Project Report, T2-23-24

6th Semester

Designing and Implementing Parallel Programming Constructs

Authors:

Vidhish Trivedi — IMT2021055
Munagala Kalyan Ram — IMT2021023
Sankalp Kothari — IMT2021028

April-May, 2024

Contents

1 Problem Description

1. In high-performance computing, parallel programming is pivotal for efficiently executing intricate computational tasks across multiple processors. Two predominant frameworks, OpenMP and MPI, cater to parallelism on shared memory architectures and distributed computing environments, respectively.

2. This study and implementation project aims to explore the core aspects of parallel programming, focusing on syntax, semantics, and mechanisms for managing parallel execution. It entails analyzing the design considerations and challenges associated with integrating parallel programming constructs into programming languages, encompassing syntax design, type safety, memory management, and error handling in parallel environments.

3. Furthermore, the project should investigate the implementation of parallel constructs within a programming language, considering compiler and runtime support. Strategies for integrating OpenMP and MPI into existing languages or designing new language features to accommodate these models is also explored.

2 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. OpenMP is often considered more user friendly with thread safe methods and parallel sections of code that can be set with simple scoping. However, it is limited to the amount of threads available on a node – in other words, it follows a shared memory model. On a node with 64 CPUs, you can use no more than 64 processors. For an overview of the OpenMP API, see: here.

2.1 Private vs Shared Variables

We will now have a look at different variable types in OpenMP and their implementation. OpenMP allows for two types of variables:

- Private types create a copy of a variable for each thread in the parallel system.
- Shared types hold one instance of a variable for all threads to share.

To indicate private or shared memory, declare the variable before the parallel section and annotate the pragma directive as follows:

```
1  #pragma omp shared(shared_var1) private(private_var1, private_var2)
```

Variables that are created and assigned inside of a parallel section of code will be private by default, and variables created outside of parallel sections will be public by default.

2.2 Goals of OpenMP

- Standardization
- Lean and Mean
- Ease of Use
- Portability

2.3 Components of OpenMP

OpenMP is comprised of three primary API components (as of version 4.0):

- Compiler Directives
- Runtime Library Routines
- Environment Variables

2.4 OpenMP Programming Model

2.4.1 Shared Memory Model

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory (Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA)).

2.4.2 OpenMP Execution Model

Thread Based Parallelism: OpenMP programs accomplish parallelism exclusively through the use of threads. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. They can be thought of as functions in execution (similar to how a process can be thought of as a program in execution). Threads exist within the resources of a single process. Without the process, they cease to exist.

2.4.3 Fork-Join Model

- All OpenMP programs begin as a single process: the **master thread**. The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK**: the master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team of threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

Single-Program-Multiple-Data (SPMD) is the underlying programming paradigm, and is described in later sections.

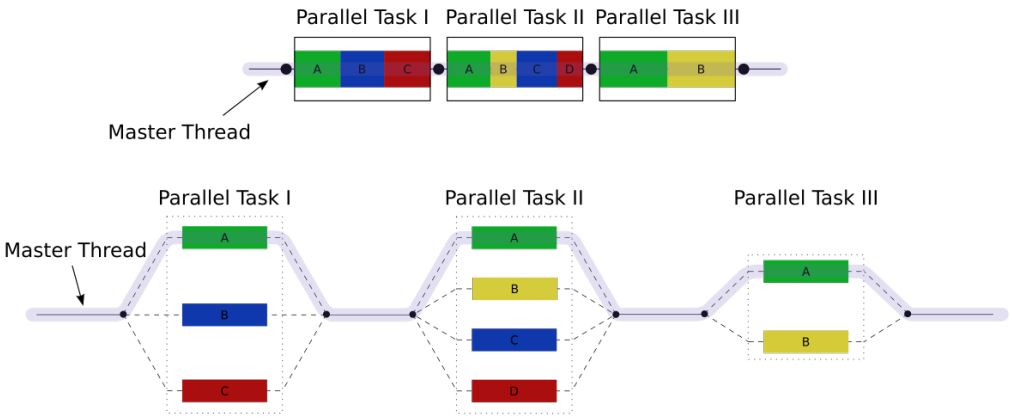


Figure 1: Fork-Join Programming Model - OpenMP. Source: Wikipedia

2.5 OpenMP Directive Scoping

2.5.1 Static Extent (Lexical)

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.

The static extent of a directive does not span multiple routines or code files.

2.5.2 Orphaned 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 (lexical) extent.

An orphaned directive can span routines and possibly code files.

2.5.3 Dynamic Extent

The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

For an example on how directives are scoped, see: here.

2.6 Directive Binding and Nesting Rules

2.6.1 Directive Binding

- The `DO/for`, `SECTIONS`, `SINGLE`, `MASTER`, and `BARRIER` directives bind to the dynamically enclosing `PARALLEL`, if one exists. If no parallel region is currently being executed, the directives have no effect.
- The `ORDERED` directive binds to the dynamically enclosing `DO/for`.
- The `ATOMIC` directive enforces exclusive access with respect to `ATOMIC` directives in all threads, not just the current team.

- The `CRITICAL` directive enforces exclusive access with respect to `CRITICAL` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing `PARALLEL`.

2.6.2 Directive Nesting

- A work-sharing region may not be closely nested inside a work-sharing, explicit `TASK`, `CRITICAL`, `ORDERED`, `atomic`, or `MASTER` region.
- A `BARRIER` region may not be closely nested inside a work-sharing, explicit `TASK`, `CRITICAL`, `ORDERED`, `ATOMIC`, or `MASTER` region.
- A `MASTER` region may not be closely nested inside a work-sharing, `ATOMIC`, or explicit `TASK` region.
- An `ORDERED` region may not be closely nested inside a `CRITICAL`, `ATOMIC`, or explicit `TASK` region.
- An `ORDERED` region must be closely nested inside a loop region (or parallel loop region) with an `ORDERED` clause.
- A `CRITICAL` region may not be nested (closely or otherwise) inside a `CRITICAL` region with the same name. Note that this restriction is not sufficient to prevent deadlock.
- `PARALLEL`, `FLUSH`, `CRITICAL`, `ATOMIC`, `TASKYIELD`, and explicit `TASK` regions may not be closely nested inside an `ATOMIC` region.

2.6.3 Closely Nested Directives

In OpenMP, closely nested directives refer to situations where multiple OpenMP directives are nested within each other, often within the same code block or function. This nesting typically involves directives like `PARALLEL` regions, loop parallelization directives (such as `PARALLEL FOR`), and work-sharing constructs (such as `SECTIONS` or `TASK`). For an example, see: here.

2.7 OpenMP Design Patterns

2.7.1 Single Program Multiple Data (SPMD)

- By creating a team of N threads, you can explicitly manage the distribution of work among threads using the thread's rank (ranging from 0 to $N-1$) and the total number of threads, N.
- Like most multithreaded programming environments, OpenMP operates as a shared memory API. All threads are part of a single process and share the process's heap. Variables outside a parallel construct are shared by default within the construct, while variables created within the construct are private by default (meaning each thread in the team has its own copy).
- It's considered good practice in OpenMP programming to clearly specify the status of variables within an OpenMP construct.

2.7.2 Loop Level Parallelism

- The Loop Level Parallelism pattern is where most people start with OpenMP.
- Parallelism is introduced through a single directive to express the parallel for construct. This construct creates a team of threads and then distributes the iterations of the loop among the threads. To accumulate the summation across loop iterations, we include the reduction clause.

2.7.3 Tasks And Divide And Conquer

- This important pattern is heavily used by more advanced parallel programmers.
- The general idea is to define three basic phases of the algorithm: split, compute, and merge.
  - The split phase recursively divides a problem into smaller subproblems.
  - After enough splits, the subproblems are small enough to directly compute in the compute phase.
  - The final phase merges subproblems together to produce the final answer.

2.8 Syntax and Semantics in OpenMP Programs

Since this is potentially a vast topic, we make the choice to omit this from our report, it is suggested that interested readers go through our GitHub repository for more details.

The OpenMP API syntax and semantics for directives and clauses, along with the general rules for scoping and restrictions (wherever applicable) for popular constructs have been covered in detail. The essentials are covered in the following documents (available in the aforementioned repository):

1. C/C++ OpenMP Directives
2. Parallel Region Construct
3. Work-Sharing Constructs
4. Task Construct
5. Synchronization Constructs

2.9 Memory Management in OpenMP Programs

OpenMP is designed for multi-processor/core, shared memory machines and makes use of a **Shared Memory Model**. Single-Program-Multiple-Data (SPMD) is underlying programming paradigm - all threads have potential to execute the same program code, however, each thread may 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. Each

thread also has its own temporary copies of variables that may be modified independent from variables in memory.

The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory.

When it is critical that all threads have a consistent view of a shared variable, the programmer (or compiler) is responsible for insuring that the variable is updated by all threads as needed, via an explicit action (FLUSH), or implicitly (via compiler recognition of program flow leaving a parallel regions).

OpenMP also provides **Data Scope Attribute Clauses** (also called Data-Sharing Attribute Clauses). Since OpenMP is based on the shared memory programming model, most variables are shared by default.

For C/C++:

- Global variables include file scope variables, static.
- Private variables include loop index variables, stack variables in subroutines called from parallel regions.

