Differentiable Programming

Jash Gautam Makhija (A59025270)
Animesh Kumar (A59023817)

Distributed AD: Add OpenMPI support to loma so that it can be distributed to multiple machines.

Abstract

The primary goal of our project is to enhance `loma` by integrating support for OpenMPI (Open Message Passing Interface)[1]. This integration will enable `loma` to distribute and parallelize the compilation process across multiple machines, significantly improving compilation speed and efficiency. By splitting a very large dataset across several machines, we can effectively test various inputs for a particular function, leveraging the computational power of distributed systems.

Loma OpenMPI Extension Architecture[1]

The OpenMPI extension of Loma is supported by the below implementation:

1. The parser codebase has been extended to support another decorator @openMPI. If this decorator is found the parser sets a parameter as is_openMPI as true in the FunctionDef.
2. The compiler codebase has been modified if a loma FunctionDef is found. And generates two different C files one which acts as a parent the other which acts as a child.
3. The parent codebase is compiled as a DLL(shared library) and can be loaded easily in python. The main purpose of the parent is to spawn n(present as a parameter to the parent function) child workers which will actually execute the function. Additionally the parent will send all the required parameter to the the child process and gather and receive the output from the child process. This gathered output is what is sent back to the python interface.
4. The child codebase is compiled as a regular object file. This is necessary as OpenMPI parallization can only be invoked in files which have a main function. This child codebase additionally contains a wrapper on top of the concerned function which receives the parameters from the parent and send back the response to the parent.

Figure 1 decribes this implementation briefly. The generated parent and child codebase are present in Listing 7.

[1]OpenMPI integration code: https://github.com/animeshk08/loma_public/tree/final_project

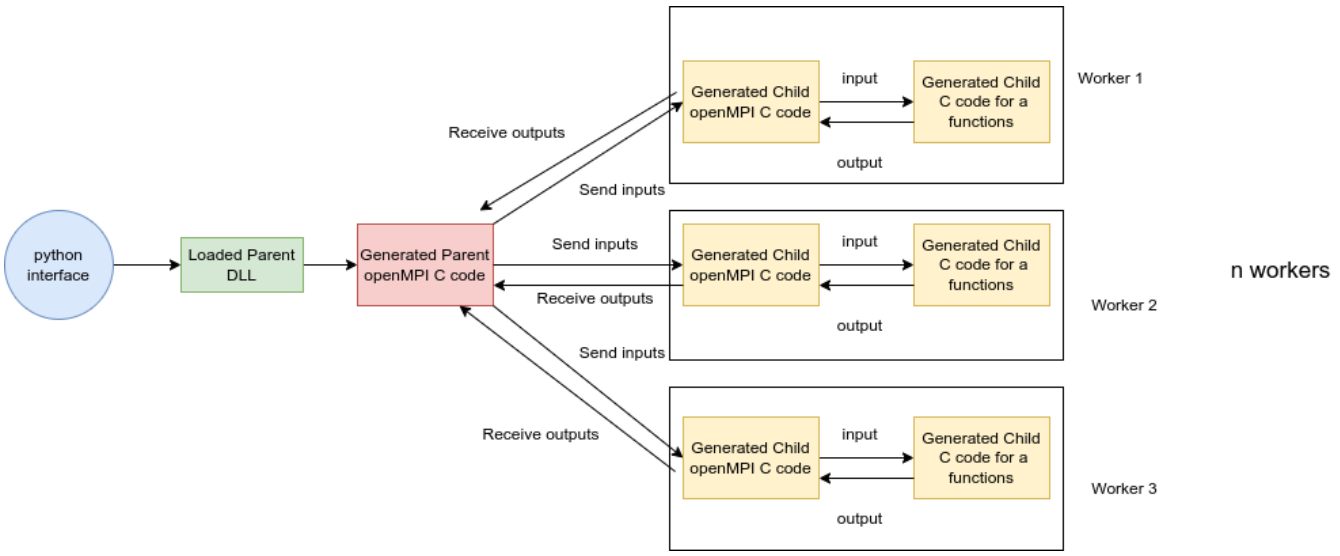

Figure 1: Open MPI implementation in Loma. The loma compiler generates two Open MPI based C code. One DLL which acts as a parent and and spawns multiple child workers. These child workers are the seconds generated files which contains the an OpenMPI wrapper on top of the actual generated C function.

