High Performance Computing(CCA3007)

OPENMP IN HIGH PERFORMANCE COMPUTING

GROUP MEMBERS:
ANURAG PRASAD 21BSA10075
MAYANK CHOUBEY 21BSA10113
VANSH THAKUR 21BSA10133

FACULTY:
DR. AJEET SINGH
Assistant Professor
VIT Bhopal University

Introduction to OpenMP

- OpenMP is an Application Program Interface (API) for explicit, portable, shared-memory parallel programming in C/C++ and Fortran.
- It provides a set of compiler directives, runtime library routines, and environment variables that allow developers to parallelize their code and leverage the multiple cores available in modern CPUs.
- OpenMP was first introduced in 1997 and is currently at version 5.0 (released in 2018).

Brief History of OpenMP

- In 1991, the Parallel Computing Forum (PCF) group invented a set of directives for specifying loop parallelism in Fortran programs.
- X3H5, an ANSI subcommittee, developed an ANSI standard based on the PCF directives.
- In 1997, the first version of OpenMP for Fortran was defined by the OpenMP Architecture Review Board.
- The binding for C/C++ was introduced later.

Key Features of OpenMP

- **Shared Memory Programming Model**: All threads have access to the same shared memory, simplifying data sharing and communication.

- **Compiler Directives**: OpenMP provides a set of compiler directives (e.g., #pragma omp) for defining parallel regions, work sharing, and synchronization in C/C++ and Fortran programs.

- **Runtime Library Routines**: OpenMP includes runtime library routines for managing threads and querying the execution environment.

- **Portable:** OpenMP is standardized for shared memory architectures, making it portable across different systems.

How to Use OpenMP?

1. **Include the OpenMP Header File:**
   - In C/C++, include the `<omp.h>` header file to access the OpenMP API.
   - In Fortran, use the `USE OMP_LIB` statement to access the OpenMP functionality.

2. **Specify Parallel Regions:**
   - Use the `#pragma omp parallel` directive in C/C++ to define a parallel region.
   - In Fortran, use the `!$OMP PARALLEL` and `!$OMP END PARALLEL` directives to mark the parallel region.
   - The code within the parallel region will be executed concurrently by multiple threads.

3. **Set the Number of Threads:**
   - Use the `OMP_NUM_THREADS` environment variable to specify the number of threads to be used.
   - This can be set either in the command line or within the job script before running the program.

How to Use OpenMP?

4. **Compile with OpenMP Support:**

   - For GCC, use the `-fopenmp` flag when compiling the code.

   - For Intel compilers, use the `-qopenmp` flag.

5. **Run the Parallel Program:**

   - Submit the compiled OpenMP program as a job to the cluster, ensuring that the `OMP_NUM_THREADS` environment variable is set correctly.

   - The program will execute with the specified number of threads, and the output will show the parallel execution.

6. **Use OpenMP Constructs:**

   - Leverage OpenMP constructs like `#pragma omp for` for parallel loops, `#pragma omp sections` for parallel sections, and `#pragma omp critical` for critical regions.

   - Utilize OpenMP runtime library functions like `omp_get_thread_num()` to obtain the thread ID.

How to Use OpenMP?

7. **Handle Limitations:**

- Ensure that the loop iterations and control expressions are suitable for parallelization with OpenMP.

- Avoid early exits from parallel loops, as they can prevent proper parallelization.

8. **Use OpenMP in Google Colab:**

- Follow the same steps, but include the OpenMP header, set the number of threads, and compile with the appropriate flags in the Colab notebook.

OpenMP Directives

- OpenMP directives are compiler directives used to provide hints to the compiler for parallelizing sections of code.
- They are preceded by #pragma omp in C/C++ or !$omp in Fortran.
- These directives guide the compiler on how to parallelize the code and how to manage data sharing and synchronization among threads.
- Here's an explanation of some common OpenMP directives:

OpenMP Directives

parallel:

Creates a team of threads that execute the enclosed code block in parallel.

Syntax:

```
#pragma omp parallel [clause [clause]...]
{
   // Parallel region code
}
```

OpenMP Directives

for:

Distributes iterations of a loop among the threads in a parallel region.

Syntax:

```
#pragma omp for [clause [clause]...]
for (init; test; incr) {
    // Loop body
}
```

OpenMP Directives

sections:

Divides the enclosed code block into independent sections that can be executed in parallel by different threads.

