Designing and Implementing Parallel Programming Constructs

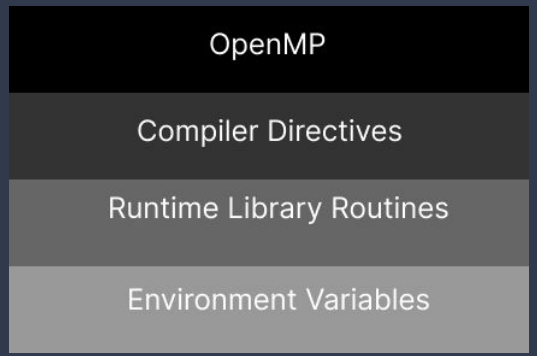
Team

- Vidhish Trivedi (IMT2021055)
- Munagala Kalyan Ram (IMT2021023)
- Sankalp Kothari (IMT2021028)

Agenda

- **OpenMP:** Cover OpenMP, execution model, scoping, common design patterns, memory model, error handling, static analysis (briefly), OpenMP and Python.
- **MPI:** Basics of MPI, OpenMPI implementations for C/C++, communication and error handling. OpenMPI implementation for Java bindings (briefly).
- **OpenMP and MPI in Other Languages :** Exploring implementation of OpenMP and MPI in Java, Python and Ocaml, proposing an high level architecture for implementation of tool in new language.

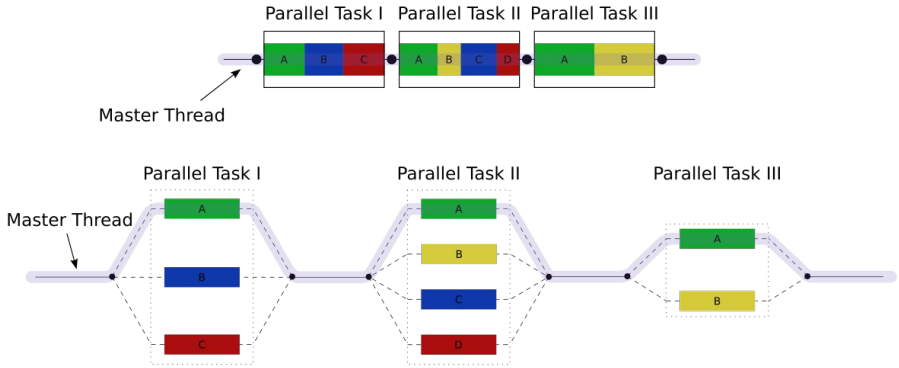
OpenMP Programming



- OpenMP (“Open Multi-Processing”) is a compiler-side application programming interface (API) for creating code that can run on a system of threads. No external libraries are required in order to parallelize your code.
- Provides a way to use thread safe methods and parallel sections of code that can be set with simple scoping.

OpenMP Execution Model

- Thread-based parallelism.
- Fork-Join Model.
- Single-Program-Multiple-Data (SPMD).



OpenMP Fork-Join Model

OpenMP Directive Scoping

- The static extent of an OpenMP directive extends to the code textually enclosed between the beginning and the end of a structured block following a directive.
- An OpenMP directive that appears independently from another enclosing directive is said to be an orphaned directive. It exists outside of another directive's static extent.
- The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Common OpenMP Design Patterns

- Single Program Multiple Data (SPMD)
- Loop Level Parallelism
- Tasks And Divide And Conquer:
    - The general idea is to define three basic phases of the algorithm: split, compute, and merge.

OpenMP Memory Management

- **Shared Memory Model:** SPMD is underlying paradigm - all threads have potential to execute the same program code, however, each thread may access modify different data and traverse different execution paths
- OpenMP provides a relaxed-consistency and temporary view of thread memory where threads have equal access to shared memory where variables can be retrieved/stored.

OpenMP Error Handling

- OpenMP forbids code which leaves the openmp block via exception.
- C++ also provides a concurrency runtime (and the associated namespace) in an effort to aid parallel, concurrent programming.