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include: `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, `SHARED`, `DEFAULT`, `REDUCTION`, `COPYIN`. They are used in conjunction with several directives (`PARALLEL`, `DO/for`, and `SECTIONS`) to control the scoping of enclosed variables.

These constructs provide the ability to control the data environment during execution of parallel constructs.

- They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back).
- They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.

Note: Data Scope Attribute Clauses are effective only within their lexical/static extent.

There are several ways to ensure OpenMP memory consistency (these are similar to how consistency is ensured when multiple transactions are executed in a database), more details can be found here. Additionally, OpenMP v5.2 also provides certain constructs and directives for managing memory at a lower level, but that is beyond the current scope of the project. Interested readers may see the OpenMP specification.

2.10 Error Handling in OpenMP Programs

OpenMP forbids code which leaves the openmp block via exception. We would want to have a way to get the exceptions from an openmp block with the purpose of rethrowing them in the main thread and handling them at a later point.
Consider the following implementation:

```
    class ThreadException {
        std::exception_ptr Ptr;
        std::mutex         Lock;
    public:
        ThreadException(): Ptr(nullptr) {}
        ~ThreadException(){ this->Rethrow(); }
        void Rethrow(){
            if(this->Ptr) std::rethrow_exception(this->Ptr);
        }
        void CaptureException() {
            std::unique_lock<std::mutex> guard(this->Lock);
            this->Ptr = std::current_exception();
        }
    };
    /* Some more code... */
    ThreadException except;
        #pragma omp parallel
        {
        try {
            /* code which may throw an exception... */
        }
        catch(...) { except.CaptureException(); }
}
```

This code provides a mechanism to capture any exceptions thrown within the parallel block, rethrow them in the main thread, and handle them at a later point.We do not worry about handling multiple exceptions at once, since in C++: Within a given thread, there can only ever be at most one exception active at any one time.

There are scenarios where a second exception would be thrown; for example, if stack unwinding causes an object to throw and an exception (say, from a destructor). Since two exceptions cannot be active at once, the program will simply terminate. At no time can there be two active exceptions such that a catch block would have to decide which to evaluate.

We could use C++ language features like templating with variadic members to clean up the above code, which is quite tedious to use (requiring us to explicitly write a try-except block for every OpenMP construct we use). If you are unfamiliar with variadic templates in C++, see here.

```
    class ThreadException {

    /* Old code as written previously... */

```

```
    template <typename Function, typename... Parameters>
    void Run(Function f, Parameters... params)
    {
        try
        {
            f(params...);
        }
        catch (...)
        {
            CaptureException();
        }
    }
};
```

Then, when using this class inside an OpenMP construct, we can use a lambda function as follows:

```
    ThreadException e;

    #pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        e.Run([=]{
            /* code that might throw an exception... */
        });
    }
    e.Rethrow()
```

This works, but is not recommended, never throw an exception out of a destructor - doing so results in undefined behavior. *Bjarne Stroustrup*, the creator of C++, makes the point that "the vector destructor explicitly invokes the destructor for every element. This implies that if an element destructor throws, the vector destruction fails... There is really no good way to protect against exceptions thrown from destructors, so the library makes no guarantees if an element destructor throws". If you wish to dive into this discussion, see this StackOverflow discussion.

C++ also provides a concurrency runtime (and the associated namespace) in an effort to aid parallel, concurrent programming. This is out of the scope of our current discussion in this project, the key points on how it can be used to handle errors in a much more consistent manner can be found here. The article again emphasizes that if a task or parallel algorithm receives multiple exceptions, the runtime marshals only one of those exceptions to the calling context. The runtime does not guarantee which exception it marshals. It is also possible to convert an OpenMP loop that uses exception handling to use the concurrency runtime, details of which can be found here.

2.11 Static Analysis of Parallel Programs

2.11.1 Manual Approach

1. **Code Review**: Start by reviewing the code to understand its structure, dependencies, and potential bottlenecks.

2. **Identify Parallelizable Sections**: Look for loops, functions, or blocks of code that can run independently of each other (tools have been developed which help a user do this). These are potential candidates for parallelization. We are, more often than not, focused on loops, but it's worth noting that sequences such as `a = 5; b = 6; c = 7; d = 23;` can also be parallelized, as discussed in the "Commutative" section below.
   Regarding independence: By tracking the flow of data within the code, sections with independent data flows can potentially be parallelized. It's crucial to identify instances where statement N+m relies on a value computed at statement N, or on a value dependent on the result of statement N. For operations involving arrays, if all array references are to offset j without any references to j+1 or j-1, then there's no need to compute the value at j-1 before calculating the value at j, enabling parallel computation of both j-1 and j values.
   Regarding commutativity: If it can be proven that performing operation A followed by operation B yields the same result as performing B followed by A, and assuming A and B are independent, then they can be executed in parallel.

3. **Data Dependencies**: Analyze the data flow within the code to identify dependencies between different sections. If a section of code relies on the output of another section, parallelizing them might introduce data hazards or race conditions.

4. **Loop Analysis**: Examine loops to determine if iterations can be executed concurrently. Loops with no dependencies between iterations are good candidates for parallel execution.

5. **Performance Modeling**: Estimate the potential performance gains from parallelization based on factors such as the number of available processors, the nature of the workload, and the overhead of parallelization. Remember that it is not always beneficial to parallelize a section of code.

2.11.2 Automated Approach

1. **Parsing**:

   - **Lexical Analysis**: The code is broken down into tokens such as identifiers, keywords, operators, and punctuation symbols.
   - **Syntax Analysis**: Tokens are analyzed to build a parse tree or abstract syntax tree (AST) by applying grammar rules to recognize the syntactic structure of the code.
   - **AST Generation**: The AST represents the structure of the code in a hierarchical form, capturing relationships between different elements of the code.

2. **Control Flow Analysis**:

   - **Basic Block Identification**: The code is divided into basic blocks, sequences of statements with a single entry point and exit point.

- **CFG Construction**: A control flow graph (CFG) is constructed based on basic blocks to represent how control flows through the code.
- **Path Exploration**: Different paths through the CFG are explored to understand how control flows under different conditions.

3. **Data Flow Analysis**:
   - **Def-Use Chains**: Analysis of how variables are defined and used throughout the code.
   - **Reaching Definitions**: Determines which variable definitions reach each point in the code.
   - **Available Expressions**: Identifies expressions whose values are available at specific points in the code without recomputation.
4. **Dependency Analysis**:
   - **Data Dependencies**: Identifies dependencies between different parts of the code, such as data dependencies, control dependencies, and resource dependencies.
   - **Interprocedural Analysis**: Analyzes interactions between different functions and their effects on the overall program behavior.
5. **Pattern Matching and Rules**:
   - **Rule-based Analysis**: Applies predefined rules or patterns to identify common programming constructs, coding errors, and performance bottlenecks.
   - **Pattern Recognition**: Looks for specific patterns or sequences of code that match predefined templates or signatures associated with common programming idioms or problematic code constructs.
6. **Symbolic Execution**:
   - **Path Exploration**: Symbolically executes the code with symbolic inputs to explore different execution paths.
   - **Constraint Solving**: Collects constraints on program inputs and path conditions, then uses constraint solving techniques to determine the feasibility of paths and identify potential issues.
7. **Reporting**:
   - **Results Presentation**: Generates reports summarizing findings, including detected issues, potential optimizations, and recommendations for improving the code.

2.12 PyOMP: Multithreaded Parallel Python through OpenMP Support in Numba

The original paper used as a reference can be found here. A more elaborate and motivating summary of the paper can be found here.

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. Finally, this LLVM code is compiled using a LLVM system (such as Intel's LLVM system) which includes support for OpenMP.

The project is currently restricted to a subset of the 21 most commonly used elements of OpenMP, known as the OpenMP common core. More details can be found here. It is noteworthy that the proposed implementation strategy exploits an existing construct of Python (specifically, the `with` statement) to introduce new functionality to the language.

2.12.1 Numba And Implementation Details of PyOMP

Numba is a Just In Time (JIT) compiler that translates Python functions into native code optimized for a particular target.

The Numba JIT compiles PyOMP to native code in 4 basic phases:

1. **Untyped phase:** Numba converts Python bytecode into its own intermediate representation (IR), including "with" contexts that are OpenMP-represented in the IR as "with" node types, and performs various optimizations on the IR. Later, Numba removes these "with" nodes by translating them to other node types in the IR. For our PyOmp implementation, we added a new OpenMP node type into the IR, and we convert OpenMP "with" contexts into these new OpenMP IR nodes.

2. **Type inference phase:** Numba performs type inference on the IR starting from the known argument types to the function and then performs additional optimizations. No changes were made to the Numba typed compilation phase to support OpenMP.

3. **IR conversion phase:** Numba converts its own IR into LLVM IR.

4. **Compilation phase:** Numba uses LLVM to compile the LLVM IR into machine code and dynamically loads the result into the running application.

2.12.2 Converting PyOMP with clauses to Numba IR

- **When removing OpenMP "with" contexts and replacing them with OpenMP IR nodes:**

    1. Numba provides basic block information to demarcate the region that the with context covers.
    2. PyOMP places one OpenMP IR node at the beginning of this region and one at the end with a reference from the end node back to the start node to associate the two.