Syntax:

```
#pragma omp sections [clause [clause]...]
{
    #pragma omp section
    {
        // Section 1 code
    }
    #pragma omp section
    {
        // Section 2 code
    }
    // Additional sections...
}
```

OpenMP Directives

single:

Specifies that the enclosed code block should be executed by only one thread.

Syntax:

```
#pragma omp single [clause [clause]...]
{
   // Single thread code
}
```

OpenMP Directives

master:

Specifies that the enclosed code block should be executed only by the master thread (the thread with ID 0).

Syntax:

```
#pragma omp master [clause [clause]...]
{
   // Master thread code
}
```

OpenMP Directives

critical:

Defines a critical section, ensuring that only one thread executes the enclosed code block at a time to avoid data races.

Syntax

```
#pragma omp critical [name]
{
    // Critical section code
}
```

OpenMP Directives

atomic:

Performs an atomic operation on a shared variable, ensuring that the operation is executed atomically without interference from other threads.

Syntax:

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

OpenMP Directives

ordered:

Specifies that the enclosed loop or section should be executed in the order of loop iterations or section directives.

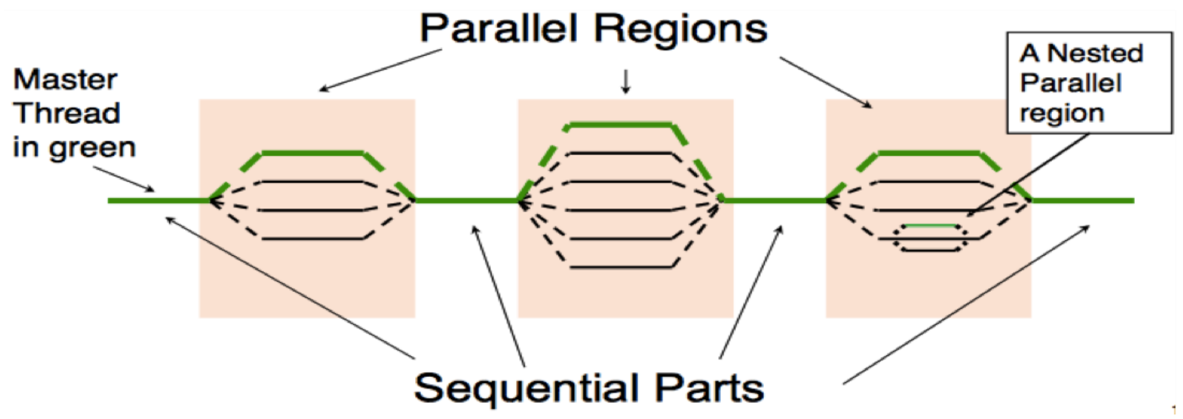
Syntax

```
#pragma omp ordered [clause [clause]...]
{
   // Ordered code
}
```

Architecture and Working of OpenMP:

1. Fork-Join Execution Model:

- OpenMP programs start with a single master thread that executes sequentially.
- When a parallel region is encountered, the master thread "forks" into a team of parallel worker threads.
- The instructions within the parallel region are then executed concurrently by the team of worker threads.
- At the end of the parallel region, the threads synchronize and "join" back to the single master thread.



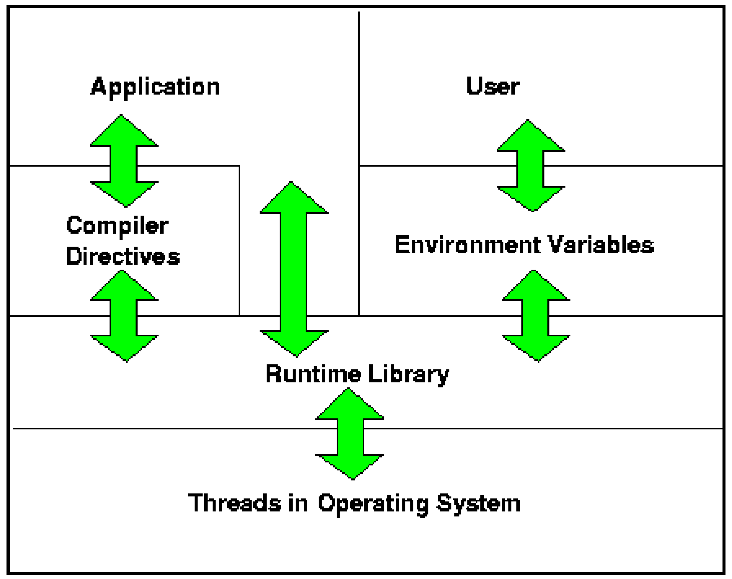
Architecture and Working of OpenMP:

2. Compiler Directives:

- OpenMP uses compiler directives (e.g., `#pragma omp`) to specify parallel regions, work sharing, and synchronization in the source code.

