Automatic High-Level Profile Directed Code Optimization for Parallel Architectures

Mirza Mohd Aileeya Qasim
Research Scholar, Department of Computer Science and Engineering, Dr. A. P. J. Abdul Kalam Technical University, Lucknow, Uttar Pradesh, INDIA

ABSTRACT
Code optimization techniques improve the performance of code at the IR (intermediate representation) level of the compiler. Parallel code optimization and execution has always been better than sequential because parallel applications can fully utilize the power of modern multicore processor architectures. A compilation framework based on LLVM featuring parallel task compilation and automatic parallel code generation is developed in this research. Good features of polyhedral model are used for optimization along with profile directed feedback technique to provide good performance. The impact of this framework on programmer’s productivity is also discussed. The results reveal that the new technique outperforms current compilers for many applications taken from Polybench/C-4.1 benchmark suite.

Keywords— Automatic Parallelization, Compiler Optimization, Polyhedral Model, Profile Directed Feedback.

I. INTRODUCTION
Modern compilers are highly modular and are nowadays better known as optimizing compilers [5] because they have a well-defined optimizer module which is responsible for improving the performance of the generated code. Optimizations can either be machine dependent or independent. Recent advances in optimization technology resulted in profile directed feedback [8] and polyhedral optimization techniques [2] which are able to push forward the state of the art in code optimization.

In polyhedral optimization, the problem of code optimization is viewed in terms of integer polyhedra. The problem of code optimization is first converted to an integer linear programming (ILP) problem, which is then solved by a linear programming solver such as Integer Set Library (ISL) [9]. The ISL is a combinatorial optimization problem solver which mathematically optimizes the solution. The solution is converted back to the original code optimization solution, which is in turn applied to the process of code optimization.

In case of profile based optimization, the program is compiled and profiled or simultaneously compiled and profiled and the performance is evaluated by running portions of code on the hardware and getting useful information which is otherwise not available at compile time. This information is then feedback to the optimizer which performs optimizations based on the profile information specific to the hardware.

Both polyhedral optimization and profile directed feedback are popular and successful in their own right, but still opportunities for improvements are available. In this research, we combine the good features of both polyhedral and profile directed optimization to increase performance. Instead of using traditional techniques in the profile directed feedback optimization, we use polyhedral optimization.

II. PROBLEM IDENTIFICATION
Various optimization techniques were analyzed on the LLVM compiler infrastructure [4] and a number of optimization techniques were applied on various programs. These techniques which are already present in the optimizer module of the compiler were able to improve the quality of sequential programs to a very large extent and optimal code was generated in many cases. This statement is only true for small sequential programs. However, when it comes to large programs, i.e. very high performance scientific computations and numerical simulations, the time taken for optimization is quite large.
The major problem found in these techniques is that for very big programs, the compilation and optimization techniques are taking much time which are not usually bearable. The performance of the object code is also not up to the mark.

In investigating the optimization techniques on a Core i3 processor, 198 MB C++ source code was compiled with all the optimizations turned on using GCC 5.3.0. The following issues were detected:

- The program took around 4.5 hours to compile. This is because the compiler is applying all the optimization techniques on the source code one after the other.
- It is also because the compiler is not using multiple cores while compiling or optimizing.
- The generated object code does not even utilize the power of modern multicore processor architecture unless and until it is explicitly made parallel by annotating the source code with OpenMP pragmas.
- Many optimization techniques were applied which could yield no performance gain and hence, unnecessarily increasing the compilation and optimization time.
- Some optimization techniques when applied to a particular program even decreased the performance of the generated object code.

### III. PRIOR APPROACH

The technique of online adaptive optimization was used in [1]. In this technique, the authors first compile the program normally and at the time of execution, they generate a profile of the program based on the execution time. Some specific optimizations are applied to portions of the program and again the program is compiled using a just in time compiler while the program is running. The program is profiled again and if the performance is better than the previous one, the previous result is discarded. The process continues on different tasks until no more optimization technique produces a better result on performance. At this stage, the program becomes stable and continues to provide high performance.