Static Analysis of Parallel Programs (Manual)

- **Manual Approach:**
  - Code Review
  - Identify Parallelizable Regions
  - Data Dependencies
  - Loop Analysis
  - Performance Modelling

Static Analysis of Parallel Programs (Automated)

- **Automated Approach:**
  - Parsing
  - Control Flow Analysis
  - Data Flow Analysis
  - Dependency Analysis
  - Pattern Matching
  - Performance Modelling
- For exploring different execution paths, one may use symbolic execution.

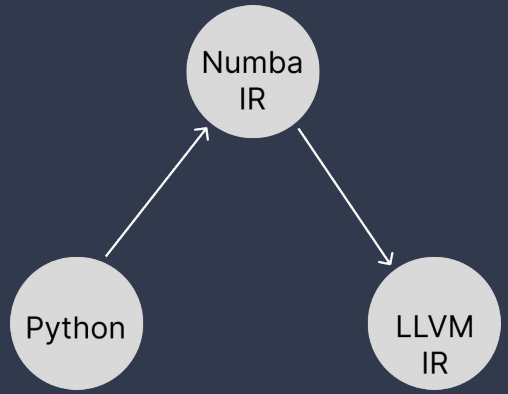
Limitations in Python

- Python is not designed for parallel programming with threads, since its Global Interpreter Lock (GIL) prevents multiple threads from simultaneously accessing Python objects. This effectively prevents data races and makes Python naturally thread safe.
- The GIL prevents multiple threads from simultaneously accessing Python objects.

Prior Approaches

- Parallel accelerators, which exploit common patterns in code to expose concurrency which is then executed in parallel.
- Embedding parallelism in functions from modules such as Numpy.

PyOMP



- A prototype system with support for OpenMP directives in Python.
- Directives in OpenMP are exposed using the Python with statement, which are interpreted by a custom Numba JIT compiler and combined with a backend that connects these constructs to analogous entry points in the generated LLVM code.
- This LLVM code is compiled using an LLVM system which includes support for OpenMP.

Results

- The results, along with the respective code files and the Jupyter notebook used for analysis can be found in our code repository here.

Programming using MPI

MPI is a specification designed for developers and users of message passing libraries, facilitating data exchange between processes in parallel computing.

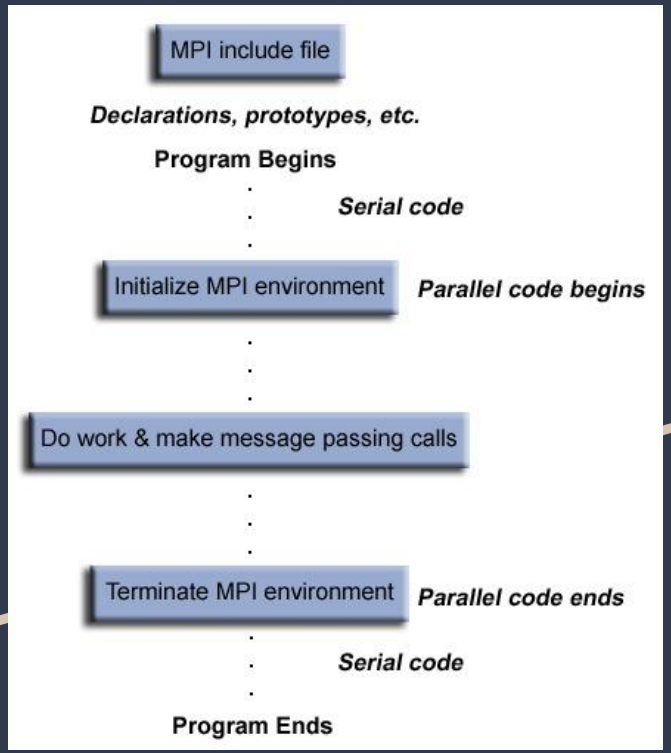
Developed initially in the 80s, but has evolved now to support various hardware platforms.