Listing 1: Forward Differentiation Array Input Loma code

```
@openMpi
def array_input(x : In[Array[float]]) -> float:
    return x[0] + x[1]

d_array_input = fwd_diff(array_input)
```

Listing 2: Python Testcase

```
    def test_array_input(self):
        with open('loma_code/array_input.py') as f:
            structs, lib = compiler.compile(f.read(),target = 'openMpi',
    output_filename = '_code/array_input')
        _dfloat = structs['_dfloat']
        num_worker = 2
        px_x = [_dfloat(0.7, 0.8), _dfloat(0.3, 0.5),_dfloat(0.3, 0.4), _dfloat(0.4,
     0.6)]
        py_size = [2,2]
        input = (_dfloat * len(px_x))(*px_x)
        input2 = (ctypes.c_int * len(py_size))(*py_size)
        py_y = [_dfloat(0, 0)] * num_worker
        out = (_dfloat * len(py_y))(*py_y)
        lib.mpi_runner(input, input2, out,num_worker)
        result = []
        for i in range(0,num_worker+1,2):
            result.append((px_x[i].val + px_x[i+1].val,px_x[i].dval + px_x[i+1].dval
    ))

        print(result)
```

```
        for worker in range(num_worker):
            flag = False
            print(f"Result from Worker {worker+1}: val = {out[worker].val}, dval = {out[worker].dval}")
            for res in result:
                if abs(out[worker].val - res[0]) < epsilon and abs(out[worker].dval - res[1]) < epsilon:
                    flag = True
                    break
            assert flag
```

