ISSN 2047-3338

Automatic Parallelization Model for Processors

Asif Ali[1] and Zahid Anwar[2]

[1,2]Department of Computer Science, COMSATS Institute of Information Technology, Vehari, Pakistan

[1]asifali@ciitvehari.edu.pk, [2]zahidanwar@ciitvehari.edu.pk

***Abstract*— Performance Improvements and decreasing execution time had been started half a century ago, along with the development of new chipsets and microprocessors with increased clock speeds, the software engineers have also developed ways to increase the performance of their developed systems by introducing new language constructs and other performance improvements. As we know there are many software modules that are complex like airline monitoring systems, multi-variable differential equations, AutoCAD 3D drawing, and HD graphics video games that require immense computations that a single processor could not perform. The solution to this problem is to utilize the ubiquitous commodity of modern world – The Multi-Core Processors. A major hurdle in utilizing this commodity is the overhead needed at the developer end to convert a single threaded application into a multi-threaded application. This paper intends to find a comprehensive model that could be used to overcome this hurdle by introducing new pragmas in existing code, or tweaking in the compiler so that automatic parallelization is introduced at the compiler level.**

***Index Terms*— Automatic Parallelization, Multi-Threaded Execution and Performance Improvements**

I. INTRODUCTION

IN today's modern world speed is the foremost priority in every sphere of our life, be it travelling, delivering products, shipments, sending of data, or its processing. With the world heading towards a global community merged together by technology in each aspect of its inhabitants' lives, speed of delivering meaningful information has become compulsory. The most important part of delivering data is the ability of the sender to firstly encode the information it needs to send using software made for the encoding process. The software that encodes the data is executed by a computer that has a processor installed. The processor executes the software and encodes the data. It has become mandatory to increase the processing speed of every software if we want to move fast in the modern world. The main purpose of this research study is to increase the speed of execution of the programs. We can achieve this by either increasing the clock frequency of the processors that we use in our computers or introduce more processors or cores on a single chipset. The chipsets now available have maximum number of cores deployed on one or more processors. To exploit the number of cores available to the software is a major problem to the users and developers alike. Previously developed programs were made keeping in the mind the sequential nature of the execution of every processor available then. With the emergence of multi-core processors we need to exploit the extra cores now included in the chipset. Unfortunately the cores available are usually unemployed or underemployed by the variety of programs. For utilizing the complete prowess of these powerful chipset we need to parallelize our execution flow so that each processor and each of its cores are used during the flow of execution of our program.

There are libraries available in different languages that enable us to utilize each processor but using them while developing is an overhead at the developer's end. We try exploiting parallelism in software but it requires different design decisions, significant programmer effort and other difficulties should be overcome before exploitation. The solution to this problem is Automatic parallelization. It is a promising approach that if used effectively brings drastic performance improvements in the world of computing just by using all the cores already available. But as all great things come with a price, it is a really challenging approach. Migrating single threaded applications to multicore platform is a difficult task and automating their executions in a parallel environment is much more challenging.