![Figure 1: Prior approach to adaptive optimization due to [1]](image)

In this technique, when the program is executed, initially it slows down but after sometime it adapts to the hardware, it starts giving good performance. This is because, in the initial phase, the just in time compilation, profiling, optimization, comparison and execution are all happening at the same time. But after sometime, it automatically figures out how to do it in the best possible manner.

This technique might seem quite promising but it has several disadvantages and issue. These are listed as follows:

- What if the adaptive optimization is not successful as in case of some programs, then the computer will be forced to execute an un-optimized version of the program and hence, huge amount of performance losses.
- The optimizer is trying to optimize a program that cannot be optimized and hence increasing the execution time of the program.
- It is also possible that optimizer enter into a race condition in which it is never able to settle on a stable condition after repeatedly applying various optimization techniques.
- If the program is executed on an embedded system with power constraints, the framework will use up major fraction of the battery in just trying to arrive at an optimal conclusion rather that executing the program efficiently.
- If the program is executed in a real time environment, in which deadlines need to be met and are fixed, then in that case, the approach miserably fails because the exact execution time is not known before the execution of the program.

In view of the above issues, it is required that a new optimization technique should be developed, by which the above problems can be addressed.

### IV. SOLUTION APPROACH

In order to perform parallel code optimization, first of all the sequential application should be parallelized suitably. For this, the LLVM Polly library [2] is used to automatically parallelize [7] the application if it contains some degree of parallelism. It is not mandatory that the library is able to parallelize the code or not, as it depends upon the program itself. The intermediate code is generated from the above step in LLVM format [6]. The code contains compiler hints or annotations and references to the OpenMP runtime library. From these OpenMP library references, the parallel regions of the code are obtained. The code is bifurcated into multiple parts and can also be stored in multiple files based on these parallel regions and each region can be optimized in parallel because of the known fact that threads executing in parallel do not contain any dependencies.
There are some programs which if parallelized can even lead to performance degradation. This is because the LLVM Polly parallelizer tries to insert OpenMP pragmas to an otherwise inherently highly sequential program which in turn increases the execution time. The decrease in performance is because of the induced overhead. Thus, it is nearly impossible to parallelize these programs.

To decrease optimization time further, specific optimization techniques are applied to specific regions of code. For example, if a region does not contain any loop, then it is of no use applying 15-20 loop optimization techniques one by one and wasting time. Similarly, if the program doesn’t contain any function calls, then it is of no use applying inlining or interprocedural optimization.

It is also possible that these programs are more amenable to vectorization rather than parallelization. Or these programs are good at exploiting instruction level parallelism. Whatever the case may be, this problem is circumvented by checking the performance of the program by applying different optimization patterns to the source code and finding out the best among them all. This is because a digital image processing application is more amenable to data level parallelism rather than a Floyd-Warshall’s algorithm which otherwise is highly sequential and somewhat amenable to instruction level parallelism.

The technique can be summarized in the form of an algorithm which is as shown as follows:

**Shared Variables**
- \( n \) Number of threads
- \( i \) Thread index
- \( k \) Array index
- \( a \) An array of pointers of size 50
- \( b \) An array of pointers of size 50
- \( \text{temp} \) Temporary variable

**Private Variables**
- \( j \) Index of thread
- \( c \) Counter

**Define TASKS 100**
**Input**
- file1 Un-optimized code in LLVM IR form
**Output**
- file2 The optimized code in LLVM IR form

begin
  for \( k = 0 \) to \( n - 1 \)
    \( b[k] = \text{null} \)
  endwhile
  while(not EOF)
    “Scan the IR for the “@GOMP_parallel_start” and “@GOMP_parallel_end” regions. The code between the two contains the parallel region.”
    add the pointer to the location in the array \( a \)
  endwhile
  while (the current portion is data parallel or serial)
    add the current task to array \( b \)
    Optimize using polyhedral optimization techniques
  endwhile
  call algorithm 2 on (array \( b \))
  append the code pointed by \( b[j] \) to file2 in order
endwhile