Advantages of using MPI -

- Standardization
- Portability
- Functionality
- Availability
- Performance

MPI 3 now has around 430 routines defined for helping with parallel programming.

OpenMPI for C



In the next few slides, we will look at how OpenMPI (one of the implementations of the MPI standard) is used with C.

Environment Management Routines - help initialize the environment for OpenMPI, which would help use the MPI routines and build parallelizable code blocks.

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Abort
- MPI_Wtime
- MPI_Finalize

Communication Routines

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.

Most MPI routines require you to specify a communicator as an argument.

Communication Routines are mainly of 2 types:

- Point-to-Point
- Collective

Point-to-Point Communication Routines

MPI provides operations for message passing between tasks, including synchronous and asynchronous options.

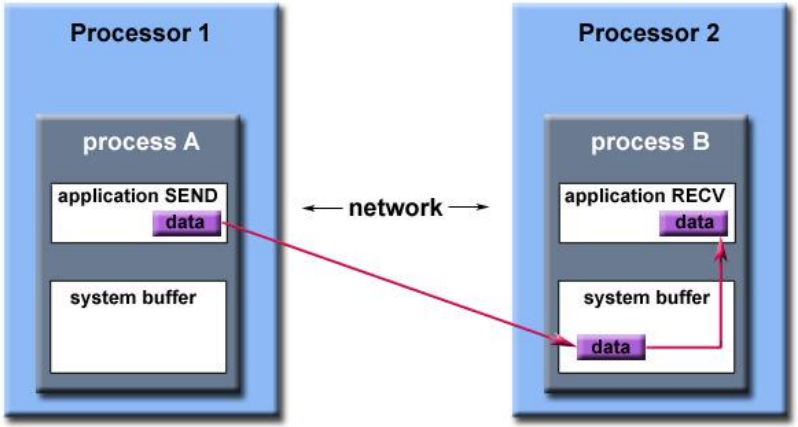
They are majorly of 2 types -

- Blocking
- Non - blocking

Point-to-Point Communication Routines

Order and fairness - MPI guarantees order and fairness for blocking calls. For non-blocking calls, user needs to take care to avoid starvation of some task.

Buffering: MPI uses system buffer to store data when sender and receiver are out of sync



Path of a message buffered at the receiving process

Collective Communication Routines

MPI offers routines for synchronization, data movement, and collective computation among tasks. For example -

MPI_Barrier

MPI_Bcast

MPI_Scatter

MPI_Gather

MPI_Reduce

MPI_Reduce_scatter

Derived Data Types

MPI allows for the creation of custom data structures, known as derived data types, based on sequences of primitive data types. These derived data types enable the representation of non-contiguous data structures conveniently, treating them as if they were contiguous.

Examples -

- Contiguous
- Vector
- Indexed
- Struct

Sample Programs and Results

To demonstrate the usage of these aforementioned MPI routines, we have written a few C codes, and compared their serial and parallel runtimes.