- **To determine what to store in the OpenMP IR node, PyOMP follows these steps:**

    1. PyOMP first parses the string passed to the OpenMP with context to create a parse tree.

2. A postorder traversal of the parse tree is performed, accumulating the information as we go up the tree until we reach a node that has a direct OpenMP LLVM tag equivalent.
3. At this point, convert the information from the sub-tree into tag form and then subsequently pass that tag up the parse tree.
4. Accumulate these tags as lists of tags up the parse tree until the traversal reaches a top-level OpenMP construct or directive, which have their own tags.

- Some directives are simple and require no additional processing, while others, particularly those that support data clauses, require additional clauses to be added to the Numba OpenMP node that are not necessarily explicitly present in the programmer's OpenMP string.

  For example, all variables used within the parallel, for and parallel for directives must be present as an LLVM tag even if they are not explicitly mentioned in the programmer's OpenMP statement. Therefore, for these directives our PyOMP prototype performs a use-def analysis of the variables used within the OpenMP region to determine if they are also used before or after the OpenMP region. If they are used exclusively within the OpenMP region then their default data clause is private. In all other cases, the default data clause is shared but of course these defaults can be overridden by explicit data clauses in the programmer OpenMP string.

2.12.3 Converting PyOMP Numba IR To LLVM

- **When a Numba OpenMP IR node is encountered in the process of converting Numba IR to LLVM IR:**

  1. The node is converted to an LLVM `OpenMP_start` (or `OpenMP_end`) call.
  2. Inside the Numba OpenMP node is a list of the clauses that apply to this OpenMP region.
  3. A 1-to-1 conversion of that list of clauses into a list of LLVM tags is performed on the LLVM `OpenMP_start` call.

- Code captures the result of the LLVM `OpenMP_start` call, and that result is passed as a parameter to the `OpenMP_end`, allowing LLVM to match the beginning and end of OpenMP regions.

- In the current PyOMP prototype, inside OpenMP regions, exception handling is currently omitted to ensure the requirement of single entry and single exit to and from OpenMP regions. In the usual case, Numba unifies the handling of exceptions with return values by adding an additional hidden parameter to the functions it compiles.

- **The Numba process of converting Numba IR to LLVM IR introduces many temporary variables into the LLVM IR not present in the Numba IR:**

  – Such temporaries used solely within an OpenMP region are classified as private in the tags associated with the surrounding OpenMP region's `OpenMP_start` demarcation function call.

  - PyOMP implements a callback in the Numba function to create these LLVM temporary variables and adds them as private to the previously emitted tags of the surrounding OpenMP region.

- **Certain OpenMP directives such as single and critical require the use of memory fences with acquire, release, or acquire/release memory orders:**
  - The current prototype stores this information in the Numba OpenMP IR node as created during the untyped phase.
  - During conversion of those OpenMP IR nodes to LLVM, if the node requires memory fences, then the equivalent LLVM fence instructions are inserted into the LLVM IR.

2.13 Running-time Comparisons

Below we summarize the time taken by various programs to run when executed serially, or in parallel with 4 and/or 16 threads. Please note that all time values are measured in seconds, and are measured for the execution of the algorithm itself. As a consequence, the time values do not include time for I/O and initialization done before the respective function is called.

Serial	Parallel (4 threads)
0.1093	0.024675
0.111296	0.028048
0.108654	0.03841
0.106431	0.021033
0.107702	0.024388
0.103633	0.020935
0.103937	0.021566
0.108137	0.027966
0.110272	0.022501
0.103302	0.040382
0.103955	0.028359
0.10953	0.026498
0.10676	0.021088
0.103185	0.02242
0.102722	0.023444

(a) Comparison of Serial and Parallel (4 threads) Execution Times for Computing Running Sum from 1 to 1,00,000,000 in C++ and OpenMP.

Serial	Parallel (4 threads)
5.956	2.142
6.029	2.052
5.978	2.205
5.923	2.153
5.982	2.248
5.919	2.213
5.997	2.097
5.731	2.045
5.939	2.057
5.87	2.234
6.092	1.996
5.833	2.342
5.874	2.101
6.01	2.386
6.01	2.334

(b) Comparison of Serial and Parallel (4 threads) Execution Times for Computing Matrix Multiplication between 2 Matrices of size 1000×1000 in C++ and OpenMP.

Figure 2: Results - I

Serial	Parallel (4 threads)
0.0435164	0.0125475
0.0445689	0.0139922
0.042894	0.0145378
0.0428758	0.0145505
0.0431849	0.0127452
0.0445349	0.0132974
0.0440722	0.0121217
0.0448087	0.0132808
0.0475593	0.0134824
0.0427257	0.0133697
0.0430829	0.0135711
0.0424828	0.013606
0.0445851	0.0127044
0.0425597	0.0138185
0.043664	0.0128437

Table 1: Comparison of Serial and Parallel (4 threads) Execution Times for Computing Dot Product between 2 Vectors of size 10,000,000 in C++ and OpenMP.

Serial	Parallel (4 threads)	Parallel (16 threads)
15.2575	8.08901	2.88764
15.3137	8.25734	2.77966
15.3512	8.10748	2.72371
15.331	8.32063	2.85347
15.3587	8.26409	2.77054
15.4488	8.46121	2.72428
15.5203	8.1501	2.74454
15.3239	8.28929	2.69408
15.3449	8.51524	2.70538
15.3866	8.29935	2.71911
15.4819	8.41149	2.70983
15.4003	8.12392	2.78333
15.338	8.20179	2.93494
15.416	8.09601	2.83811
15.3457	8.70035	2.822

Table 2: Comparison of Serial, Parallel (4 threads), and Parallel (16 threads) Execution Times for Counting Prime Numbers between between 1 and 5,00,000 in C++ and OpenMP.

Serial	Parallel (4 threads)	Parallel (16 threads)
8.015401	5.320782	2.10765
8.042783	5.430551	2.016979
8.333938	5.470242	2.075684
8.192614	5.223805	1.990315
7.954656	5.674587	2.116606
8.039976	5.05922	2.022639
7.791612	5.08978	2.129192
8.009237	5.15405	2.138217
7.782089	5.230838	2.04382
7.925361	5.498542	2.058118
7.823359	5.55267	2.102212
7.838853	4.390961	2.059307
7.75797	5.452149	2.142883
7.874805	5.435591	2.160916
7.83288	5.067389	1.979952

Table 3: Comparison of Serial, Parallel (4 threads), and Parallel (16 threads) Execution Times for Merge Sort on an array with 1,00,000,000 elements in C++ and OpenMP.

Serial	Parallel (16 threads)	PyOMP (16 threads)	Python (Serial)
0.41121	0.057464	0.019570112	5.845307112
0.420943	0.060519	0.014625072	5.840228796
0.419706	0.067449	0.024492025	5.890022278
0.42357	0.055788	0.01462698	5.857032299
0.414644	0.079546	0.016587019	5.784207344
0.416654	0.061623	0.014477015	5.837875128
0.412854	0.062451	0.015317917	5.83210206
0.411772	0.062103	0.01697588	6.061603546
0.414169	0.052071	0.017638922	5.79919982
0.415654	0.063841	0.016988039	5.8336339
0.420653	0.059911	0.015735865	5.889916658
0.385774	0.067741	0.016485929	5.801063538
0.422555	0.061047	0.015494108	5.823215723
0.420575	0.055366	0.015490055	5.956813097
0.389745	0.056604	0.01515007	5.997365236

Table 4: Comparison of Execution Times for Serial (C++), Parallel (16 threads) using C++, PyOMP (16 threads), and Python (Serial) for Computing Pi Using a Definite Integral over 1,00,000,000 steps. Note that the PyOMP time values are without the JIT compilation time.

Program	C++ Parallel (4 threads)	C++ Parallel (16 threads)	PyOMP	Serial (C++)	Serial (Python)
Matrix Multiplication	2.173667	-	-	5.942867	-
Pi Computation	-	0.061568	0.016643667	0.413365	5.869972436
Dot Product	0.013365	-	-	0.043808	-
Count Primes	8.28582	2.779375	-	15.374567	-
Merge Sort	5.270077	2.076299	-	7.947702	-
Running Sum	0.026114	-	-	0.106588	-

Table 5: Comparison of the Average Execution Times for Various Programs in Different modes of execution. The input parameters were same as those described in earlier result tables for individual programs. For PyOMP, 16 threads were used. A "-" indicates that the field was not computed.

3 Message Passing Interface (MPI)

3.1 Introduction

MPI is a specification designed for developers and users of message passing libraries, facilitating data exchange between processes in parallel computing. Originally developed for distributed memory architectures in the 1980s and 90s, MPI has evolved to support various hardware platforms, including distributed memory, shared memory, and hybrid systems.

Advantages of MPI

- **Standardization:** MPI is widely recognized as the standard for message passing, ensuring compatibility across diverse HPC platforms.
- **Portability:** Applications written using MPI require minimal modifications for porting to different platforms.
- **Performance Opportunities:** Vendor-specific MPI implementations optimize performance using hardware-specific features.
- **Functionality:** With over 430 routines defined in MPI-3, the interface provides extensive functionality for parallel programming.
- **Availability:** Multiple implementations, both vendor-supported and open-source, are readily available.

3.2 Environment Management Routines

This group of routines is used for initializing and terminating the MPI environment, querying process information, and obtaining system characteristics. Some commonly used routines include:

- **MPI_Init:** Initializes the MPI execution environment and must be called before any other MPI functions.
- **MPI_Comm_size:** Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD.

