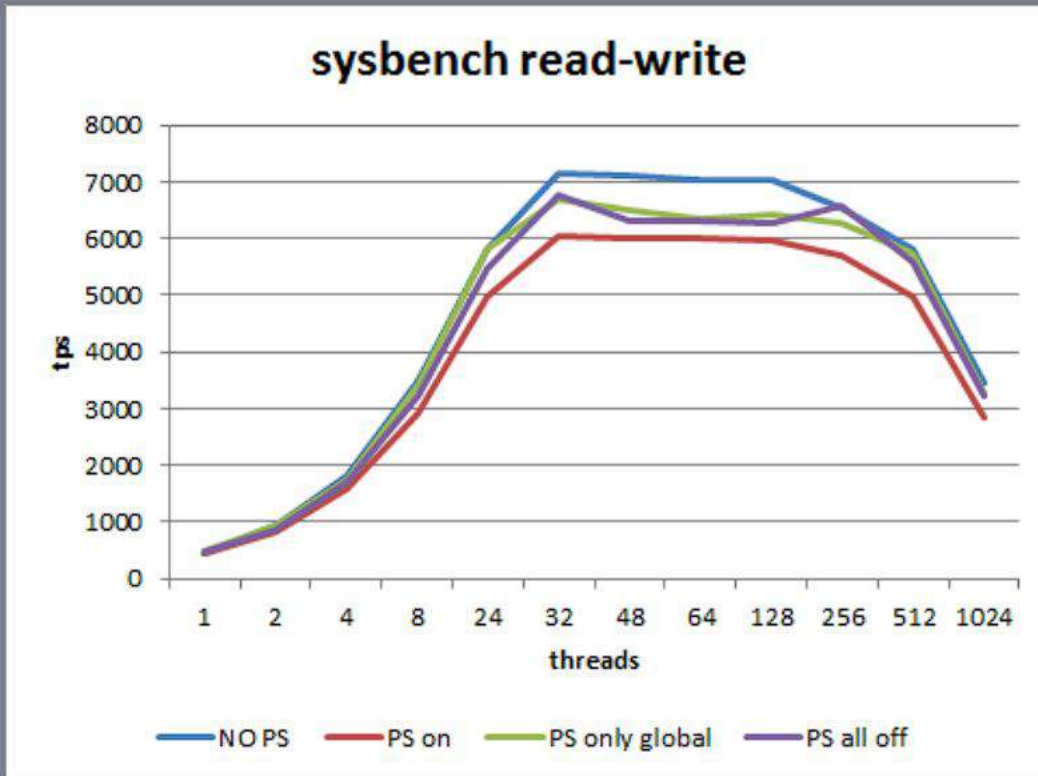
One significant limitation of OpenMP is its confinement to shared memory architectures, restricting its use in distributed memory systems. This greatly limits its applicability in large-scale computing environments often utilizing clusters or supercomputers.

# Complexity in Debugging

Debugging OpenMP programs can be complex due to the nondeterministic nature of thread execution. This often leads to challenges in reproducing errors and may require specialized debugging tools tailored for concurrent environments.



# Performance Overhead



Although OpenMP can significantly improve performance, it also introduces overhead associated with thread management and synchronization. Careful planning is needed to avoid diminishing returns where the overhead outpaces the performance gains from parallelism.



# Basic Working of OpenMP

# Compilation and Execution



OpenMP programs are compiled with standard compilers that support OpenMP, such as GCC or Intel Compiler. After compilation, the execution involves the runtime system managing threads for parallel tasks based on the directives specified in the code.

# Writing OpenMP Code

OpenMP code is written in C/C++ by including the OpenMP header `#include <omp.h>`. The `#pragma omp` directive is used to indicate parallel regions and manage task execution, allowing developers to easily convert sequential code into parallel code.

```
1 // OpenMP header
2 #include <omp.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main()
7 {
8     int nthreads, tid;
9
10    // Begin of parallel region
11    #pragma omp parallel private(nthreads, tid)
12    {
13        // Getting thread number
14        tid = omp_get_thread_num();
15        printf("Welcome Guys from thread = %d\n",tid);
16
17        if (tid == 0) {
18
19            // Only master thread does this
20            nthreads = omp_get_num_threads();
21            printf("Number of threads = %d\n",nthreads);
22        }
23    }
24 }
```

# Using Compiler Directives

Compiler directives control parallel execution in OpenMP code. These directives provide guidance on how to split tasks among threads, define parallel regions, and synchronize data, enabling scalable parallel solutions while maintaining simplicity in code structure.

```
{  
    #pragma omp for nowait  
    for ( k = 0; k < m; k++ ) {  
        fn10(k); fn20(k);  
    }  
  
    #pragma omp sections private(y, z)  
    {  
        #pragma omp section  
        { y = sectionD(); fn70(y); }  
        #pragma omp section  
        { z = sectionC(); fn80(z); }  
    }  
}
```



# Thread Management

OpenMP abstracts thread management to simplify parallel programming. The runtime library efficiently manages thread creation, scheduling, and termination based on the defined parallel regions, enhancing program scalability without extensive multi-threading expertise.

## Parallel Sections

```
#pragma omp parallel shared(A,B)private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for(i=0; i<n; i++)
            A[i]= A[i]*A[i]- 4.;
        #pragma omp section
        for(i=0; i<n; i++)
            B[i]= B[i]*B[i] + 9.;
    } // end omp sections
} // end omp parallel
```

## Example Code Snippet

A basic OpenMP parallel code snippet could be: `#pragma omp parallel for`, which distributes loop iterations across threads. This simple line conveys the intention to parallelize operations, embodying OpenMP's premise of ease and efficiency in code transformation.

# Conclusion

- OpenMP simplifies parallel programming on shared memory systems using easy-to-apply compiler directives.
- It enables efficient thread management and synchronization with minimal code changes.
- Common directives like parallel, for, and critical improve performance in multi-core systems.
- Overall, OpenMP is a practical tool for accelerating compute-intensive applications.

**ANY  
QUESTIONS?**



**VIT**

Vellore Institute of Technology  
Knowledge • Energy • Action



# THANK YOU