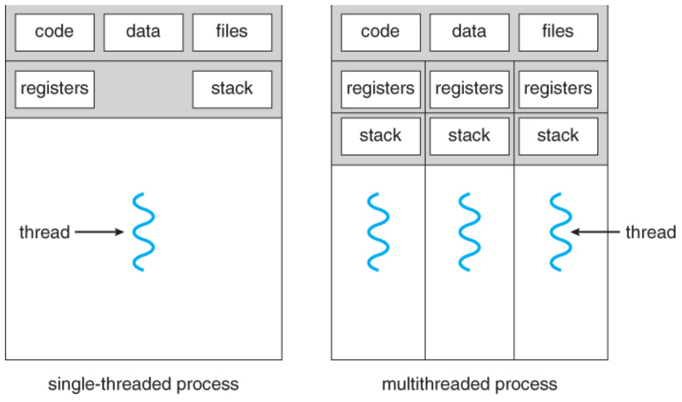

Fig. 1: Single Thread vs. Multi Thread

Fig. 1 shows us what we can achieve using the Multi-Threaded environment of the multi-core processors made now a days. It clearly depicts that if all the cores are utilized than the performance gain be multiple that is directly proportional to the number of cores inside the processor or chipset. Each processor or its core has its own cache, a set of registers used while executions as well as its own stack and data given to it. To attain this individuality of code, data, and the processing result is the main challenge encounter by the modern world. We intend to find a complete model that firstly solves the problem of parallelization and secondly uses an automatic technique for parallelizing the flow of execution by assigning each task to a core or processer so that execution is done in a multi-threaded manner. Our ultimate purpose of this research is to find a model that automatically collects program information without requiring any modification in the program design or developer involvement and automates the processing of the program. There are many ways to get a multi-threaded environment but converting existing environments and code is definitely a challenge. We want to achieve parallelization by introducing necessary tweaks in either the kernel or compiler so that the parallelization is automatic. There is less developer overhead and more performance improvement. We can even insert new code in appropriate places in the application using automatic parallelization techniques so we overcome the challenges that have a detrimental impact on compile-time analysis required for automatic parallelization.

This paper will have different sections, Section II will discuss the concept of parallelization in detail, Section III deals with the parallelization of different statements, conditional and repetition constructs present in the different programming languages. Section IV discusses the automatic parallelization study and research done in different programming languages. It explains the automatic parallelization in popular languages like C, JAVA, and other languages. Section V will discuss the comprehensive model that could be used for automatic parallelization. Section VI concludes the findings of the research and discusses the impact of automatic parallelization of the software products if the proposed comprehensive model is implemented and be followed.

II. CONCEPT OF PARALLELIZATION

The concept of Parallelization involves two major phases. The first phase is the actual process of designing and writing a computer program or software that has the ability to process the given data in parallel. Parallel processing means that the code written by the programmer is actually self-capable of using multiple cores to perform its execution. For most of our lives we have seen that the computer programs perform computation serially. Serially means they perform one computation first and then move towards performing another computation. Similarly this process of executing or performing a computation one after another is followed until the program execution is not complete. If a code written by the developer or the complete software is parallelized, then the code itself divides the process, the complete process to be precise in sub-processes. These processes can then be executed independently and the computations are performed in a parallel fashion by different microprocessors. The main idea behind this concept is to convert serial or sequential processing into parallel processing. If a programmer introduces the parallelization in its code and then optimize the code specifically for parallel computation then the software can perform all the computations in a much faster way as compared to the simple serial flow of computation used now a day.

The concept of Parallelization is incorporated in the computing realm for many years. This concept and its use were usually limited to their use in the field of supercomputing. In the last ten years, the micro-processors have reached their physical capacity. This capacity is the clock frequency, or in simple words the execution of number if instructions in one clock cycle. Sure the clock frequency can still be increased but it is not feasible for personal computers. The reason for it being the power consumption and heat generation due to a more powerful micro-processor comes with these drawbacks. The technology now has to mature in other parts of computation rather than just the clock cycle or speed of the processor. This is a major design decision in handhelds, PCs, and even mobiles. In today's world most of the smartphones, personal computers/laptops and modern desktop computers have multiple cores and even processors on their CPU that enable the parallel processing within the operating system using either software additions or some software tweaks.

The concept of parallelization should be applied, but sometimes the total time taken in the execution of a program with parallelization embedded exceeds the one with no parallelization. This is due to the fact that parallelization comes with a price. It has its own overhead of managing the data, memory, and cache across the different cores and processors. The overhead is in fact directly proportional to the number of cores and threads in a system. If we have a Dual Core CPU then we have to manage two cores along with their cache and the data among them. Similarly if the cores are increased to 8, then the management overhead also increases with the number of cores. Now the data between the 8 cores as well as the 8 cores themselves have to be managed. This decreases the efficiency of the computer and the program takes much more time during its execution compared to a simple processor with only one CPU. The most time consuming activity in parallelization is the synchronization of data during the whole execution. The data has to be synchronized completely after the execution of each statement by each of the processor to ensure that the data is consistent.