- These directives are embedded in the C/C++ or Fortran code and are recognized by the OpenMP-compliant compilers.

OpenMP Architecture



Architecture and Working of OpenMP:

3. Runtime Library:

- OpenMP provides a runtime library with functions and environment variables to manage the parallel execution.
- This includes routines for setting the number of threads, querying the execution environment, and controlling various aspects of the parallel program.

4. Shared Memory Model:

- OpenMP is designed for shared-memory parallel programming, where all threads have access to the same shared memory space.
- This simplifies data sharing and communication between the parallel threads.

5. Portability:

- The OpenMP API is designed to be portable across different shared-memory architectures, allowing the same code to run on a variety of systems.

Architecture and Working of OpenMP:

6. Incremental Parallelization:

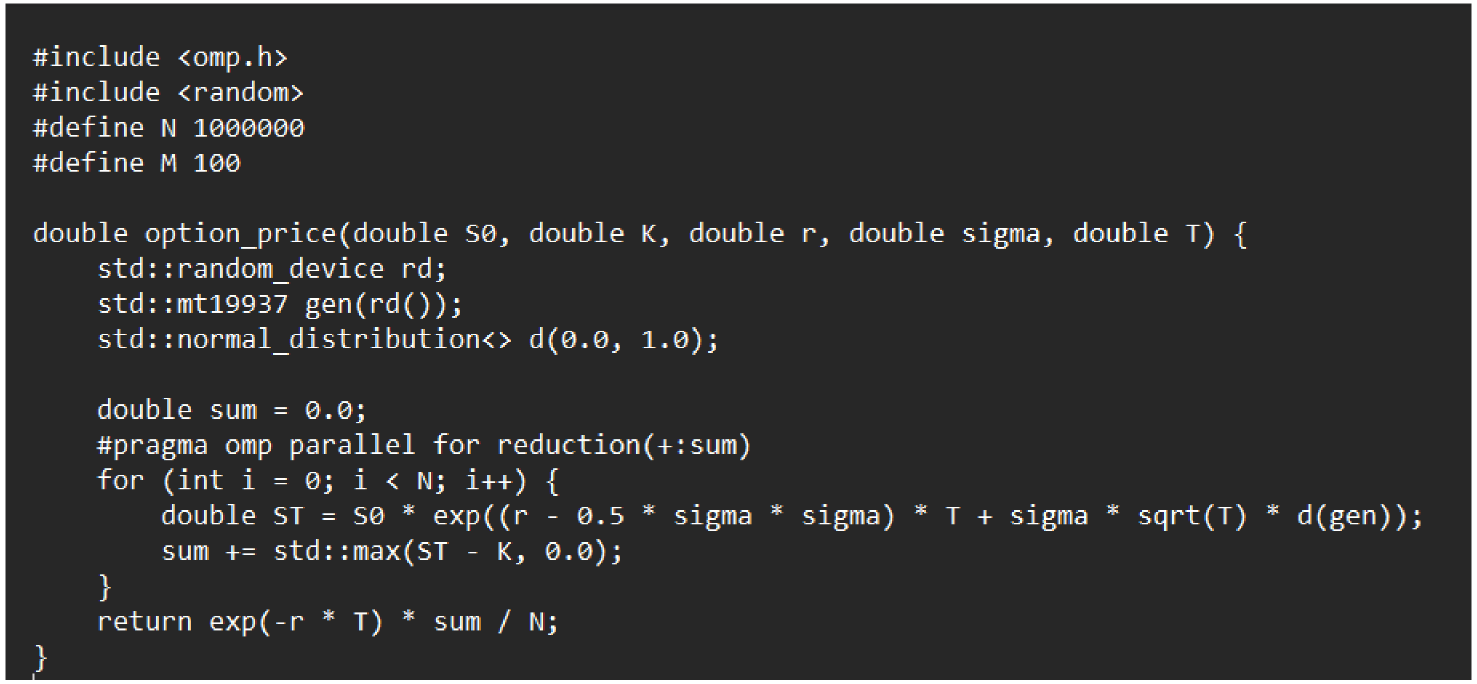
- OpenMP allows developers to incrementally parallelize their code, starting with small parts of the application and gradually expanding the parallel regions.
- This makes it easier to transition existing serial code to a parallel implementation.