**ALGORITHM 2: For task-parallel programs**

begin
  for \( k = 0 \) to \( \text{TASKS} - 1 \)
    if \( b[k] \neq \text{null} \)
      \( \text{size} = \text{size} + 1 \)
    endif
  endfor
  \( \text{temp} = \text{size} \)
  \( \text{size} = \text{size} / n \)
  if \( (\text{temp} \% n != 0) \)
    \( \text{size} = \text{size} + 1 \)
  endif
  for each thread \( t_i \) where \( t_0 \leq t_i < t_n \) do
    \( j = i \)
    \( c = 1 \)
    while (\( c \leq \text{size} \))
      if (\( b[j] \neq \text{null} \) && \( i == n \% j \))
        system (\( \text{opt} \ b[j] \))
      endif
      update the new pointers
    \( b[j] = \text{null} \)
    \( j = j + n \)
    \( c = c + 1 \)
  endwhile
endfor
end
Our technique is a variation of online adaptive optimization. The major difference is that the optimizations which were taking place at runtime are brought back to compile time profiling and polyhedral optimizations are used instead of traditional techniques. In the new technique, various optimization patterns if applied to the source code and then profiled and checked can lead to huge performance improvement.

- Optimize for instruction level parallelism
- Optimize for data, thread and task level parallelism
- Optimize for data, thread and task level parallelism + vectorization
- Apply all optimizations

Compare performance in all cases by executing the code and recording the execution time. The code is generated for the technique with the best execution time.

V. FINDINGS

In case of parallel code optimization, it was found that traditional compilers were not utilizing the power of the modern multicore processor. They were only confined to a single processor and as a result, the optimizer module of the compiler was applying multiple optimization techniques on the same piece of code in a sequential manner. This was the case even in programs where there was no dependency whatsoever between the basic blocks.

In case of profile based optimization it was found that different programs contain different properties. Some programs can easily be parallelized, while others cannot. Some programs can be vectorized while others cannot. Some programs are amenable to polyhedral optimization while others just cannot be parallelized. If these programs are to be parallelized, they may even slow down the performance of the program. The slowdown is due to the overhead which is introduced in the applications. The overhead may be due to the use of pragmas and annotations and other compiler hints which when processed even decrease the speed of the execution of code. So, in view of the above problems, the current research focuses on the development of an optimizer which automatically tells as to what optimization pattern should be applied to the currently executing code.

Figure 3 shows the relative performance of all three optimization techniques taken into consideration. Manual parallelization clearly the winner because it is taking much less time. Then comes the performance of the proposed technique and finally the technique due to [1].

VI. RESULT

The technique of parallel code optimization was evaluated on various benchmark programs. These programs were mainly taken from Polybench/C benchmark suite [3]. In many cases, it resulted in a better or equivalent performance gains.

The performance levels of various programs were evaluated. These programs were taken from Polybench/C-4.1 benchmark suite which contains 30 programs taken from diverse domains such as data mining, linear algebra, medley and stencil computations as shown in figure 5. All of these programs were analyzed and studied. The programs in the Polybench/C-4.1 benchmark suite are not only amenable to traditional optimizations but also amenable to modern polyhedral, high level loop and data locality optimizations. Because programs are also being profiled with polyhedral optimization in consideration, therefore, Polybench/C-4.1 is the best option.

The results generated were mapped to the vertical axis of the graph with time as a parameter.
Benchmarking was done by using different types of optimizations pattern such as optimizing for instruction level parallelism, thread level parallelism, task level parallelism, vectorization, polyhedral optimization and a combination of some approaches using a specially developed tool shown in figure 4.