Listing 3: Generated OpenMpi Code for forward diff

```

Forward differentiation of function d_array_input:
def d_array_input(x : In[Array[_dfloat]]) -> _dfloat:
        return make__dfloat((((x)[(int)(0)]).val) + (((x)[(int)(1)]).val),(((x)[(int)(0)]).dval) + (((x)[(int)(1)]).dval))

Generated C code for child:

#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
        float val;
        float dval;
} _dfloat;
float array_input(float* x);
_dfloat d_array_input(_dfloat* x);
_dfloat make__dfloat(float val, float dval);
float array_input(float* x) {
        return ((x)[(int)(0)]) + ((x)[(int)(1)]);
}
_dfloat d_array_input(_dfloat* x) {
        return make__dfloat((((x)[(int)(0)]).val) + (((x)[(int)(1)]).val),(((x)[(int)(0)]).dval) + (((x)[(int)(1)]).dval));
}
void d_array_input_mpi_worker(){

    MPI_Init(NULL, NULL);

    MPI_Comm parent_comm;
    MPI_Comm_get_parent(&parent_comm);

    if (parent_comm == MPI_COMM_NULL) {
        MPI_Finalize();
        return;
    }

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int x_size;
MPI_Recv(&x_size, 1, MPI_INT, 0, 0, parent_comm,MPI_STATUS_IGNORE);
_dfloat* x = (_dfloat*)malloc(x_size * sizeof(_dfloat));MPI_Recv(x, x_size * sizeof(
    _dfloat), MPI_BYTE, 0, 0, parent_comm,MPI_STATUS_IGNORE);
_dfloat out = d_array_input(x);
                MPI_Send(&out, sizeof(_dfloat), MPI_BYTE, 0, 0, parent_comm);

MPI_Finalize();}

int main() {
    d_array_input_mpi_worker();
    return 0;
}
_dfloat make__dfloat(float val, float dval) {
        _dfloat ret;
        ret.val = 0;
        ret.dval = 0;
        (ret).val = val;
        (ret).dval = dval;
        return ret;
}

Generated C code for parent:

#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
        float val;
        float dval;
} _dfloat;
void mpi_runner(_dfloat* x,int* x_size, _dfloat* output, int __total_work) {

    MPI_Init(NULL, NULL);

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank == 0) {
        // This is the parent process
        MPI_Comm child_comm;
        MPI_Info info;
        MPI_Info_create(&info);
        // Spawn NUM_CHILDREN child processes
        MPI_Comm_spawn("_code/array_input.o", MPI_ARGV_NULL, __total_work, info, 0,
    MPI_COMM_SELF, &child_comm, MPI_ERRCODES_IGNORE);
        int* array_end = (int*)malloc(1 * sizeof(int));
        int end = 0;
        int array_index = 0;
        for (int i = 0; i< 1;i++){
            array_end[i] = 0;
```

```
        }
        for (int i = 0; i < __total_work; i++) {
            array_index = 0;
            // Send input to child process
            MPI_Send(&x_size[i], 1, MPI_INT, i, 0, child_comm);

                _dfloat* x_i = (_dfloat*)malloc(x_size[i] * sizeof(_dfloat));

                end = array_end[array_index];
                for (int j = 0; j < x_size[i]; j++) {
                    x_i[j] = x[end++];
                }
                array_end[array_index] = end;
                array_index = array_index + 1;
                MPI_Send(x_i, sizeof(_dfloat)*x_size[i], MPI_BYTE, i, 0, child_comm)
    ;
}

            // Communicate with children and receive responses
            for (int i = 0; i < __total_work; i++) {
                _dfloat local_output;
                MPI_Recv(&local_output, sizeof(_dfloat), MPI_BYTE, i, 0, child_comm,
     MPI_STATUS_IGNORE);

                output[i] = local_output;
            }
            MPI_Info_free(&info);
        }
        MPI_Finalize();
    }
```

Listing 4: Result

```
Result from Worker 1: val = 1.0, dval = 1.2999999523162842
Result from Worker 2: val = 0.7000000476837158, dval = 1.0
```

Listing 5: Reverse Differentiation Plus Loma code

```
@openMpi
def plus(x : In[float], y : In[float]) -> float:
    return x + y

d_plus = rev_diff(plus)
```

Listing 6: Python Testcase

```
   def test_plus_rev(self):
        with open('loma_code/plus_rev.py') as f:
            structs, lib = compiler.compile(f.read(),target = 'openMpi',
    output_filename = '_code/plus_rev')
        num_worker = 2
        py_x  = [ctypes.c_float(0.0), ctypes.c_float(0.0)]
```

```
        py_y  = [ctypes.c_float(0.0), ctypes.c_float(0.0)]
        x = [5.0,1.0]
        y = [6.0,4.0]
        dout =  [3.0,4.0]
        arg_dx = (ctypes.c_float * len(py_x))(*py_x)
        arg_x = (ctypes.c_float * len(x))(*x)
        arg_dy = (ctypes.c_float * len(py_y))(*py_y)
        arg_y = (ctypes.c_float * len(y))(*y)
        arg_dreturn = (ctypes.c_float * len(dout))(*dout)
        lib.mpi_runner(arg_x,arg_dx,arg_y,arg_dy,arg_dreturn, num_worker)
        for i in range(num_worker):
            print(f"Result from Worker {i+1}: {arg_dx[i]}, {arg_dy[i]}")
            assert abs(arg_dx[i] - dout[i]) < epsilon and  abs(arg_dy[i] - dout[i])
   < epsilon
```