7. Flexibility in Parallelism:

- OpenMP supports both fine-grained and coarse-grained parallelism, providing flexibility in how the parallelism is expressed in the code.

Applications/Examples

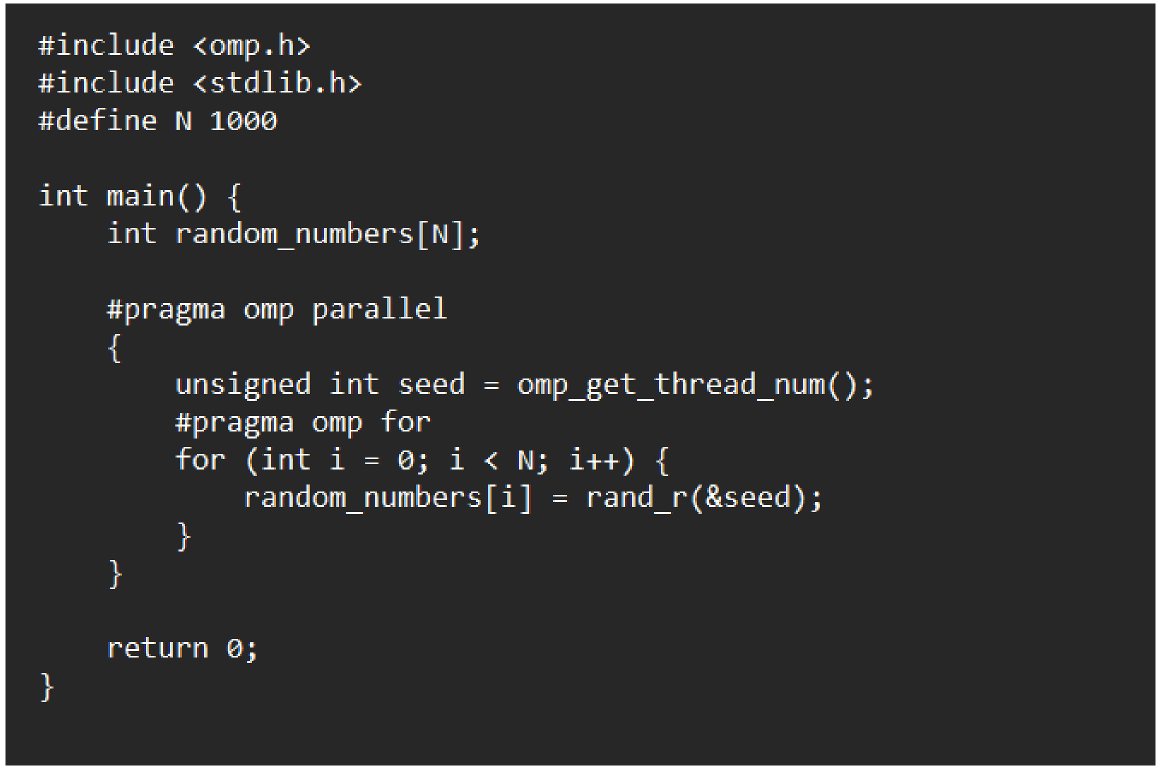
Monte Carlo Simulation - Option Pricing:



```
#include <omp.h>
#include <random>
#define N 1000000
#define M 100

double option_price(double S0, double K, double r, double sigma, double T) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<> d(0.0, 1.0);

    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < N; i++) {
        double ST = S0 * exp((r - 0.5 * sigma * sigma) * T + sigma * sqrt(T) * d(gen));
        sum += std::max(ST - K, 0.0);
    }
    return exp(-r * T) * sum / N;
}
```

