Abstract

MPI is the de facto standard for communication among processes that model a parallel program running on a distributed memory system. The communication between different processes is very high, so it is desirable to reduce the communication between process. It is also desirable to parallelize communication with computation to full utilize the computation power of each CPU. In this paper, we present a llvm compiler pass that can change MPI synchronous call to asynchronous call to parallelize communication and computation to improve the performance of MPI programs. We ported different parallel algorithms and use the LLVM pass to transform the algorithm implementation code. Experiments show that such transformation can improve the performance of a program.

1 Introduction

Due to physical limitations, the clock of CPU can not growth exponentially as predicted by Moore’s Law. Light isn’t getting any faster. Today, major processor manufactures such as Intel and AMD has switched to hyperthreading and multicore architectures instead of driving clock speed higher. Most personal computers now have 2 or 4 cores. It is unavoidable to shift the programming paradigm to parallel programming. As Herb Sutter said: “The Free Lunch Is Over”[7]. OpenMP and Open MPI is introduced to simplify the task of developing parallel program. However, when programming using Open MPI, the communication cost is very high. So the algorithm designer and programmer must minimize inter process communication as much as possible. While programming in a parallel way is challenge for programmers, minimize the communication between will burden programmer more. In addition, synchronous MPI calls will block the calling process until the operation complete. It is desirable to parallelize synchronize communication and computation. The programmer could use asynchronous call but it will introduce more complications and make the code difficult. In this paper, we implemented a LLVM pass that can automatically change synchronous MPI call to asynchronous MPI call and move as much instructions as possible between asynchronous calls to full utilize the time between MPI calls.

2 MPI and LLVM Basics

Message Passing Interface (MPI) is a specification for an API that allows many computers to communicate with one another. It is used in computer clusters and supercomputers[1]. MPI support both point-to-point and collective communications. MPI is the de factor standard for communication among processes. A typical implementation of MPI include a set of API that can send/recv messages, broadcast messages and manage the MPI environment. Some of the related concept of MPI that is related to our work is presented in the following paragraph.

Communicator is an object connecting groups of processes in the MPI session. It determines the scope of the communication universe in which point-to-point communication or collective operations is to operate. MPI_COMM_WORLD is a predefined communicator that consists of all the processes in a MPI session.
**Rank** is the integer used to identify a process in a group.

**Message** is an encapsulation of the data that is to be send for sender to receiver. Simply put, message = data + envelop. The envelop include the rank of receiver, sender, tag and communicator.

**MPI_Send** and **MPI_Recv** are two of the most used functions of MPI that is used to send a message from one process to another. They are blocking API calls which means the process will block until it is safe to modify the sending buffer. MPI also support synchronous send by calling MPI_Ssend which will block until the receiver correctly received the message.

**MPI_Isend** and **MPI_Irecv** has the almost the same semantic of MPI_Send and MPI_Recv except that it will return immediately and notify the application when it finish the sending.

In this paper, a LLVM pass is implemented to replace blocking call such as MPI_Send with MPI_Isend and safely move instructions between send and receive to parallelize the program.

**LLVM**, Low-Level Virtual Machine, is a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmer[5]. LLVM defines a common, low-level code representation in Static-Single-Assignment form, with a simple, language-independent type-system that expose the primitives commonly used to implement high-level language features and an instruction for typed address arithmetic and a simple mechanism that can be used to implement the exception handling feature of high level language. LLVM provide a tool chain to compile and analyze the program.

**llvm-gcc** is the LLVM C front end which compiles program to native objects, LLVM bitcode or LLVM assembly language. The **-emit-llvm** and **-c** will generate LLVM bitcode. The **-S** will generate assembly code.