Before the advent of multi-core processors, when a large amount of computations were needed to solve a complex problem, the scientists or the users usually waited for the arrival of a new micro-processor that has a higher clock cycle and would be much more efficient so that there computation would be performed. The people waited owing to the understanding of *Moore's Technological Advancement Law,* which they interpreted to mean that the speed of computers would approximately be doubled about after every two years. But analyzing the recent advancements in the technological sphere of science, Moore's Law does not hold any longer. For instance, if we took a micro-processor installed inside a desktop computer that is two years old and check its speed it usually results to be 2.5 to 3.0 GHz. So according to the Moore's law this speed should have been 5.0 GHz to 6.0 GHz. But this is wrong, the speed has barely increased if it is else we can now find many flagship processors and chipsets with lesser frequency and speed. The main reason behind this was the failure to develop processors with such high speeds with lower power consumption suitable for personal use and to be low cost as well. So the computer manufacturers have doubled even tripled the number of processors or cores on a single chip. Even very cheap smartphones now have OctaCore-8 cores (processors), similarly we can CPU chips with 16 cores, and this number will soon increase. Now a day even graphics processing units (GPUs) are bundled with over 100 highly specialized processors used specifically for Graphics processing. This is another example of Moore's technological law.

He used to say that the number of transistors would keep doubling every two year. This has made the rate of improvement slow down, but there is a significant increase in the number of transistors. This trend should continue for at least ten more years until we advance in some other direction. If there is a small number of computing cores inside the microprocessor of the computer then it is simple to find tasks that can be done in a parallel fashion, such as waiting for keystrokes and running a browser. But as we increase the number of processors or cores, the parallelization problems such as synchronization of data between the processors becomes a very large overhead.

III. AUTOMATIC PARALLELIZATION OF DIFFERENT STATEMENTS

Automatic Parallelization is difficult to achieve. Similarly parallelizing each and every module of our computer program is more problematic and increases the overhead so much, that there is no performance improvement. Sometimes the performance is reduced and the program takes more time in execution when automatic parallelization is employed. This section discusses the reasons, techniques, and the result of automatic parallelization for different types of statements and functions in popular programming languages such as JAVA and C.

A) Multi-Threaded Execution of For Loops

The most important statement usually during the computation is a loop. Loops are mostly the most time consuming activity during the course of execution. The time-complexity of any algorithm depends upon the number of loops and nested loops employed in its implementation by the developer. Monitoring the "for" loop closely, gives us enough evidence that our loop structure is very slow and we need to add parallelization to speed up our loop by using multi threads or cores.

The concept of parallelization can be added after we analyze "for loops" in C code using Open MP API. To add the parallelization automatically we insert and use two of the following data structures:

- Variable Table
- Loop Table

According to the research" Towards Automatic Parallelization of "for" Loops", Automatic parallelization is achieved by a simple algorithm designed in the research itself. The algorithm takes as input a sequential C code, performs some computations, inserts some lines of code that becomes an overhead in real-time processing and then outputs a C Code which has parallelization incorporated in it and is ready for parallel execution on a multi-core processor machine or multi processors. The algorithm devised in the aforementioned research paper converts a sequential C code into a parallel executable program using the following steps, these steps ensure pure parallelization is achieved. The steps followed by the program are discussed next.

The parallel executable program is created using the following steps:

- In the first step the header of the C code – i.e. the first statement used in the "for" loop is analyzed by the algorithm. The header must have a signed integer, comparators – for the condition validation and a variable that is compared should either be incremented or decremented.
- In the second step the data that is defined, altered and used in the loop's Scope is analyzed. The variables are checked and their entries are made in the Variable Table, so that they can be updated during the parallel execution.
- The third step is a basically a check. A check that is used to in determining whether the loop given to the algorithm to be parallelized, is parallelizable or not by the algorithm. This check is formally a dependency check.
- The fourth step is the determination of the efficiency gain achieved by the Loop Parallelization process. This gain can be decrease in total Time Taken by the loop in its execution, the number of context switching occurred in Threads, Synchronization needed to be performed between the multiple threads, or the time taken in the initialization of variables in each of the thread.
- The last step in the process of parallelization of "for" loop is the generation of OpenMP Clause that is to be inserted inside the code of the "for" loop. These clauses are inserted in appropriate places inside the sequential code which in turn enable the parallelism in it.