- **MPI_Comm_rank:** Returns the rank of the calling MPI process within the specified communicator.
- **MPI_Abort:** Terminates all MPI processes associated with the communicator.
- **MPI_Wtime:** Returns the wall clock time in seconds on the calling processor.
- **MPI_Finalize:** Terminates the MPI execution environment and should be the last MPI routine called in every MPI program.

Note: These routines are essential for initializing MPI, obtaining process information, and managing the MPI environment. Code examples and snippets have been illustrated in the repository provided in Section 5.

3.3 Communication Routines

3.3.1 Point-to-Point Routines

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

Blocking Operations: These operations, such as MPI_Send and MPI_Recv, block the sender or receiver until communication is complete. They ensure synchronous and safe data transfer between processes. Blocking operations are straightforward but may lead to inefficiencies if the sender and receiver are not properly synchronized.

Non-Blocking Operations: MPI also offers non-blocking operations like MPI_Isend and MPI_Irecv, which allow the sender or receiver to continue executing instructions without waiting for the communication operation to complete. These asynchronous operations return immediately after initiation, enabling overlap of communication with computation for improved performance. However, they require careful handling of data to avoid race conditions or data corruption issues.

Order and Fairness: MPI guarantees message order and fairness among processes. However, for non-blocking operations, the user must take care to avoid starvation of any task.

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

Note: For more information, please look here

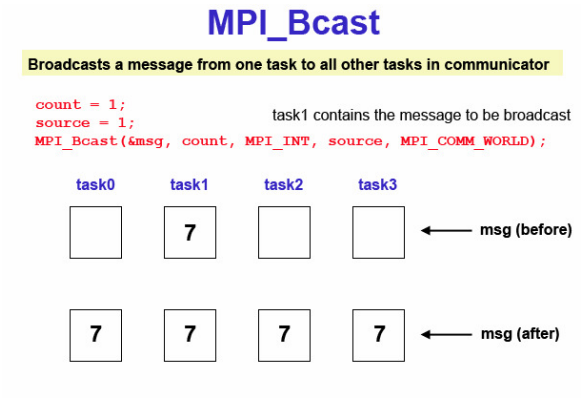
3.3.2 Collective Communication Routines

MPI offers routines for synchronization, data movement, and collective computation among tasks. A few of them have been discussed below -

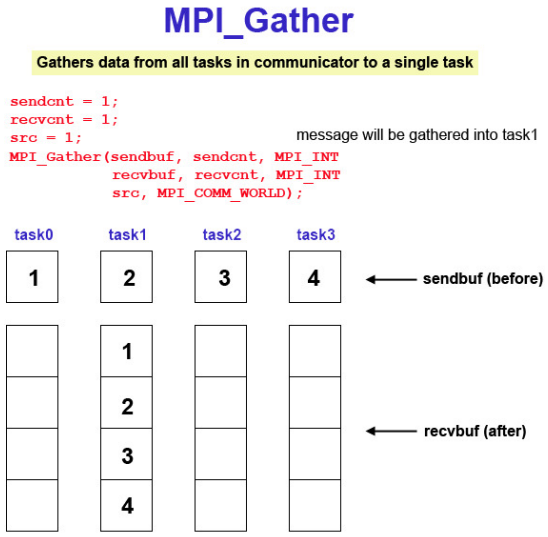
MPI_Barrier: Synchronization operation that creates a barrier.
MPI_Bcast: Broadcasts a message from a root process to all others.
MPI_Scatter: Distributes distinct messages from a source task to each task in the group.

MPI_Gather: Gathers distinct messages from each task to a single destination.
MPI_Reduce: Applies a reduction operation on all tasks and places the result in one task.
MPI_Reduce_scatter: Combines reduction and scatter operations.

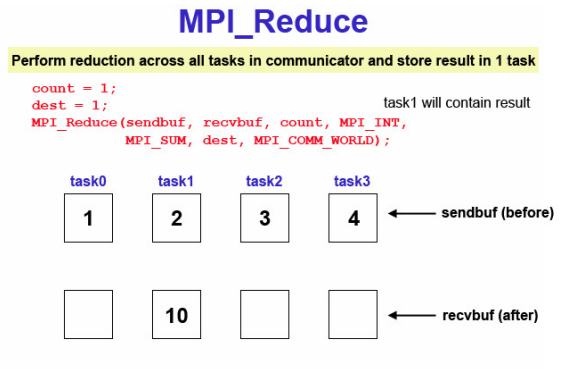
Syntax and illustration -



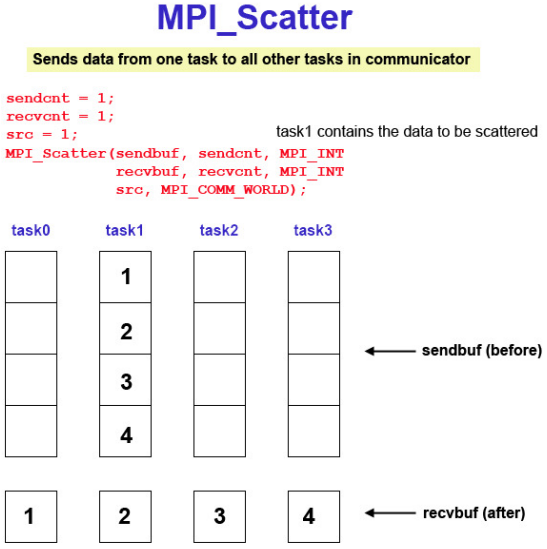
(a) Syntax of the MPI_Bcast routine



(b) Syntax of the MPI_Gather routine



(c) Syntax of the MPI_Reduce routine



(d) Syntax of the MPI_Scatter routine

3.4 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.

There are several methods provided by MPI for constructing derived data types:

Contiguous: Replicates a given data type a specified number of times, resulting in a derived data type where elements are contiguous in memory.

Vector and Hvector: Similar to contiguous types but allow for regular gaps (stride) in displacements, enabling the definition of a strided pattern of elements.

Indexed and Hindexed: Allow specifying a list of blocks with different sizes and displacements, providing more flexibility in defining data layouts.

Struct: Forms a new data type according to a completely defined map of component data types, similar to the C struct type.

MPI provides routines for handling derived data types:

- `MPI_Type_contiguous`
- `MPI_Type_vector` and `MPI_Type_hvector`
- `MPI_Type_indexed` and `MPI_Type_hindexed`
- `MPI_Type_struct`
- `MPI_Type_commit`
- `MPI_Type_free`

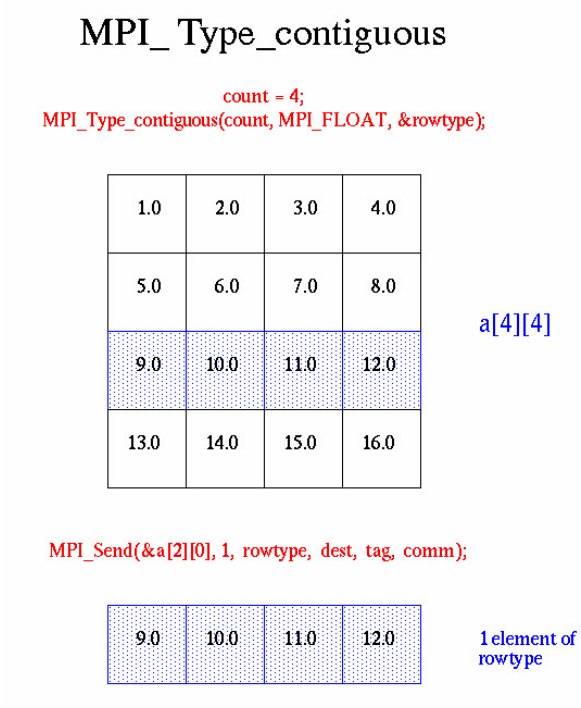
These routines allow for the creation, modification, and deallocation of derived data types. Examples demonstrate how to create derived data types for distributing arrays and structures among MPI processes.

Examples : Kindly refer Fig 4 -

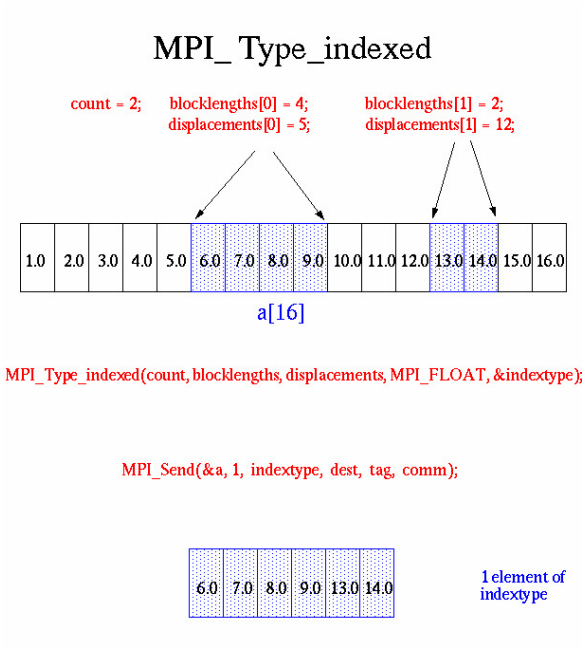
3.5 Virtual Topologies

In MPI, virtual topologies represent how MPI processes are organized into various geometric shapes, such as one-dimensional arrays, two-dimensional grids, or graphs. They facilitate communication patterns and simplify parallel programming by organizing processes effectively.