**lli** directly execute programs in LLVM bitcode format using a just-in-time compiler.

**llvm-dis** is the disassembler of LLVM used to to view the LLVM assembly code.

**llc** compiles the LLVM bitcode to native assembly.

In this paper, we implemented a LLVM pass to transform a MPI code to parallelize communication and computation. LLVM provide a pass framework to simplify the development of a pass which perform the transformations and optimizations that make up a compiler. All LLVM pass are subclass of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. There are more specific classes that a pass can inherited from depending on the specific task a pass performs. The most common classes are ModulePass, CallGraphSCCPass, FunctionPass or LoopPass, or BasicBlockPass. In our project, we inherited BasicBlockPass to identify the independent instruction and move them. A basic block pass implement the **RunOnBasicBlock** method with the BasicBlock &BB as it’s arguments.

### 3 N-body Simulation

Particle Simulation is an important area of physical research. It can simulate the celestial mechanics, plasma simulation and molecular dynamics. All such simulations can be abstracted as the N-body simulation problem. The naive N-body simulation take $O(n^2)$ since it calculate the force between each pair of particles. To speed up the simulation, many algorithm is presented. Barnes-Hut (BH) algorithm is a hierarchical solution to the N-body problem[2]. The ideal of BH tree algorithm is that when a set of particles is far away from the particle whose force is calculated, the set of particles can be represented by a single particle whose mass is the sum of all particles in the set and it’s position is the mass-center of the set of particles. The complexity of BH Tree algorithms is $O(n \log n)$. Fast multipole method (FMM) is another mathematical technique that was developed to speed up the calculation of long-ranged forces in N-body system. FMM has a running time $O(n)$ for uniform distributions, but the algorithm is difficult to implement.

Parallelizing N-body simulation is a non-trivial problem due the high volume of communication between each process. It is also difficult to balance the load since the BH Tree is created every time. Maintaining a tree across distributed memory and minimizing communication is of challenge. Many research is conducted to construct high performance N-body simulation such as [6, 4, 3].
As explanatory example for this project, we ported a open source naive N-body simulation\[1\] which will distributed the particles according to their positions. To update the force of a particle, the force between particles inside a single process is first calculated, then the particle is send to each process to update its force by adding the force between all particles in that process. This method is simple and avoid the complexity of sending a BH Tree over MPI. The design of this implementation is presented in figure1. Each process is written using OpenMP which contains 2 thread. One thread will calculate the force of local particles. When it need the information of other process, it will send the information of particle to other processes to finish the calculation. The other thread will receive calculation request from other process and add the force of local particles to the requesting particle and send the particle information back to the requesting process.

Figure 1: Parallel Naive N-body simulation

4 LLVM Transform

4.1 The transformation

We intended to convert the blocking synchronous MPI calls into non-blocking asynchronous calls. The rationale is that if we created an equivalent non-blocking call, the processor might use the time (spent waiting for the call to get back) doing useful computation and thereby improve program performance. So the blocking call:

\[
\text{MPI} \text{Send}(&\text{outmsg}, 1, \text{MPI} \text{CHAR}, \text{dest}, \text{tag}, \text{MPI} \text{COMM} \text{WORLD});
\]

will be transformed into the call

\[
\text{MPI} \text{Isend}(&\text{outmsg}, 1, \text{MPI} \text{CHAR}, \text{dest}, \text{tag}, \text{MPI} \text{COMM} \text{WORLD}, &\text{req}[0]);
\]

The 2 calls are almost identical textually except for the last argument of MPI_Isend which is an OUT argument. This request number is later used in the wait call to check for the completion of the asynchronous call. The asynchronous call returns immediately so we can move the call to the last instruction where one of the operands in the call was defined. Finally we will have a wait instruction before the first use of outmsg (the communication buffer).