After the code given to the algorithm in the research, new code is generated which is parallel executable. This code could be executed on any multi-core processor. The number of cores or threads does not necessarily increase the performance of the execution as there as an overhead in executing a serial program in a parallel fashion. The synchronization time is increased directly with the increase in the number of threads and cores. For the evaluation of the algorithm and analysis of automatic parallelization we used the algorithm and performed tests on a piece of code written in C language. The code took the input either by a file or by on screen input. The input includes 8 arrays and the function needed to be performed on these arrays. The function can either be the reversal of an array or sorting of an array. The sorting function employs two loops in which one is nested inside the first loop, whereas reversal employs only one loop that reverses the inputted array. The code file is written in C language and employs a sequential execution technique. The code written is given to the algorithm devised in the research; the resultant code is a new program that should be run in a parallel fashion using a multi-core processor or a micro-processor that has more than one thread installed on its chip. For the purpose of finding out the results of the effectiveness of the algorithm we executed the program on three different computers. The results are explained below:

Results: The program was first executed on an Intel i7-2670QM CPU, which had 8 threads each clocked @ 2.20GHz and with a Random Access Memory of 8GB. The L1 cache was 256KB, L2 cache was 1MB and L3 cache was 3.0MB.

Execution Time in serial program = 11.758s
Execution Time in parallel execution = 2.877s

This clearly shows that the performance of the program developed for this research has increased if it is executed in a multi-threaded environment.

The program was then executed on an Intel i5-2430M CPU, which had 4 threads each clocked @ 2.40GHz and with a Random Access Memory of 6GB. The L1 cache was 128KB, L2 cache was 512KB and L3 cache was 3.0MB.

Execution Time in serial program = 9.85s
Execution Time in parallel execution = 3.56s

This clearly shows that the performance of the program developed for this research has increased if it is executed in a multi-threaded environment. The important thing to note here is that the overhead that occurs does have a significant effect on the execution time. The execution time should have increased two times as the first PC had an 8 thread CPU to perform the execution as compared to this PC with only 4 threads, but the performance gain occurred in first PC was 1.5 instead of 2. This shows that it overhead of parallelization has a significant impact on the PC

The program was at last executed on an Intel Pentium B950 CPU, which had 2 threads each clocked @ 2.50GHz and with a Random Access Memory of 2GB. The L1 cache was 64KB and L2 cache was 256KB. The L3 cache of this computer was only 1MB

Execution Time in serial program = 8.79s
Execution Time in parallel execution =4.57s

This clearly shows that the performance of the program developed for this research has increased if it is executed in a multi-threaded environment. The important thing to note here is that the overhead that occurred is much more significant as compared to the previous computers. The execution time should have increased two times as the second PC had a 4 thread CPU to perform the execution as compared to this PC with only 2 threads, but the performance gain occurred in second PC was 1.25 instead of 2. This shows that it overhead of parallelization has a significant impact on the PC.

B) Multi-Threaded Execution of Recursive Calls

Automatic parallelization of sequential programs has been introduced to provide programmers with the ability to parallelize applications easily. In the paper "An Automatic Parallelization Tool for Recursive Calls" the author have discussed the parallelization for recursive function calls. He has identified and analyzed recursive function calls to get the characteristics of the recursive functions including the number of recursive calls a function issues, its size in terms of statements, memory or space usage, data synchronizations, stack management etc. The previous work done on parallelization is restricted. It does not deal with recursive calls which have data dependency inside the functions and it must have void return type. So, the purpose of the research was to develop an algorithm to deal with this situation i.e. recursive functions parallelization and for better performance of recursive call using parallelization. The author had tested many algorithms, specially the ones that had recursive algorithms and the results are different for different input programs. In some of the cases the Execution time is increased of parallelized program than sequential program and it decreases in some other cases.