Listing 7: Generated OpenMpi Code for reverse differentiation

```
    Reverse differentiation of function d_plus:
def d_plus(x : In[float], _dx : Out[float], y : In[float], _dy : Out[float],
    _dreturn : In[float]) -> void:
        tape : Array[float, 1]
        tape_ptr : int = (int)(0)
        _adj_0 : float
        _adj_1 : float
        _adj_2 : float
        _adj_3 : float
        _adj_4 : float
        _dx = (_dx) + (_dreturn)
        _dy = (_dy) + (_dreturn)

Generated C code for child:

#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
        float val;
        float dval;
} _dfloat;
float plus(float x, float y);
void d_plus(float x, float* _dx, float y, float* _dy, float _dreturn);
_dfloat make__dfloat(float val, float dval);
float plus(float x, float y) {
        return (x) + (y);
}
void d_plus(float x, float* _dx, float y, float* _dy, float _dreturn) {
        float tape[1];
        for (int _i = 0; _i < 1;_i++) {
                tape[_i] = 0;
        }
        int tape_ptr = (int)(0);
        float _adj_0;
```

```
37          _adj_0 = 0;
38          float _adj_1;
39          _adj_1 = 0;
40          float _adj_2;
41          _adj_2 = 0;
42          float _adj_3;
43          _adj_3 = 0;
44          float _adj_4;
45          _adj_4 = 0;
46          (*_dx) = ((*_dx)) + (_dreturn);
47          (*_dy) = ((*_dy)) + (_dreturn);
48 }
49 void d_plus_mpi_worker(){
50
51     MPI_Init(NULL, NULL);
52
53     MPI_Comm parent_comm;
54     MPI_Comm_get_parent(&parent_comm);
55
56     if (parent_comm == MPI_COMM_NULL) {
57         MPI_Finalize();
58         return;
59     }
60
61     int world_rank, world_size;
62     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
63     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
64     float x;
65 MPI_Recv(&x, 1, MPI_FLOAT, 0, 0, parent_comm,MPI_STATUS_IGNORE);
66 float _dx;
67 float y;
68 MPI_Recv(&y, 1, MPI_FLOAT, 0, 0, parent_comm,MPI_STATUS_IGNORE);
69 float _dy;
70 float _dreturn;
71 MPI_Recv(&_dreturn, 1, MPI_FLOAT, 0, 0, parent_comm,MPI_STATUS_IGNORE);
72 d_plus(x, &_dx, y, &_dy, _dreturn);MPI_Send(&_dx, 1, MPI_FLOAT, 0, 0, parent_comm);
73 MPI_Send(&_dy, 1, MPI_FLOAT, 0, 0, parent_comm);
74
75 MPI_Finalize();}
76
77 int main() {
78     d_plus_mpi_worker();
79     return 0;
80 }
81 _dfloat make__dfloat(float val, float dval) {
82          _dfloat ret;
83          ret.val = 0;
84          ret.dval = 0;
85          (ret).val = val;
86          (ret).dval = dval;
87          return ret;
88 }
89
90 Generated C code for parent:
91
```

```
#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
        float val;
        float dval;
} _dfloat;
void mpi_runner(float* x, float* _dx, float* y, float* _dy, float* _dreturn, int __total_work) {

    MPI_Init(NULL, NULL);

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank == 0) {
        // This is the parent process
        MPI_Comm child_comm;
        MPI_Info info;
        MPI_Info_create(&info);
        // Spawn NUM_CHILDREN child processes
        MPI_Comm_spawn("_code/plus_rev.o", MPI_ARGV_NULL, __total_work, info, 0, MPI_COMM_SELF, &child_comm, MPI_ERRCODES_IGNORE);
        int* array_end = (int*)malloc(0 * sizeof(int));
        int end = 0;
        int array_index = 0;
        for (int i = 0; i< 0;i++){
            array_end[i] = 0;
        }
        for (int i = 0; i < __total_work; i++) {
            array_index = 0;
            // Send input to child process
            MPI_Send(&x[i], 1, MPI_FLOAT, i, 0, child_comm);
MPI_Send(&y[i], 1, MPI_FLOAT, i, 0, child_comm);
MPI_Send(&_dreturn[i], 1, MPI_FLOAT, i, 0, child_comm);
}

            // Communicate with children and receive responses
            int* rec_end = (int*)malloc(0 * sizeof(int));
            int rend = 0;
            int rec_array_index = 0;
            for (int i = 0; i< 0;i++){
                rec_end[i] = 0;
            }
            for (int i = 0; i < __total_work; i++) {
                rec_array_index = 0;
                float _temp__dx;MPI_Recv(&_temp__dx, 1, MPI_FLOAT, i, 0, child_comm, MPI_STATUS_IGNORE);
_dx[i] = _temp__dx;
float _temp__dy;MPI_Recv(&_temp__dy, 1, MPI_FLOAT, i, 0, child_comm, MPI_STATUS_IGNORE);
_dy[i] = _temp__dy;
```

```
143
144             }
145             MPI_Info_free(&info);} MPI_Finalize();}
```