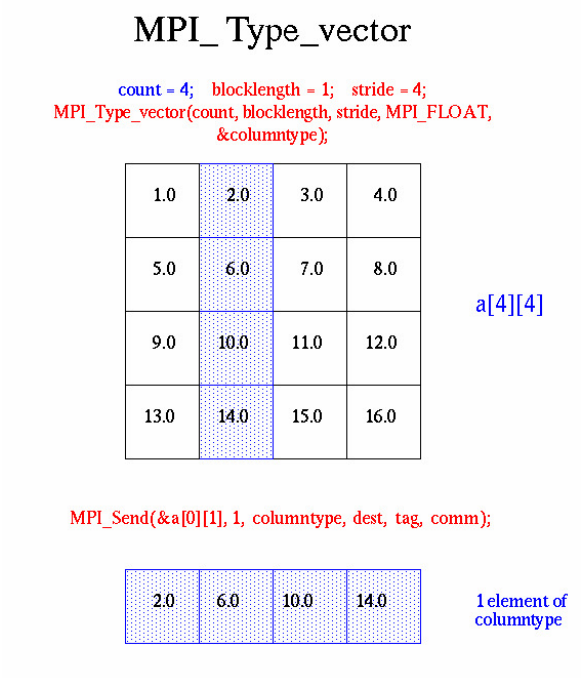
Virtual topologies are constructed based on the logical organization of processes within MPI communicators and groups, rather than reflecting the physical layout of the parallel machine.



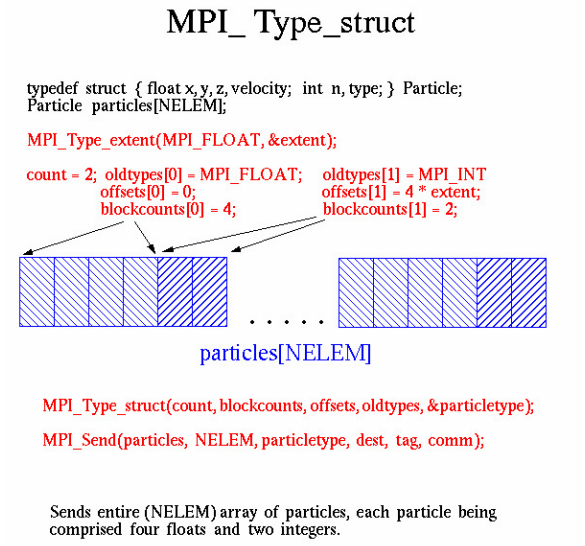
(a) Contiguous data type in MPI



(b) Indexed data type in MPI



(c) Vector data type in MPI



(d) Struct data type in MPI

Figure 4: Derived data types in OpenMPI in C

3.5.1 Why Use Them?

- **Convenience:** Virtual topologies match common communication patterns, making code writing easier. For example, grids are great for neighbor communication.
- **Communication Efficiency:** They speed up communication between processes by organizing them effectively, especially on different hardware architectures.

3.5.2 Types of Virtual Topologies

- **Cartesian (Grid) Topology:** Processes are organized in a grid, suitable for structured communication like neighbor interactions.
- **Graph Topology:** Processes are like dots connected by lines, allowing for flexible communication patterns.

3.6 Example Programs and Results

Below are the results of several codes written using OpenMPI for C, demonstrating a comparison of runtime between serial and parallel implementations.

The sample programs and their descriptions can be found here.

Results - (on the following page)

3.7 OpenMPI in Java

The paper Design and implementation of Java bindings in Open MPI explores the integration of Java bindings within Open MPI, facilitating the development of high-performance computing (HPC) applications in Java. It addresses initial concerns regarding Java's performance overhead for computationally intensive tasks, highlighting recent studies that demonstrate its viability, especially with optimized numerical libraries like MTJ.

Key points discussed in the work include:

1. **Introduction to Java in HPC**: The paper introduces the increasing interest in using Java for HPC, noting its built-in support for threads and efforts to utilize parallel primitives.
2. **Overview of MPI and Java**: The paper provides an overview of MPI, emphasizing its various types of communication and the use of handles to refer to underlying implementation objects. It also explains Java's main features, including bytecode compilation and execution on the JVM.
3. **Integration of MPI with Java**: It emphasizes the preference for adopting the MPI standard in Java due to its rich feature set, particularly in collective communication. The integration of Java MPI bindings within Open MPI is presented as a solution to this preference.

Serial	Parallel (4)	Parallel (8)
0.013394	0.021644	0.029763
0.01329	0.023515	0.033096
0.013784	0.024214	0.041335
0.012766	0.025921	0.032666
0.013458	0.027621	0.036264
0.012756	0.031196	0.03942
0.013484	0.023622	0.03223
0.013195	0.032709	0.030122
0.01328	0.021038	0.02982
0.013762	0.025338	0.033057

(a) Array sum execution times (in seconds)(Array size - 2000000)

Serial	Parallel (4)	Parallel (8)
0.3841	0.0975	0.0843
0.3794	0.099	0.0568
0.3907	0.1116	0.0708
0.4125	0.1016	0.0693
0.3924	0.1132	0.0785
0.3886	0.1378	0.0574
0.3818	0.1204	0.0614
0.389	0.107	0.0677
0.3851	0.1197	0.0784
0.3807	0.1386	0.0729

(b) Matrix Mult execution times (in seconds)(Matrix size - 500×500)

Serial	Parallel (4)	Parallel (8)
0.090473	0.095953	0.133123
0.089791	0.092924	0.122851
0.111547	0.096912	0.128095
0.093388	0.091687	0.133284
0.089141	0.099094	0.126191
0.110896	0.09349	0.130727
0.096676	0.094268	0.131422
0.111731	0.092251	0.134584
0.089925	0.099125	0.127469
0.104073	0.100257	0.140128

(c) Pi calculation using dartboard method (in seconds)

Serial	Parallel (4)	Parallel (8)
0.212334	0.058013	0.045149
0.196966	0.107986	0.042701
0.211369	0.06526	0.044497
0.197061	0.069116	0.052633
0.210145	0.074216	0.046462
0.19613	0.063497	0.04419
0.214646	0.066364	0.042466
0.198862	0.059317	0.058656
0.207433	0.06648	0.051323
0.19631	0.06553	0.043347

(d) Prime finder (in seconds)(Limit - 2500000)

Figure 5: Results - MPI

4. **Java Native Interface (JNI)**: The JNI is discussed as a crucial component for calling native subroutines and libraries from Java, facilitating the integration of optimized numerical kernels.

5. **Open MPI's Java Bindings**: The paper highlights the development of Open MPI's Java bindings, which offer a new implementation that directly integrates into the Open MPI distribution. This eliminates the need for JNI and ensures compatibility with Open MPI while adhering to modern Java practices and the MPI 3.0 specification.

6. **Performance Evaluation**: The performance evaluation results are presented, showcasing the effectiveness of Java MPI bindings in Open MPI for developing parallel applications in Java.

3.7.1 Internal Implementation

JNI Interface - The JNI interface facilitates communication between Java and native code in the Open MPI Java bindings.

```
public class Win{
    ...
    public Group getGroup() throws MPIException{
        MPI.check();
        return new Group(getGroup(handle));
    }
    private native long getGroup(long win) throws MPIException;
    ...
}
```

Native Code - Native code written in C/C++ interfaces directly with the underlying Open MPI library, handling fundamental MPI functionalities.

```
JNIEXPORT jlong JNICALL Java_mpi_Win_getGroup(JNIEnv *env, jobject jthis,
    jlong win) {
    MPI_Group group;
    int rc = MPI_Win_get_group((MPI_Win)win, &group);
    ompi_java_exceptionCheck(env, rc);
    return (jlong)group;
}
```

The above 2 code blocks illustrate the invocation of a JNI method from Java (top) and the corresponding JNI implementation in C (bottom).

Java Classes - Various Java classes represent MPI entities, providing a high-level abstraction for interacting with MPI functionality.

JNI Methods - Java methods annotated with `native` serve as entry points for invoking MPI operations from Java, delegating functionality to native functions.

Error Handling - Mechanisms propagate errors from native code to Java, allowing seamless error handling within Java applications.