VII. DISCUSSION

The GCC compiler also allows parallel program compilation and optimization. But their technique is drastically different from this one. In GCC, parallel code optimization only takes place when the program is distributed among multiple files and all these files are needed for final linking. The GCC compiler creates multiple instances of the same compiler, the instances being the number of threads on the processor and then, each instance of the compiler compiles and optimizes different source files in parallel. This seems a naïve approach and cannot optimize parallel programs in parallel, and hence wasting time. This research only addresses the optimizer, not a linker, not even a compiler.

A program development cycle consists of many phases. The time required in each of the phases is listed below:

- **Time taken in sequential program development:** In this phase, the programmer writes a program for a single processor.
- **Time taken in program parallelization:** In this phase, the programmer analyzes the program for parallel regions. Make those regions parallel by making use of a shared or distributed memory parallel programming library.
- **Time taken in debugging:** In this phase, the program is debugged and tested on various inputs.
- **Time taken in compilation:** In this phase, the programmer compiles the program by using a compiler. This step takes the high level source code as input and generates an intermediate representation for further optimization. It may also directly generate the code for the target architecture.
- **Time taken in optimization:** In this phase, the program is then optimized by applying various optimization techniques to the intermediate code in either machine dependent or independent manner.
- **Time taken in execution:** In this phase, the program runs on the hardware.

The programmer uses 70% of the time in writing the sequential program, then 20% of the time in parallelizing it. So, manually parallelized OpenMP program requires 90% of the programmer’s time. This is huge time, but once the program is developed, it gives a very good speedup on execution. The speedup because the programmer has manually parallelized the program and has taken care of all aspects of the program.

This hand crafted program gives a very good performance, but it requires a lot of expertise. As it is well known that parallel programming is a challenging task and very few people are expert in this art. So, in order to get a good performance, a person also need to be a good parallel programmer.

As discussed in [1], authors try to reduce the burden of the programmers by automatically parallelizing the application. This is done by introducing polyhedral model of integer linear programming to code optimization. The polyhedral model is a mathematical optimization tool which finds out the most optimal solution to any given problem.

The authors first convert the problem of code optimization to an integer linear programming problem. Then, they solve the problem using methods of combinatorial optimization, and once solved, they convert it back into the solution of the code optimization problem. In this way, a truly optimal code is generated. An external linear programming library is used to solve the linear programming problems and also optimize high level loops and data locality which is again a big plus.

The lifecycle of a program which uses adaptive optimization, it is seen that 80% of the time is spent in sequential program development and manual parallelization of program is completely ruled out. This is because automatic parallelization comes into picture and the programmer need not to parallelize the application all by himself. Small fraction of time is consumed in compiling and applying for traditional optimization techniques. The execution time drastically slows down because of adaptive optimization to 15%.

The major problem in their approach is that the execution, optimization and profiling of the program is taking place at the same time, i.e. runtime. In this approach, initially the program slows down because it
doesn’t know what optimizations should be applied, but slowly the speed of execution increases because it figures out how to do it in the best possible manner.

This initial slowdown is a major drawback of the previous approach. The time which was wasted in online adaptive optimization is removed from here. This has been achieved by profiling the program before its execution.

Figure 6 shows the proposed approach. In this figure, although the development time and compilation times are similar to the previous approach, however the optimization time and the execution time have changed considerably. This is because, in the proposed approach, profiling and optimizations are done before the execution of the program. This gives us a good performance.

One thing should be noted here is that the performance in this case is not better than the previous OpenMP version of the program. The reason being that the OpenMP program is manually parallelized whereas the current optimized program is automatically parallelized.

It is widely accepted that programmer’s comfort sacrifices performance. An assembly language program gives a very good performance but programming in assembly language is a quite challenging task. To give comfort to the programmers, programming languages and compilers were developed which sacrificed some aspects of performance but were very much programmer friendly. Then came the era of parallel computing in which programming in pthreads used to give a good performance but still it was quite challenging. A pthread wrapper named as OpenMP was created which again sacrificed the performance to a certain extent, and still quite programmer friendly. The disadvantage of OpenMP is that the programmer is required to understand the underlying architecture of the hardware which is otherwise unexpected from an application developer or a web developer.