For most of the input sizes, parallel implementation takes less time compared to the sequential program. The AUTOPAR algorithm used by the author had the following source code, the functions called did exactly what their name was. The IDENTIFY-FUNCTION-DEFINITIONS took the source as input and returned the total functions definitions that were present in the code. The IDENTIFY-FUNCTION-CALLS took the source as input and returned the total functions calls that were present in the code. The IDENTIFY-RECURSIVE-CALL took the source as input and returned the total recursive calls that were made in the code. The ANALYZE-RECURSIVE-CALLS took the source as input and returned the analysis of all the recursive calls that were made in the code. INTRODUCE-OPENMP was the last function and it introduced OpenMp pragmas in the recursive functions where they were necessary for the parallelization. The function took the calls themselves, and their analysis, along with the code as its input. The output code was the new source. The output code was a transformation of the sequential code to a code that could be executed in a parallel fashion.

The AUTOPAR algorithm structure or the baseline is given in the next paragraph.

```
AUTOPAR(Source)

defs ? IDENTIFY-FUNCTION DEFINITIONS(Source)

calls ? IDENTIFY-FUNCTION-CALLS(Source, defs)

recs ? IDENTIFY-RECURSIVE-CALLS(defs, calls)

anls ? ANALYZE-RECURSIVE-CALLS(recs, calls)
```

```
Source ? INTRODUCE-OPENMP(Source, recs, anls)

return Source
end
```

C) Executing Different Functions Using PAP

PAP known as the Pluto Automatic Parallelizer is a tool that generates parallelized code. This code is automatically parallelized and does not involve any developer effort for the use of parallelization on a multi-processor system. In the paper "Automatic Parallelization Experiments on 16PE" automatic parallelization experiments are performed. The code is generated using the PAP as mentioned above. The computer on which the experiments were performed is a 16PE NOC based MPSOC which was designed and implemented on a single FPGA chip it had an integrated sixteen Micro Blaze based Processing Elements (PE) Tiles System in itself.

To test this multiprocessor system, four major experiments are performed.

*1) Matrix multiplication 128 * 128*

Matrix multiplication is among others the most parallelizable application because of its high data independency. The maximum reduction in time-complexity has resulted in Big-Oh of n raise to the power 2.78, hence it is a problem that needs efficient solutions the most. The parallelized code generated by Pluto is actually a block based matrix multiplication. The resulting matrix is divided into blocks. The results of this experiment showed, on the one hand, the NoC (Network on Chip) is far from saturation as proved by the near perfect scalability and still have space for even heavier traffic loads, and on the other hand, the possibility o f hide the communication latency with calculation is a promising technique for better performance.

*2) Seidel 128 *128*

The seidel problem is also a known problem in computing realm so the author also used a PAP generated parallelized code for experiment. The cycle counts for different processor numbers show a relatively low but still satisfying parallelizability of Seidel Algorithm compared to that of the Matrix Multiplication. The performance scaling keeps track of resource scaling until 8 cores. When passing from 8 cores to 16, we introduce only 27% cycle reduction.

*3) DCT (Blocksize: 4*4)*

In this application, data block size is fixed to 4 *4. This was also observed that beyond 8 nodes, additional processor does not introduce any performance improvement.

4) Jacobi_1d (Vector size: 1000 Iteration: 2)

It is noted that there is a limitation of the PLuTo parallelizer for this particular application and some others (LV decomposition), can enjoy an efficient parallelization only when the Iteration number is great or when the work load is large, respectively. Processors, instead of working in parallel, take turns to execute different parts of the work bringing out the same performance as one processor taking charge of all the work.

All reported results from above performed experiments show that speed-up becomes limited beyond 8 processors in this external memory constrained environment.