Listing 8: Result

```
1 Result from Worker 1: 3.0, 3.0
2 Result from Worker 2: 4.0, 4.0
```

Implementation

To enable Loma to handle OpenMPI operations effectively, we made several significant modifications to the code base. These enhancements are detailed below:

1. **`Codegen_openMPI.py`**: This introduces a sophisticated architecture for managing OpenMPI operations across parent and child nodes.
   (a) A parent-child structure was developed. The parent function is responsible for the following:
      i. Spawning Children: It initiates child processes according to the user specified number of workers.
      ii. Data Distribution: Inputs from the test case are broken up and forwarded to the respective child nodes.
      iii. Result Aggregation: It aggregates the results received from child nodes and returns them to the test case.

      The parent node acts as a conduit between the test case and the child nodes, relaying parameters to the children and gathering results for return to the test case.
   (b) The parent node incorporates the main OpenMPI main function, which processes all function parameters. Key functionalities include:
      i. Parameter Handling: Each parameter is transmitted as an array containing values for each child node.
      ii. Array Management: When an array is transmitted, it is amalgamated into a single large array. An additional array indicating the size of each child's data portion is also sent.
      iii. Data Distribution: During the spawning of child nodes, the parent node splits the large array into chunks based on the size array and sends these chunks to the corresponding child nodes.
   (c) At the child nodes:
      i. Data Reception: The nodes receive their respective data portions.
      ii. Function Execution: They invoke the main function with the received input values.

iii. Result Transmission: The results/output of forward differentiation or the necessary variables for reverse differentiation are sent back to the parent node.

(d) The parent node performs the following:

i. Data Collection: It collects outputs from all child nodes.

ii. Aggregation: The results are aggregated into a comprehensive array, which is then sent back to the test case.

(e) Special attention is given to the output of arrays:

i. Flattening: Arrays from all child nodes are flattened into a single array.

ii. Reassembly: At the test case level, the array is split back into its original segments based on the size information and the results are displayed accordingly.

(f) Special Consideration:

i. Data Types: Careful handling of various data types such as float and integers is implemented.

ii. Multiple Outputs: For multiple outputs, especially arrays, the system considers size values to split and combine arrays properly.

iii. In scenarios with array outputs, the parent function aggregates values sequentially from the children; in other cases, the order of reception is randomized to optimized for speed.

2. **Python Test Case's**: The Python test cases are adapted to support multiple value transmissions and parallel processing:

(a) Worker Specification: For functions like `array_input`, multiple test inputs are processed in parallel. For instance, if the number of workers is set to 2, then two different test input configurations can be tested in parallel.

(b) Array Handling: Inputs for each worker are combined into a single array, with an accompanying size parameter specifying each child worker's array size.

(c) Output Validation: If arrays are outputted, the parent's output is split according to size parameters to validate correctness.

(d) Expected Results: Expected outcomes are computed and stored in a results array. The output from the parent is checked against these expected results for correctness.

(e) We tested our program against 30 test cases for forward and reverse differentiation respectively.