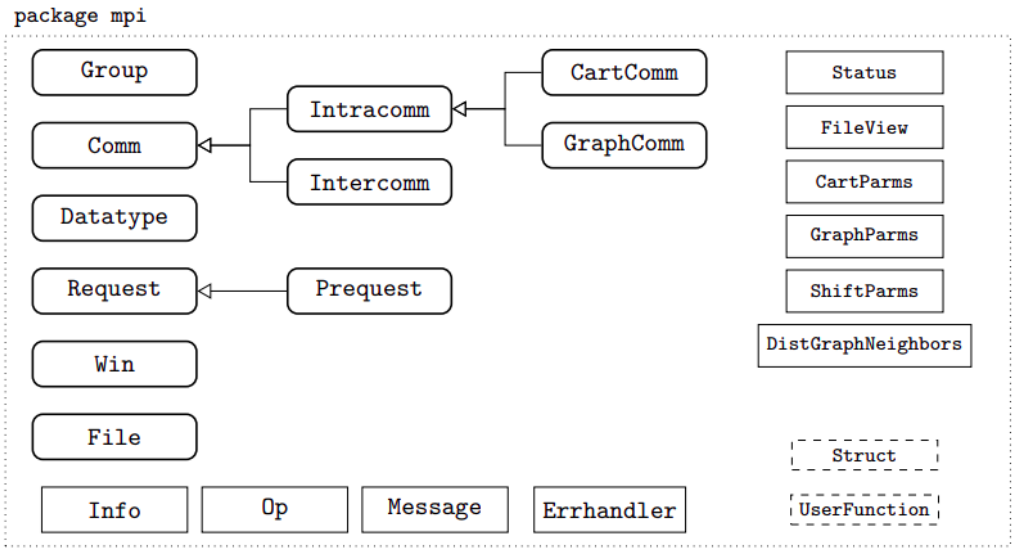


Figure 6: Class hierarchy of the Open MPI Java bindings

```
try {
    File file = new File(MPI.COMM_SELF, "filename", MPI.MODE_RDONLY);
}
catch(MPIException ex) {
    System.err.println("Error Message: "+ ex.getMessage());
    System.err.println(" Error Class: "+ ex.getErrorClass());
    ex.printStackTrace();
    System.exit(-1);
}
```

Resource Management - Efficient management of resources such as memory buffers and MPI communicators is crucial for optimal performance.

Memory Management - Challenges in memory management are addressed using techniques like direct buffers to optimize performance while minimizing interference with garbage collection.

Thread Safety - Synchronization mechanisms ensure thread safety when accessing shared MPI resources across multiple threads.

3.7.2 Communication Routines

3.7.2.1 Point-to-Point Routines -

Send and Receive: Java MPI provides Send and Recv methods for exchanging messages between processes.

Non-blocking Operations: Java MPI supports non-blocking operations like Isend, Irecv, Wait, and Test for asynchronous communication.

3.7.2.2 Collective Routines -

Broadcast: The Bcast method broadcasts data from one process to all other processes in a communicator.

Scatter and Gather: Java MPI provides Scatter and Gather methods for distributing data between processes.

Reduce: The Reduce method aggregates data from multiple processes into one process.

All-to-All: Java MPI provides Allgather, Allreduce, and Alltoall methods for exchanging data between all processes.

3.7.2.3 One-sided Routines -

Java MPI supports one-sided communication operations like Put and Get for accessing remote memory.

3.7.2.4 Intra and Inter Communicators -

Intra-communicators: These include processes within the same MPI job and are typically created using MPI.COMM_WORLD.

Inter-communicators: These involve processes from different MPI jobs and are created using Create or Split methods.

3.7.3 Derived Datatypes

In Open MPI for Java, derived datatypes such as contiguous, vector, and indexed datatypes are defined similarly to their counterparts in C. These datatypes allow for efficient description of non-contiguous data layouts or portions of arrays. Methods like `getSize()`, `getExtent()`, and `getLb()` provide information about the size and layout of derived datatypes, aiding in memory understanding and manipulation.

However, defining struct-like datatypes in Java involves a specialized approach. In Java, a subclass of `Struct` is typically used, where data fields are defined along with corresponding access methods. Once defined, these struct-like datatypes can be instantiated to create objects representing their type, enabling efficient data manipulation.

Here's an example illustrating the definition of a struct-like datatype `Complex` in Java:

```
1  public class Complex extends Struct {
2      // Define offsets of the fields
3      private final int real = addDouble(),
4                        imag = addDouble();
5
6      // Method to create a data object
7      @Override
8      protected Data newData() { return new Data(); }
9
10     // Inner class to define access methods
11     public class Data extends Struct.Data {
12         // Methods to read from the buffer
13         public double getReal() { return getDouble(real); }
```

```
14          public double getImag() { return getDouble(imag); }
15
16          // Methods to write to the buffer
17          public void putReal(double r) { putDouble(real, r); }
18          public void putImag(double i) { putDouble(imag, i); }
19      } // Data
20  } // Complex
```

In C, defining structs involves specifying the layout and types of the fields directly within the struct declaration. The process is more straightforward and does not require the creation of specialized classes.

Despite these differences, the concepts and functionalities of derived datatypes remain consistent between Java and C in Open MPI, providing flexibility and efficiency in data manipulation across both languages.

4 Integrating Parallel Constructs In Other Languages

OpenMP and MPI have become industry standards for implementing parallel programming, but they are primarily defined for C/C++ and Fortran. In this section we will look at how programmers have looked to extend these tools to other programming languages and finally propose some design elements to consider if someone wants to implement these tools in a new programming language.

4.1 OpenMP

OpenMP is widely used for implementing parallel programming constructs in a local environment/cluster and is considered an industry standard. It leverages the native thread functions to perform tasks parallelly . We will explore how OpenMP like tools are being developed in Java and Python.

4.1.1 JOMP : An OpenMP-like Interface for Java

The paper describes a prototype implementation of an OpenMP-like set of directives and library routines implemented completely in java, including the compiler and runtime library that is used in JOMP.

It presents a high level abstraction of the POSIX threads, like how it is in the original implementation and aims to help the programmer write code that is closer to the sequential version of the parallel code and avoids the need to use low level thread functions.

Key Features of JOMP :

1. To make use of the directives, the paper adopts the approach used in Fortran where the directives are embedded into comments , i.e using ”//” .

2. JOMP provides directives for parallel, reduction, sections and section, like how its present in the C/C++ implementation. Example :

```
    //omp parallel shared(a,b)
        {
            //omp for
            for (i=1; i<n; i++){
                b[i] = (a[i] + a[i-1]) * 0.5;
            }
        }
```

3. It also provides other directives like the master, single, critical ,and the barrier directive which have the same functionality as its Fortran counterpart. Also has support for orphan directives.

4. It also provides library functions to get essential metadata like number of threads, max threads, thread number, etc. Example : getNumThreads, setNumThreads(n), getMaxThreads(), getThreadNum(), inParallel() .

JOMP Compiler :

JOMP provides its own compiler to detect the directives written in the code and converts it to the parallelized code which replaces the directives with the corresponding calls to the jomp.runtime class's.

Example of a simple "Hello World" program :
JOMP provides its own compiler to detect the directives written in the code and converts it to the parallelized code which replaces the directives with the corresponding calls to the jomp.runtime class's.

Example of a simple "Hello World" program
JOMP provides its own compiler to detect the directives written in the code and converts it to the parallelized code which replaces the directives with the corresponding calls to the jomp.runtime class's.

Example of a simple "Hello World" program

```
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) ;
        }
    }
}
```

On compiling this code using the JOMP compiler, the code gets converted to :

```
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.OMP.doParallel(__omp_obj_0);
        }
        catch(Throwable __omp_exception) {
            System.err.println("OMP Warning: exception in parallel region
                ");
        }
    }

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

JOMP Runtime :

It provides the necessary functionality to support parallelism by implementing classes that internally call the native Java thread functions. The package **jomp.runtime** contains a library of classes and routines used by compiler-generated code.

Key Features Of JOMP Runtime :

1. The core of the library is the **OMP class** which contains the routines used by the compiler to implement parallelism in terms of Java's native thread model.

2. Each thread has a unique thread id which can be queried using the **static currentThread()** method of the Thread class.

3. During the runtime of the JOMP complied code, a initialization step takes place which consists of 2 parts :

   - **Static initialization** : The OMP class reads the system properties like jomp.threads [number of threads to use during execution], jomp.schedule, etc.
   - **start() method** : This is called when the first parallel block is encountered and it initializes thread specific data , creates a team of threads and the threads are set to be running where they are called based on the scheduler (static or dynamic).

4. Each thread is termed as a task where each task is an instance of the **BusyTask** class that has a go method that contains the code to be executed.

5. **Master-Worker Interaction** : JOMP makes use of the master worker model where one thread manages all the threads in the team.

   - During serial regions, threads pause at the first barrier, waiting for the master thread.
   - The master thread calls **doParallel()**, setting up tasks for each thread and reaching the global barrier.
   - Other threads execute their tasks, and the master executes its own task before reaching the barrier again.

6. **Barrier Class** : It implements the **DoBarrier()** method that takes as a parameter a thread number, and causes the calling thread to block until it has been called the same number of times for each possible thread number.

7. **Reduction Class** : This class is used to implement the reduction directive. It provides methods for the different reductions on like addition , multiplication, etc. A team of threads has a common reducer whose reference is maintained by each thread.

8. **Scheduing** : Scheduling is handled by the LoopData and the Ticketer class based on the required functionality. LoopData class is used for static scheduling and the Ticketer class is used for dynamic scheduling.

The [OMP4J - OpenMP Library For Java] provides an updated implementation of the concepts introduced in JOMP.

4.1.2 Pymp - OpenMP Like Python Programming

Pymp aims to bring OpenMp-like functionality by implementing some of the concepts like a single master thread forking multiple threads, sharing data , synchronizing between threads and destroying threads, etc to python.

When a Pymp Python code hits a parallel region, processes – termed child processes are forked and are in a state that is nearly the same as the “master process.” Here processes are forked instead of threads. The shared memory is referenced by the processes.

The underlying implementation makes use of the ”multiprocessing.py” library which provides process-based parallelism. Pymp is an abstraction/interface that makes it easier to write parallel code.

Key Features of Pymp :

1. **pymp.Parallel(n)** defines the parallel construct , i.e the start of parallel region and number of processes.

2. **Scheduling** : Using **pymp.range** corresponds to the static schedule by returning a complete list of indices, while **pymp.xrange** returns an iterator and corresponds to dynamic scheduling.

3. **Variable Scopes** : The implemented variable scopes are firstprivate, shared and private. All variables that are declared before the pymp.Parallel call are termed as firstprivate.

4. **Sections** : Pymp doesn't provide any explicit implementation for sections and section , but the behaviour can be reproduced by using **pymp.range** and **if-else** conditions.

5. When a parallel region ends, all the child processes exit and only the master process continues. Similar to OpenMp, Pymp also numbers its child processes with the master process having "thread_num 0".

Example Program : This program contains a shared array that all the processes can access and the p.range (equivalent to the for directive) function splits the task statically, i.e batches to execute the logic.