Sample programs - here

Results on runtime - here

OpenMPI in Java

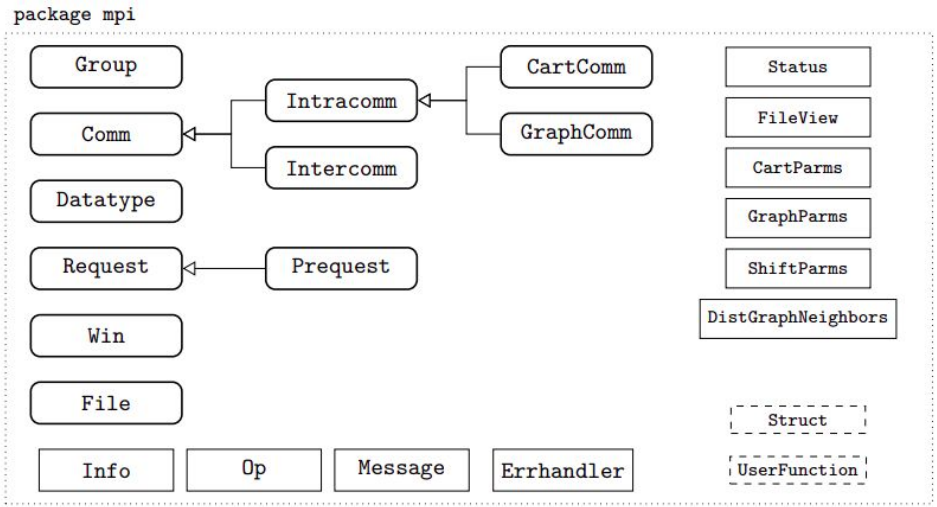
Paper referenced - here

The work referenced above explores how Java bindings can be integrated with OpenMPI to allow for HPC applications written in Java.

In using OpenMPI for Java, we use something known as the JNI (Java Native Interface) which allows us to directly access the memory via C/C++ directives and functions.

OpenMPI in Java - Internal Implementation

- JNI interface
- Java Classes
- Resource Management
- Memory Management
- Thread Safety
- Error Handling



Communication Routines and Derived Data Types

Communication Routines remain more or less the same, but their implementation in Native code differs, because of the fact that Java does not have pointers.

In Open MPI for Java, derived datatypes such as contiguous, vector, and indexed data types are defined similarly to their counterparts in C.

In Java, a sub-class of Struct is typically used, where data fields are defined along with corresponding access methods.

Exploring OpenMP and MPI Tools in Other Programming Languages

OpenMP and MPI is implemented mainly for C/C++ and Fortran.

Some tools have been built that mimic the functionality of OpenMP and MPI in Java, Python and Ocaml.

OpenMP :

- JOMP : An OpenMP-like Interface for Java
- Pymp - OpenMP Like Python Programming

MPI :

- MPI For Python - mpi4py
- OCamlMPI

JOMP : An OpenMP-like Interface for Java

Implements an OpenMP-like set of directives and library routines in Java, including the its own compiler and runtime library.

The directives are embedded into comments , i.e using ”//” and the syntax of directives is similar to OpenMp in C.

The parallel, reduction, sections and section, barrier, single and critical directives are provided by JOMP.

Also provides library functions to get metadata like number of threads, max threads, thread number, etc.

The jomp.runtime library provides the necessary functionality to support parallelism by implementing classes that internally call the native Java thread functions. These classes and functions are called by the JOMP compiled code.

JOMP Compiler

```
import jomp. runtime. * ;
public class Hello {
    public static void main (String argv[]){
        int myid;
        //omp parallel private(myid)
        {
            myid = DMP.getThreadNumO ;
            System.out.println("Hello from " + myid) ;
        }
    }
}
```

The JOMP compiler converts the code with directives to the parallelized code with calls to the corresponding functions/classes.

“doParallel” is called by the master thread to initialize threads (tasks) and also synchronizes between threads.

Each task is an instance of the busyTask class and it has the “go” function that contains the logic that should be executed by the task

```
import jomp.runtime.*;
public class Hello {
    public static void main (String argyll) {
        int myid;
        __omp_class_0 __omp_obj_0 = new __omp_class_0();
        try {
            jomp.runtime.0MP.doParallel(__omp_obj_0);
        }
        catch(Throwable __omp_exception) {
            System.err.println("OMP Warning: exception in
                ") ;
        }
    }

    private static class __omp_class_0
            extends jomp.runtime.BusyTask {
        public void go(int __omp_me) throws Throwable {
            int myid;
            myid = OMP.getThreadNumO;
            System.out.println("Hello from " + myid);
        }
    }
}
```

Pymp - OpenMP Like Python Programming

```
ex_array = pymp.shared.array((100,), dtype='uint8')
with pymp.Parallel(4) as p:
    for index in p.range(0, 100):
        ex_array[index] = 1
        # The parallel print function takes care of asynchronous output.
        p.print('Yay! {} done!'.format(index))
```

Pymp implements some of the OpenMP features like a single master thread forking multiple threads, sharing data, synchronizing between threads in python.

Processes are forked instead of threads by the master process.