Experiment and Observation

To evaluate the performance and benefits of our OpenMPI-based automatic differentiation system, we conducted two key experiments:

1. **Generality and Efficacy of Our Implementation:**

(a) **Objective:** Demonstrate the robustness and versatility of our system by optimizing a third-order multivariate polynomial using a gradient descent routine, where the gradient is computed via automatic differentiation.

(b) **Implementation:**

i. **Task Distribution:** The task is distributed across 2 worker nodes.

- **Node Responsibilities:**
  - One worker node computes the gradient in the x-direction.
  - The other worker node computes the gradient in the y-direction.
- **Gradient Calculation:** Each node performs gradient calculations independently.
- **Position Update:** The results from both nodes are aggregated to update the position in the optimization space, moving towards the minimum of the loss function.

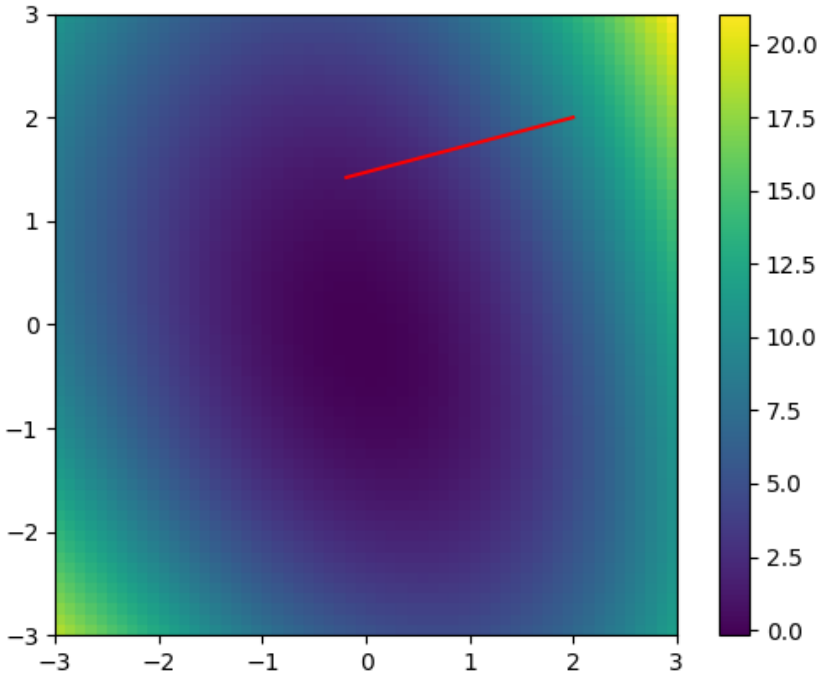

Figure 2: Optimization of a third-order multivariate polynomial using Loma with OpenMPI. The loss function landscape and the optimization trajectory (red curve) show the system starting from (2,2) and progressing towards the center where the loss is minimized.

(c) **Results:**

- **Figure 2**: Illustrates the optimization trajectory overlaid on the loss function landscape. The red curve indicates the movement from the initial point $(2, 2)$ towards the center, achieving a lower loss.

(d) **Limitation:** The current implementation limits the parent process to spawn only once, preventing multiple iterations within a gradient descent loop. Consequently, the system computes gradients and updates positions only for a single iteration. This limitation arises from the restriction that the `lib.mpi_runner` function cannot be invoked repeatedly within a loop to dynamically update parameters.

2. **Performance Evaluation: OpenMPI vs. Sequential Execution**

(a) **Objective:** Assess the performance gains achieved by our OpenMPI-based implementation compared to a traditional sequential execution approach.