To increase the programmer friendliness further, automatic parallelization techniques are developed, which again sacrifice performance to a certain extent. But things are improving and in probably some years from now, good automatic parallelizing compilers will be seen.

In order to check the performance of their technique, the authors in [1] have benchmarked the process on Polybench/C-3.2 benchmark suite which is quite old and was released in 2012. As of 2015, a new version of the benchmark is released as Polybench/C-4.1. The developers of Polybench/C have clearly indicated that one should not use Polybench/C-3.2 because it contains a lot of anomalies. So, this means that the result of [1] is not perfect. The reason being that the program which is being profiled does not utilize all the cores present in the system because some of the cores are being utilized by the profile, just in time compiler, a scheduler and a comparator. This again results in inconsistent performance. Those anomalies were corrected in Polybench/C-4.1 version which has been used in the current proposed work.

**VIII. FUTURE RESEARCH**

The technique presented in this research optimizes the programs in parallel at a higher level of abstraction which introduces huge amount of overhead. It is simply the interconnection of various modules into a unified framework. If this technique be implemented in the core of the infrastructure, then huge performance gains can be achieved. At the lower level of abstraction, there is very little overhead and the performance achieved can be better than the current one. The current version 3.7.0 of LLVM compiler infrastructure is still a sequential compiler.

Once this technique is implemented inside the compiler, complex dependency analyses can be figured out and a more optimal compiler can be developed. These core optimizations will rule out overhead and as a result a high performance compiler can be developed.

In this research, different optimization techniques were brought together and scheduled in a specific way. This technique can also be generalized in to a very big framework if the nature of the application is not known in advance.

For example, there are many different compilers in diverse domains. These domains include digital image processing, data mining, linear algebra, etc. All these domains have their own specific compilers which are specific to their own particular area. But what if the nature of the application is not known in advanced, then a large number of compilers can be combined into a single unified framework and based on the optimization pattern specific to each application domain, the application can be optimized suitably.

Meaning to say that, if a digital image processing application is first optimized for instruction level parallelism and then parallelization. Clearly, parallelization will improve the performance because digital image processing applications are more amenable to
parallelization. It is also possible that a combination of more than one technique would serve the purpose.

This process can also be applied to legacy code in which the exact pattern of optimization is not known until execution. This is because some details can only be revealed at the time of the execution of the program.

IX. CONCLUSION

In this research, a parallel code optimization technique was described and developed. It was discussed that by optimizing independent regions of code in parallel, a good speed can be achieved. The dependency between fragments of code was resolved by LLVM Polly library. Because of the independent regions of code can be optimized in parallel, different code fragments were distributed along different threads, which optimize the program in parallel. It was seen that because the optimization of code in parallel is a very short duration, not much speedup was obtained. This is because of less execution time and high overhead.

Along with this, a profile based optimization technique was discussed in which it was demonstrated that different programs contain different types of parallelisms. It was shown that it is nearly impossible to parallelize some types of applications. The performance of some types of applications was affected because of the overhead introduced. These applications were otherwise highly sequential and cannot be parallelized or vectorized.

The technique of profile driven optimization was also implemented in C on a Linux system and it was found that it gave correct results on all of the sample benchmark programs taken. Because time is a measure of performance, time was measured after applying different optimization patterns and the best outcome among them all was selected and the code was generated.

Also, the technique in this research was proved to be better than the technique due to [1]. In their technique, the code was simultaneously profiled and optimized while the program was executing. This technique of profiling and optimizing was shifted and allowed to be performed before the execution of the program which in itself is a big plus.

Programmer’s productivity was also described in which it was discussed that how using the tool developed in this research can reduce the burden of the programmers by sacrificing little performance. Now, the programmer is relieved of the burden of manually parallelizing his program because automatic parallelization is being done by the compiler itself. It was also discussed that for a high level programmer like a web developer or an application engineer does not need the knowledge of multicore architecture, but still he wants performance in his application. The technique discussed here clearly addresses this issue.

REFERENCES