The underlying implementation makes use of the **”multiprocessing.py”** library which provides process-based parallelism.

When a parallel region ends, all the child processes exit and only the master process continues.

Key Features Implemented :

Master Process - The master process has thread_num 0 and responsible to synchronize between processes

Sections and section - Each thread assigned different task.

pymp.Parallel(n) - Defines the start of parallel region. “n” is number of processes.

Scheduling - Static schedule done by **pymp.range** and dynamic scheduling done by **pymp.xrange**

MPI For Python – mpi4py

MPI for Python (mpi4py) provides Python bindings for the Message Passing Interface (MPI) standard, i.e MPICH package implemented in C.

The setup process of mpi4py will compile the necessary python bindings and link them with the MPICH libraries.

It basically acts as an interface between Python and MPICH.

Key Functionalities of MPI implemented :

- **Communication Support**
- **Communicating Python Objects :** By using pickle library to convert object to binary format.
- **Blocking Communication**
- **Non-Blocking Communication**
- **Environment Management**

OCamlMPI - MPI in Ocaml

OCamlMPI provides OCaml bindings for a large subset of MPI functions.

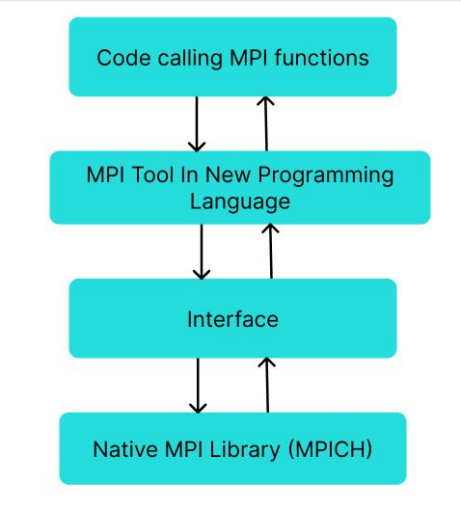
Flow of an Ocaml Program With OcamlMPI :

- **Binding Generation :** OCamlMPI provides OCaml bindings for the MPI functions by adding wrappers that internally call the native MPI libraries.
- **Compilation :** Ocamlc links the Ocaml code and the MPICH library functions and generates the executable.
- **Execution :** During execution the native MPI library handles the actual communication between processes.

Key Functionalities of MPI Implemented :

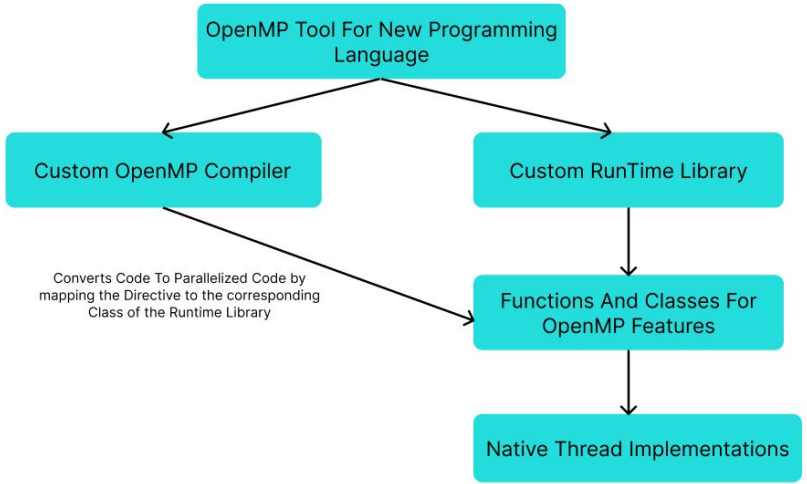
- **Communicating Data Objects :** Pickling objects to get binary .
- **Point-to-Point Communication**
- **Environment Management**
- **Collective Communication**

Implementing OpenMP and MPI in another Language



We propose an high level abstraction that a programmer can use while implementing OpenMP and MPI in a new programming language.

Takes inspiration from the tools discussed.



Thank You!