(b) **Experimental Setup:**

i. **Test Scenario 1: Varying the number of input values for a single test case to measure execution time.**

- **Methodology:**
  - **OpenMPI Implementation:** Deployed 100 worker nodes to handle input data in parallel.
  - **Sequential Implementation:** Employed a standard loop to process each input test case individually.
- **Results:**
  - **Figure 3**: Shows the time comparison between OpenMPI and sequential execution for varying input sizes. Initially, the sequential method is faster due to the overhead associated with initializing MPI processes. However, as input size scales up to 10^8, OpenMPI outperforms due to its ability to handle large datasets more efficiently by distributing tasks across multiple nodes.

ii. **Test Scenario 2: Evaluating performance for varying iteration counts in a while loop.**

- **Methodology:**
  - **OpenMPI Implementation:** Distributed tasks across 100 worker nodes for each iteration.
  - **Sequential Implementation:** Executed iterations in a conventional loop.
- **Results:**
  - **Figure 4**: Demonstrates that for small iteration counts, both approaches have comparable performance. As the iteration count increases, OpenMPI gains a performance edge due to its ability to parallelize the workload effectively.

The experiments were run on a Apple Macbook Pro (3.2 GHz Octa-Core Apple M1 Pro) with 16 GB RAM. These results underscore the potential of integrating OpenMPI with automatic differentiation

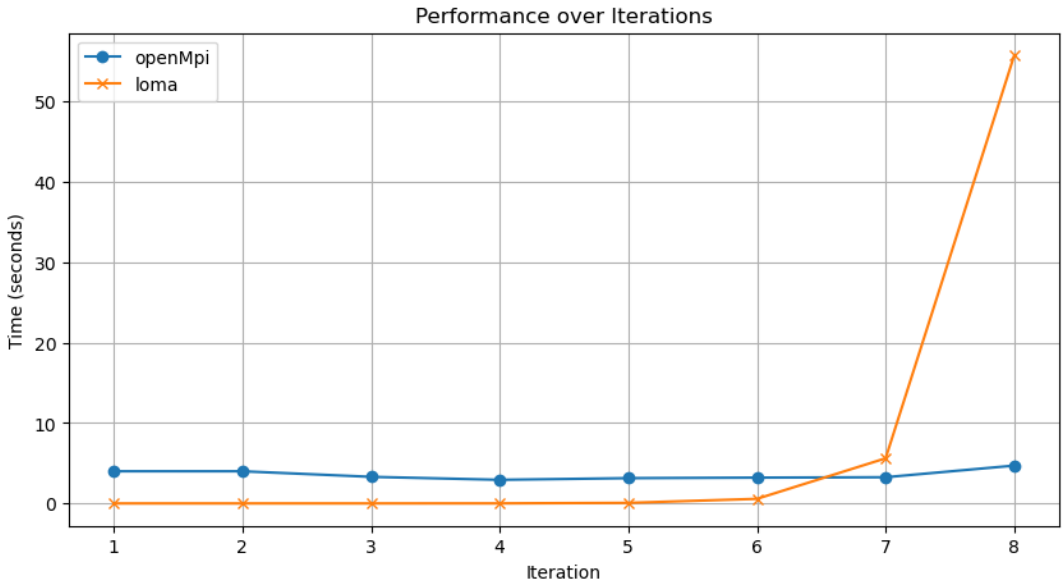

Figure 3: Time comparison for executing varied input sizes between OpenMPI and a traditional sequential approach. The graph indicates that OpenMPI's time advantage becomes significant as input size increases.

frameworks to enhance computational efficiency and scalability for various applications in scientific computing and machine learning.

Limitation

We have created a simple and straightforward OpenMPI extension to the Loma Compiler. However, there are still a few limitations which are listed below:

1. **Single Function OpenMPI Conversion**: The OpenMPI extension generates two C compiled files to be created, a regular compiler object file for the child which includes the actual function call wrapped in an OpenMPI environment wrapper, and a second, a parent which is compiled as a DLL and which spawns the child code. Thus, the child needs to have a main function and the parent needs to know which child object file to call. Such tight coupling limited our current implementation to convert only a single method to be compiled by the OpenMPI compiler. A possible fix for this can include creating multiple child object files and including the name of the same as a parameter to the parent code base.