```
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))
```

4.2 MPI

Message Passing Interface (MPI), is a standardized and portable message-passing system designed to function on a wide variety of parallel computers. It is popular for distributed-memory parallel programming in SPMD. We will explore how MPI has been implemented in Python, Java and Ocaml.

4.2.1 MPI For Python - mpi4py

MPI for Python provides Python bindings for the Message Passing Interface (MPI) standard, allowing Python applications to exploit multiple processors on workstations, clusters and supercomputers.

It provides an object oriented approach to message passing which is based on the standard MPI-2 C++ bindings (MPICH package) and is designed to be as close to the C++ MPI implementation as possible.

The mpi4py package requires the installation of the native MPI library (MPICH) which is implemented in C. During installation, the setup process will compile the necessary python bindings and link them with the MPICH libraries. It basically acts as an interface between Python and MPICH.

During execution of a program that uses mpi4py, MPICH is responsible for managing the execution of the MPI processes.

Key Functionalities of MPI implemented :

1. **Types of Communication Support** : It supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communication of a Python object.

2. **Communicating Python Objects** : It uses the pickle and marshal module to serialize(binary format) the Python objects and is sent while sending and restored while receiving.

3. **Communicators** : In MPI for Python, Comm is the base class of communicators. The two predefined intra-communicator instances are available: COMM_SELF and COMM_WORLD. It is for defining processes that are involved in message passing.

   - **Point-To-Point Communication** : Provides a set of send and receive functions allowing the communication of typed data with an associated tag.
   - **Blocking Communication** : Comm.Send, Comm.Recv and Comm.Sendrecv functions can block the caller until the data buffers involved in the communication can be safely reused by the application program.
   - **Non-Blocking Communication** : The Comm.Isend and Comm.Irecv methods provide non blocking communication facility which allows for caller/receiver to continue execution without waiting for completion of communication.
   - **Collective Communication** : Allows the transmittal of data between multiple processes of a group simultaneously.

     The Comm.Bcast, Comm.Scatter, Comm.Gather, Comm.Allgather, Comm.Alltoall methods provide support for collective communications of memory buffers. It also supports reduction operations through the Comm.Reduce, Comm.Reduce_scatter methods.

4. **Environment Management** : The Module functions Init or Init_thread and Finalize provide MPI initialization and finalization respectively. Other environment management functions like Get_processor_name , etc are provided.

4.2.2 OCamlMPI

OCamlMPI provides OCaml bindings for a large subset of MPI functions. The file mpi.mli lists the MPI functions provided by this package.

When a program written with this tool is executed, the following occurs :

1. **Binding Generation** : OCamlMPI provides OCaml bindings for the MPI functions by generating OCaml code that interfaces with the C MPI library. This code defines OCaml functions that wrap the corresponding MPI functions, handling conversions between OCaml data types and C data types as needed.

2. **Compilation** : When the OCamlMPI-based code is compiled, the OCaml compiler (ocamlc or ocamlopt) compiles both the OCaml code and the generated C code into bytecode or native code, respectively.

3. **Linking** : The OCaml compiler links the compiled OCaml code with the MPI library (e.g., MPICH or Open MPI) and any other necessary libraries. This allows the resulting executable to make calls to the MPI functions provided by the native MPI library.

4. **Execution** : When the compiled OCamlMPI program is run, it interacts with the native MPI library to perform message passing and other parallel computing tasks. The native MPI library handles the actual communication between processes running on different nodes in the cluster or computing environment.

Key Functionalities of MPI Implemented :

1. **Communicating Data** : The data is serialized (binary format) by marshaling using the Marshal.to_channel function. This is sent to the receiver who will restore the data.
2. **Mpi.send** : This function sends the data to a given remote process on a communicator with a given tag.
3. **Mpi.receive** : This function receives data from a given remote process on a given communicator with a given tag.
4. **Mpi.comm_rank** : This function determines the rank of the current process on a given communicator. It takes the communicator as input and returns the rank of the current process.
5. **Mpi.comm_size** : This function determines the size (i.e, the number of processes) of a given communicator.
6. **MPI.comm_world** : It defines the number of processes that are involved in message passing. (All specified while running program)
7. **Collective Communication** : This package provides support for broadcast , scatter , gather and reduce.

4.2.3 MPI Java

This package aims to implement MPI as close to the C/C++ implementation as possible. It implements MPI features by using Java wrappers to invoke C MPI calls through the Java Native Interface (JNI).

JNI : The Java Native Interface (JNI) is a programming framework that allows Java code to call and be called by native applications and libraries. Native applications are programs specific to a hardware and operating system platform. Libraries can be written in other languages such as C, C++, and assembly.

The Diagram below shows the important classes present in the mpiJava package.

1. **MPI Class** : Only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator COMM WORLD.
2. **Comm Class** : All the communication functions are a member of or a subclass of this class. It implements the logic of the communicator as seen in MPI.
3. **Datatype Class** : This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions. The diagram below shows a list of datatypes available in MPI Java.

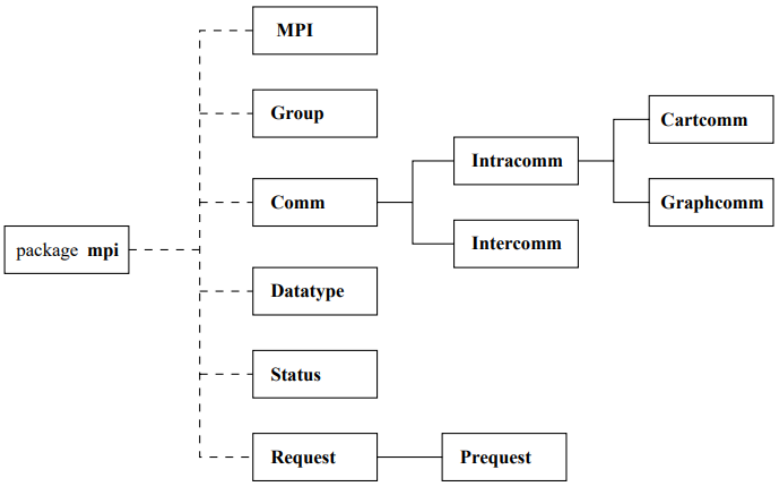


Fig. 1. Principal classes of `mpiJava`

Figure 7: Classes of MPI Java

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.PACKED	

Fig. 2. Basic datatypes of `mpiJava`

Figure 8: Datatypes of MPI Java

Example of a Simple Message Passing Between 2 Processes :

```
import mpi.*;

class Hello {
    static public void main(String[] args){
        MPI.Init(args);
        int myrank = MPI.COMM_WORLD.Rank();

        if(myrank == 0) {
            char[] message = "Hello, there".toCharArray() ;
            MPI.COMM_WORLD.Send(message,0,message.length, MPI.CHAR, 1,
                99);
        }
        else {
            char[] message = new char[20] ;
            MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
            System.out.println("received:" + new String(message) + ":");
        }