First we did the above transformation for basic blocks. We can do some lazy code motion to get the asynchronous call as far back as possible. We can compute the AVAIL sets for basic blocks that will tell us how far backward the asynchronous call can move and how far forward the wait call can move. The AVAIL set calculation can be formalized as the following data flow problem:

\[
\text{AvailIn}(n) = \cap \text{AvailOut}(m), n \neq n_0
\]

\[
\text{AvailOut}(m) = \text{DEExpr}(m) \cup (\text{AvailIn}(m) \cap \text{ExprKill}(m))
\]

4.2 LLVM Implementation

LLVM provides you with an IR that is common to all languages. It also gives you a framework to analyze/modify this IR. There is a basic block pass that iterates over all basic blocks in the program. A basic block is
represented as an instance of a class called BasicBlock in the framework. An Instruction is contained inside a BasicBlock. A BasicBlock is contained in a Function object which in turn is contained in a Module object. The basic block object is passed to the method runOnBasicBlock where you can run over the instructions contained in that object and modify them. Here is the pseudo-code for the analysis pass:

```cpp
virtual bool runOnBasicBlock(BasicBlock &blk) {
    // blk is a pointer to a BasicBlock instance
    BasicBlock::iterator i = blk.end();
    Module* mdl = (Module *) blk.getParent()—>getParent();
    while (i != blk.begin()) {
        --i;
        if (CallInst* callInst = dyn_cast<CallInst>(&*i)) {
            // The instruction is a call instruction
            if (callInst is a call to MPI_Send) {
                // create the equivalent asynchronous call
                // iterate over all the operands of the call
                // and determine the last def site for the operands
                // insert the non-blocking call at this point
                // delete the blocking call
            }
        }
    }
    // insert the wait call here
    return true;
}
```

LLVM has class hierarchies that you must obey to create new function calls. First you have to create a FunctionType associated with the function. Every FunctionType is registered in the module. So for creating a new call instruction you have to create the corresponding function if it does not already exist. For creating a FunctionType, you need to know the return type and the types of the formal parameters. LLVM has its own way of representing the variable types, e.g. The MPI defined types are represented as opaque types. After you create the function type, you have to create the call instruction with all the actual parameters with their types and values. Finally you can use use-def chains to find the last definition for the operand. The IR being in SSA form, there is only one definition. The problem here is the definition might be outside the basic block and its difficult to modify another basic block inside the one we are currently iterating. So we just inserted the call at the beginning of the block if we did not find any definition inside the current basic block.

I searched for ways to automatically solve dataflow equations using LLVM. We did not find anything in the documentation. Then we looked at the code for some of the already built-in passes e.g. ConstantFolding, MemoryDependenceAnalysis. They basically iterate forward/backwards over the basic blocks of the program. BasicBlock has methods that give you access to the previous/next basic block in the program. And you can iterate over the instructions of a basic block as shown in the pseudo-code above.

### 4.3 Difficulties Encountered and Lessons Learned

The main difficulty we faced was finding out the structure of the LLVM IR and how things are represented inside it. Also building/modifying the IR requires considerable playing around with LLVM and understanding its class structures. The way the types are represented makes it difficult to do above transformations. The MPI types are represented as opaque types (a kind of LLVM types). The IR actually creates structs that represent the MPI types and then bit casts them to the opaque types while calling the function. So its a little difficult to create function call instructions out of these types because you have to worry about the opaque types and the bitcasts. The transformation is basically a source to source transformation and it should be done at the source level. The
choice of doing the transformation at the IR level was influenced by the expressiveness and the generality of the IR. In theory the transformation can be done with the IR but it requires a lot of effort and playing around with LLVM. Its easier to do this transformation at source level. One has to find frameworks that can convert different languages to a common IR (maybe AST?) that is closer to the source language.

5 Future Work

For MPI calls that has large amount of data to transfer, it is desirable to split them into small pieces and parallelize it’s computation. In the N-body simulation, one improvement would be send the all particles in one process to another process in a batch and calculate the force. Since calculation requires only one particle, we can split the MPI recv so we can do the calculation each time we receive a particle when we are receiving more particles.

6 Conclusion

In this report, we presented a automatic transformer that can automatically transform synchronous MPI calls to asynchronous calls to parallelize communication and computation. The transformer will move the independent instruction between MPI calls to do computation while the program is waiting for communication. We evaluated the transformer with a naive transformer. Although we haven’t yet finish all the transformation, we already have a partial transformer. Given more time, we will be able to do experiments on the N-Body simulation code.

References