2. **Single Execution Environment**: Since the OpenMPI environment can only be "initialized" and "finalized" once in a process, the current OpenMPI extension can run only one single instance of an execution. This means that current multiple OpenMPI test cases cannot run together or multiple calls to the same OpenMPI based parent-child framework are also not supported. A possible fix for this is to invoke the OpenMPI executions as separate processes within the Python-based invocation mechanism.

3. **Limited Support for Arrays**: The current implementation of OpenMPI struggles with arrays as input, since in such cases the actual input to the OpenMPI wrapper will be multiple

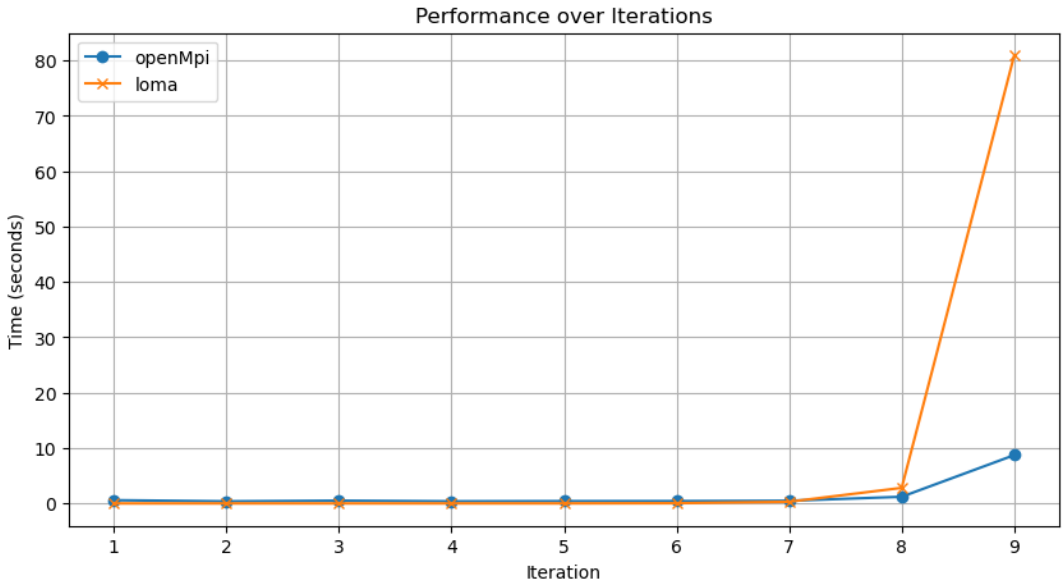

Figure 4: Time comparison for executing varied iteration counts in a while loop between OpenMPI and a traditional sequential approach. OpenMPI exhibits increasing efficiency with higher iteration counts.

such arrays, or a 2-D array. Since 2D arrays are hard to compile using the `ctypes` library, the OpenMPI extension suffers from the same problems. The current implementation flattens the 2D array and passes another list of delimiter indexes to unflatten the array later. Another possible way could be to convert the 2D array to an array of structs. Nevertheless, these approaches are obviously not scalable and hence we claim only limited support for arrays in our implementation.

Result

In this project, we have completed a simple, straightforward, and comprehensible extension to the Loma compiler to support distributed AD using OpenMPI. As part of this project, we have achieved the following milestones:

1. Created an innovative way to compile and spawn parent-child based MPI interface.
2. Supported native message passing between parent and child nodes.
3. Supported backwards compatibility with existing Loma test cases to compile and run in an OpenMPI environment.
4. Demonstrated a simple real-world example (single step of gradient descent) to show applications and advantages of OpenMPI based distributed AD.
5. Conducted a simple benchmark analysis to compare the performance of native Loma compiled C code and OpenMPI extension compiled C code.

References

[1] Open MPI. https://github.com/open-mpi/ompi. 2004.