Applications/Examples

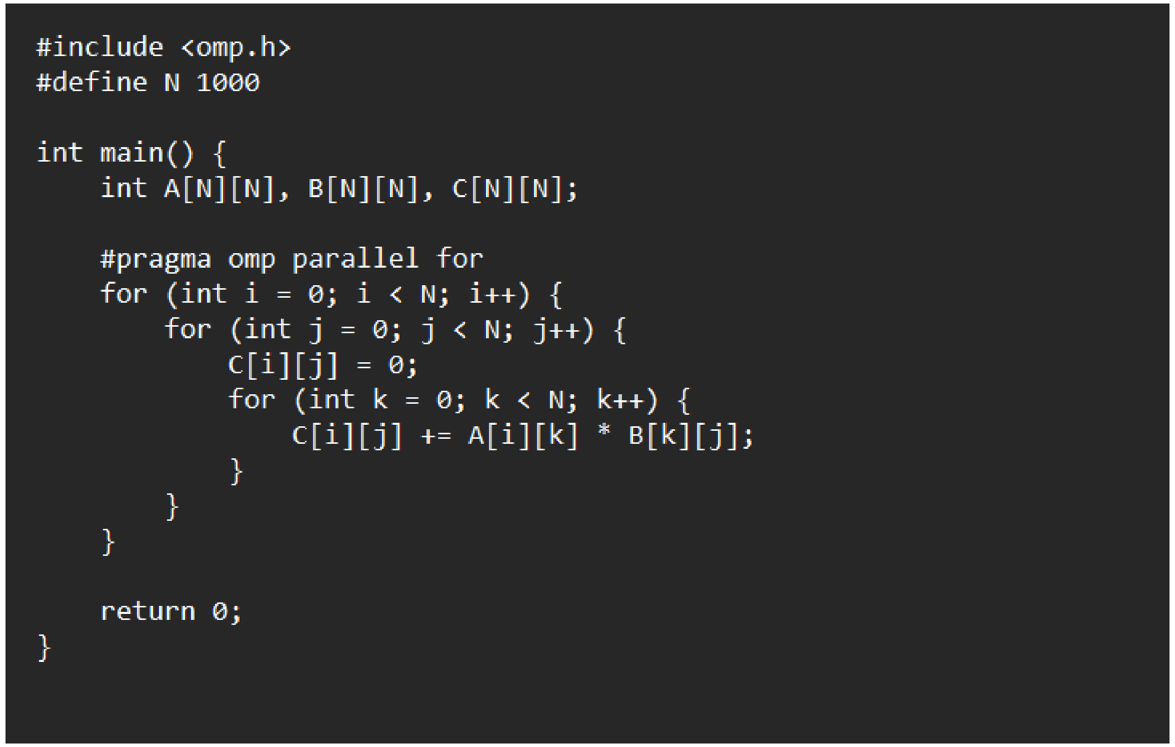
Random Number Generation:



```
#include <omp.h>
#include <stdlib.h>
#define N 1000

int main() {
    int random_numbers[N];

    #pragma omp parallel
    {
        unsigned int seed = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < N; i++) {
            random_numbers[i] = rand_r(&seed);
        }
    }

    return 0;
}
```

Applications/Examples

Matrix Multiplication:



```c
#include <omp.h>
#define N 1000

int main() {
    int A[N][N], B[N][N], C[N][N];

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return 0;
}
```

Comparison with CUDA & MPI

Feature	OpenMP	CUDA	MPI
Model	Shared memory (threads share address space)	Data-parallel on NVIDIA GPUs	Distributed memory (processes have own memory)
Parallelism	Shared-memory within a single node	Data-parallel on GPUs	Distributed-memory across multiple nodes
Ease of Use	Easy with high-level directives	Requires specialized GPU programming knowledge	Medium (needs message passing understanding)
Portability	High (across shared-memory architectures)	Low (limited to NVIDIA GPUs)	High (across distributed-memory systems)
Performance	Efficient for multicore CPUs	Excellent for data-parallel GPU problems	Scalable for large workloads (comm intensive)
Complexity	Lower for shared-memory systems	Higher due to GPU architecture and optimization	Medium (requires message passing knowledge)
Hybrid Approach	Yes (combines with MPI for both)	Yes (combines with MPI for hybrid parallelism)	Yes (combines with OpenMP for intra-node)
Best Use Cases	Loop parallelism, multicore CPUs	Large-scale data-parallel computations on GPUs	Distributed workloads, communication-intensive

References

- The references for the provided OpenMP resources are as follows:

- https://cces.unicamp.br/2022/10/25/the-openmp-cluster-programming-model/
- https://en.wikipedia.org/wiki/OpenMP
- https://www.hpe.com/psnow/resources/ebooks/a00115296en_us_v1/OpenMP_Overview.html
- https://researchcomputing.princeton.edu/education/external-online-resources/openmp
- https://people.math.sc.edu/Burkardt/c_src/openmp/openmp.html
- https://www.roe.ac.uk/ifa/postgrad/pedagogy/2009_kiessling.pdf
- https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html

Thank you