CSE 291: Differentiable Programming
Final Project: Checkpoint

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 easilt 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 3.

[1]OpenMPI integration code: https://github.com/animeshk08/loma_public/tree/checkpoint1

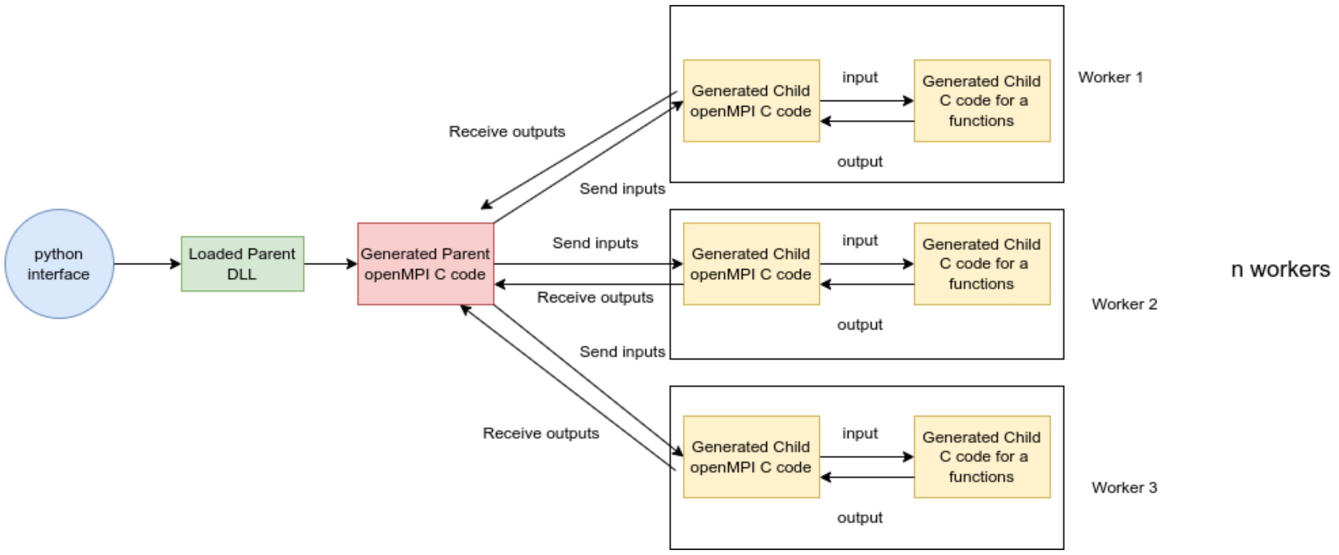

Figure 1: Open MPI implementation in Loma. The loma compiler generates two Open MPI based C code. One DLL which acts as a parent and and spawns multiple child workers. These child workers are the seconds generated files which contains the an OpenMPI wrapper on top of the actual generated C function.

Listing 1: Plus Loma code

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

d_plus = fwd_diff(plus)
```

Listing 2: Python Testcase

```
def test_plus(self):
    with open('loma_code/plus.py') as f:
        structs, lib = compiler.compile(f.read(),target = 'openMpi',output_filename
    = '_code/plus')
    _dfloat = structs['_dfloat']
    num_worker = 4
    px_x = [_dfloat(5.0, 0.5), _dfloat(6.0, 0.3),_dfloat(7.0, 0.2),_dfloat(8.0, 0.1)
    ]
    px_y = [_dfloat(6.0, 1.5), _dfloat(7.0, 2.5),_dfloat(8.0, 3.5),_dfloat(9.0, 4.5)
    ]
    input1 = (_dfloat * len(px_x))(*px_x)
    input2 = (_dfloat * len(px_y))(*px_y)
    py_y = [_dfloat(0, 0)] * num_worker
    out = (_dfloat * len(py_y))(*py_y)
    lib.mpi_runner(input1,input2,out,num_worker)
    result = [(11,2),(13,2.8),(15,3.7),(17,4.6)]

    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_plus:
def d_plus(x : In[_dfloat], y : In[_dfloat]) -> _dfloat:
        return make__dfloat(((x).val) + ((y).val),((x).dval) + ((y).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 plus(float x, float y);

_dfloat d_plus(_dfloat x, _dfloat y);

_dfloat make__dfloat(float val, float dval);

float plus(float x, float y) {
    return (x) + (y);
}

_dfloat d_plus(_dfloat x, _dfloat y) {
    return make__dfloat(((x).val) + ((y).val),((x).dval) + ((y).dval));
}

void d_plus_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);

    _dfloat x;
    MPI_Recv(&x, sizeof(_dfloat), MPI_BYTE, 0, 0, parent_comm,MPI_STATUS_IGNORE);

    _dfloat y;
    MPI_Recv(&y, sizeof(_dfloat), MPI_BYTE, 0, 0, parent_comm,MPI_STATUS_IGNORE);

    _dfloat out = d_plus(x, y);
    MPI_Send(&out, sizeof(_dfloat), MPI_BYTE, 0, 0, parent_comm);
    MPI_Finalize();
}

int main() {
    d_plus_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, _dfloat * y, _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/plus.o", MPI_ARGV_NULL, __total_work, info, 0,
   MPI_COMM_SELF, &child_comm, MPI_ERRCODES_IGNORE);

        for (int i = 0; i < __total_work; i++) {
            // Send input to child process
            MPI_Send(&x[i], sizeof(_dfloat), MPI_BYTE, i, 0, child_comm);
            MPI_Send(&y[i], sizeof(_dfloat), 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, MPI_ANY_SOURCE, 0,
    child_comm, MPI_STATUS_IGNORE);
            output[i] = local_output;
        }

        MPI_Info_free(&info);
    }
    MPI_Finalize();
}
```

Listing 4: Result

```
Result from Worker 1: val = 13.0, dval = 2.799999952316284
Result from Worker 2: val = 15.0, dval = 3.700000047683716
Result from Worker 3: val = 11.0, dval = 2.0
Result from Worker 4: val = 17.0, dval = 4.599999904632568
```

Checkpoint 1 Results and Implementation

Till now we have setup the OpenMPI framework integration in Loma and completed the validation of 17 test cases for the forward differentiation module.
To integrate OpenMPI into the current pipeline, the following changes have been implemented:

1. **Decorator Addition**: In the forward differentiation test cases, the decorator `@OpenMpi` was added at the beginning of the function definitions.
2. **MPI Integration in testcase:** The parameters of the function are now each an array with every entry as the value that should be computed in a worker. These paramters along with an output array and total number of workers is sent to the mpi runner function, which resides in the parent code. The parent process then forwards the input parameter at i^{th} index to the i^{th} worker to received the i^{th} output and gathers it back to output array. This aggregated result is then sent to python testcase for validation.
3. **Input Distribution**: The framework allows for distributing different inputs to various child processes, enabling performance checks for different inputs across multiple nodes.

Scope and Timeline

Final Submission

- **Objective**: Implement robust synchronization and communication mechanisms for handling large-scale distributed execution and result aggregation.
- **Future Plan**:
  - We have completed 17 test cases for the forward differentiation module. Our next steps include finalizing the remaining test cases and extending the solution to support reverse differentiation for which we need to incorporate support for sending the output parameter as a pointer in the input parameter.
  - Enable the execution of programs with large input data concurrently across multiple machines. For instance, running computationally intensive algorithms such as gradient descent in parallel, followed by an all-gather operation to consolidate and aggregate outputs from all nodes, thereby significantly reducing overall execution time.

References

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