        MPI.Finalize();
    }
}
```

MPI Java works by taking the java program with the package imported, then during compilation, the bindings of the package are linked to the native MPI functions that are written in C (inside MPICH package). This linking is done by JNI. So during execution the bytecode corresponding to the MPICH functions are executed. We can see the entire flow of this process in the diagram below.

4.3 Comparing Runtime

We will look at the run times of a serial program and a program with parallel constructs for 2 algorithms :

1. **Matrix Multiplication**
2. **Finding Primes Upto N**

4.4 Design Choices To Implement Tools in New Programming Languages

4.4.1 OpenMP

We have seen the implementation of OpenMP like tools in python and java, i.e Pymp and JOMP respectively. In both these implementations we have noticed that they rely on native thread/processes libraries implemented in the respective language and add an interface/abstract over these libraries.

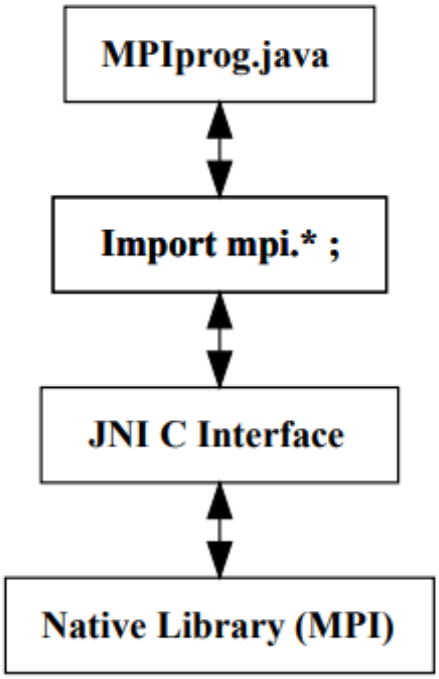


Fig. 4. Software Layers

Figure 9: Architecture of MpiJava

Algo	Tool	Serial	Parallel (4)	Info
1	mpi4py	12.1846	4.0660	Matrix of 500*500
1	pymp	12.229	2.365	Matrix of 500*500
2	pymp	1.2672	0.4637	N=1000000
2	mpi4py	4.6067	1.2685	N=2500000

This interface implements the key features of OpenMP that is present in the original implementation, i.e in C/C++.

JOMP in particular made great efforts to be as close to the OpenMP implementation as possible by providing directives to add parallel blocks in the code. The key contribution from this work is the JOMP compiler that takes the code with directives and converts it to the parallelized code that calls the JOMP runtime functions.

So to implement OpenMP in a new language, the language should have an existing library that provides functions to low level thread functions, scheduling, locks, etc.

A custom compiler can be implemented that takes the code with directives and converts it to the parallelized code that calls the corresponding functions implemented in the Runtime Library.

The Runtime Library has all the functions that implement the features of OpenMP by making use of the native thread functions, other functions that directly interact with the Operating System.

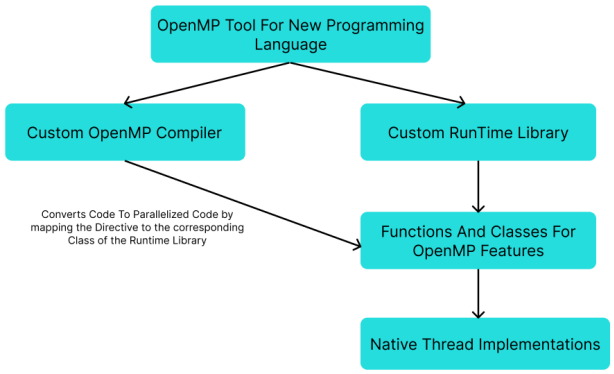


Figure 10: Software Design of OpenMP Like Tool in New Language

4.4.2 MPI

We have seen the implementation of MPI like tools in Python, Java and Ocaml, i.e mpi4py, MPI Java, and OcamlMPI respectively.

In each of the implementations , we saw that the tool provided bindings to MPICH package which is the original implementation of MPI in C/C++. In each of the languages, there is support to link with libraries written in different languages or with byte code. We saw this with Ocamlc and JNI.

So to make a tool that has the features of MPI, the new programming language should have the support like an interface to call and link with code written in other languages, particularly C as the MPICH package is written in C.

An module can be implemented that provides the key features of MPI and its use should be as close to the MPI implementation in C. The module will then use the interface that links the necessary MPI libraries during compilation.

During execution the native MPI library will then handle the actual communication between the processes.

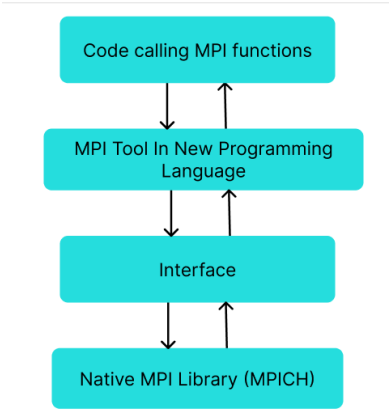


Figure 11: Software Design of MPI Like Tool in New Language

5 Relevant Links

- **Main Repository Link:** here
- **Slide Deck For Presentations:** here

6 Author's Contributions and Work Distribution

Vidhish Trivedi (IMT2021055): (Primarily worked on section 2 of this report, and the OpenMP, Experiments, and PyOMP directories in the repository) Explored OpenMP to understand how parallel programs are implemented by writing sample programs to demonstrate certain algorithms and computations. Investigated how parallel programming can be integrated in languages like Python, specifically looked at PyOMP. Created extensive documentation and guides for parallel programming by incorporating information from multiple sources, and simplifying them. Surveyed how the parallel programming is integrated in a language with focus on syntax, semantics, safety, and error handling. Responsible for setting up the GitHub repository. Wrote code for OpenMP implementations of several programs, which can be found here, did a naive comparison of execution times between serial and parallel versions, available here.

Munagala Kalyan Ram (IMT2021023): (Worked focused predominantly on Section 4 of the report and some parts of MPI). Explored how tools similar to OpenMP and MPI have been developed in other programming languages which include Java, Python and Ocaml. The key features of OpenMP and MPI were documented and a few codes were written to test its working and compared the runtime with the serial code. This study helped in finding common design patterns in the tools implemented in the different languages. An architecture is proposed which gives a programmer an idea on how to approach implementing OpenMP and MPI in a

new programming language/existing language that doesn't have this tool.

Sankalp Kothari (IMT2021028): (Worked focused predominantly on Section 3 of the report, and MPI and OpenMPI for Java sections of the repository) Explored the intricacies of OpenMPI for C to delve into parallel programming concepts. Wrote sample programs to illustrate parallel algorithms and computational tasks, exploiting the features of OpenMPI and conducting comparisons between serial and parallel executions to gauge performance improvements. Created comprehensive documentation by referencing various sources, ensuring clarity on syntax, semantics, safety measures, and error handling within the OpenMPI framework. Worked on comprehending the paper for understanding the implementation of OpenMPI for Java.

References

[1] MPI Java :https://surface.syr.edu/cgi/viewcontent.cgi?article=1006&context=npac

[2] JOMP :https://dl.acm.org/doi/pdf/10.1145/337449.337466

[3] OMP4J - OpenMP Library For Java : https://dl.acm.org/doi/abs/10.5555/3636517.3636530

[4] Pymp : https://www.admin-magazine.com/HPC/Articles/Pymp-OpenMP-like-Python-Programming

[5] MPI For Python - mpi4j : https://mpi4py.readthedocs.io/en/stable/

[6] OcamlMPI :https://github.com/coti/ocamlmpi?tab=readme-ov-file

[7] OpenMPI for Java : https://riunet.upv.es/bitstream/handle/10251/81655/ompi-java.pdf?sequence=3

[8] Official OpenMP Specification: link.

[9] Variadic Templates in C++ (Microsoft): link.

[10] Exception Handling in Concurrency Runtime (Microsoft): link.

[11] Converting an OpenMP loop to use the Concurrency Runtime: link.

[12] Anderson, Todd A., and Tim Mattson. "Multithreaded parallel Python through OpenMP support in Numba." SciPy. 2021. Available here.

[13] OpenMP Common Core: link.

A Appendix: Preprocessor Directives

A.1 What are Preprocessor Directives?

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the *preprocessor*. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

A.2 General Syntax

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

For more information, see: here.

B MPI Call Structure

B.1 Basic MPI Program Structure

Most MPI calls broadly follow the format in the figure :

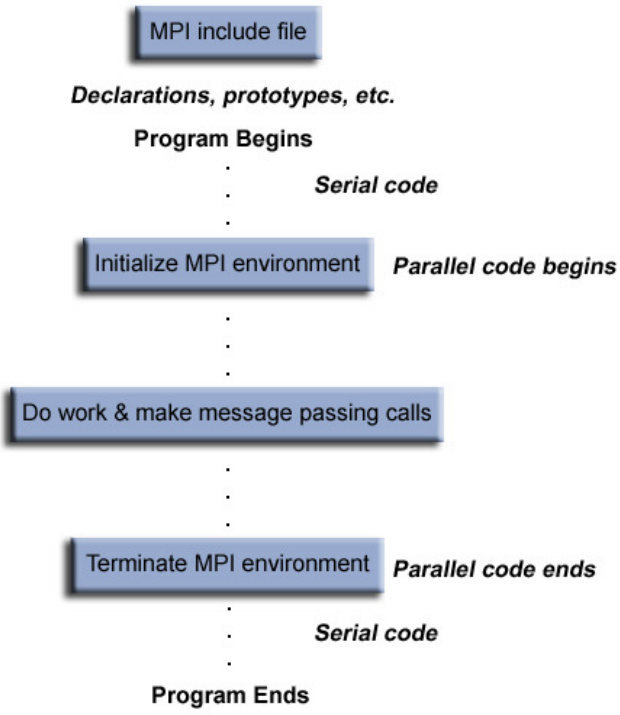


Figure 12: MPI Program Structure. Source : Lawrence Livermore National Laboratory

B.2 MPI Call Structure

MPI calls adhere to a specific format, comprising:

- **Communicators and Groups:** Communicators and groups define the collection of processes permitted to communicate.
- **Rank Assignment:** Each process within a communicator is assigned a unique integer identifier, known as a rank. Ranks are used to identify message sources and destinations.
- **Error Handling:** MPI routines include error codes to handle exceptional situations, with default behavior to abort on error. Custom error handling can be implemented, although error display varies among implementations.

C Further Readings

Interested readers may go through the following topics:

- Amdahl's law - Wikipedia
- Official OpenMP Page