IV. PARALLELIZATION IN DIFFERENT PROGRAMMING LANGUAGES

Characteristics of full applications found in scientific computing industries today lead to challenges that are not addressed by state-of-the-art approaches to automatic parallelization. These characteristics are not present in CPU kernel codes non linear algebra libraries, requiring a fresh look at how to make automatic parallelization apply to today's computational industries using full applications. Therefore, parallelization that is specific to a Language and is optimized for that language is necessary. For multifunctional applications, the compiler must assume that all combinations of choices are possible since the compiler cannot determine which of the choices a user will select. Multi-functionality causes the amount of compile-time analysis required to multiply as the compiler attempts to account for many possible control flow paths, precise analysis can become infeasible in terms of the analysis techniques required to compare array access patterns across control flow paths, even though the paths may never be taken within the same execution in practice.

Consequently, the compiler makes conservative assumptions which can reduce the compiler's ability of finding significant parallelism. To enable reuse and iterative development without requiring recoding of the application's execution framework, a layer of abstraction is added between the main execution process and the sub processes containing the computational techniques. Due to a layer of abstraction, compiler analysis and transformations must function with limited knowledge of the control flow across computational modules. The compiler can determine the control flow for portions of the code, such as the control flow within the code of a computational module in SEISMIC, but it is not feasible for the compiler to determine the full global control flow of large application suites since any computational module may follow any other. Without control flow information, analysis techniques, such as dataflow analysis, cannot be performed across a layer of abstraction. Data structures can be shared across the layer of abstraction. Consequently, the same data structures, allocated in outer contexts, may be used by multiple computational modules to house different types of data. State-of-the-art automatic parallelization techniques fail to perform precise comparisons among an array's accesses when the size and multidimensional shape of the representation of an array's accesses are not clearly defined portions of the size and shape used to describe the array's declaration. The result is that the compiler makes conservative assumptions that may limit the amount of parallelism the compiler is able to discover.

A larger loop nesting depth requires additional symbolic analysis for data dependence and array privatization analysis, resulting in a greater compile-time complexity for parallelization. Since the compiler analyzes an array reference for cross-iteration dependencies for each enclosing loop, the Range Test permutes the loops in a loop nest to determine which loops are parallel, and the compiler compares array references in different loops when the loops share a common enclosing loop, the amount of symbolic analysis required relates to the loop nesting depth. The amount of symbolic analysis required also relates to the subroutine nesting depth since inter-procedural analysis or in-lining must be used to translate array access patterns within subroutines into the calling contexts of the subroutines in order to analyze cross-iteration dependencies in any loops enclosing the subroutine calls. Subroutines enclosing a loop can require symbolic analysis if an array referenced within the loop is declared in a calling subroutine. In order for automatic parallelization to become utilized in today's scientific computing industries, the mentioned challenges described must be addressed. During the course of this study I have studied many language specific parallelization technique. The best of each parallelization technique are discussed below.

A) Parallelization in the Language C

In the paper "Towards Effective Automatic Parallelization for Multicore" automatic parallelization is achieved by using polyhedral model. This is big challenge to automate parallelization of sequential codes. In C compiler parallelization also used but virtually but this is rare used by developer because it is not effected. In this method polyhedral model is used to program transformation and for data dependencies. Automatic parallelization has been available in commercial compilers for many years. But unlike vectorization technology, which was indeed heavily used in practice by developers of production application codes on vector machines, automatic parallelization across multiple processors has not yet been sufficiently effective to draw much interest from application developers. Intel's production compiler incorporates automatic parallelization and automatic vectorization. But in its automatic vectorization capability is very good, its automatic parallelization is not effective.

The polyhedral model provides a powerful abstraction to reason about transformations on such loop nests by viewing a dynamic instance (iteration) of each statement as an integer point in a well-defined space called the statement's polyhedron. The optimization of polyhedral model on parallelization is viewed in three terms.

1) Static dependence analysis of the input program
2) Transformations in the polyhedral abstraction
3) Generation of code for the transformed program

Now a day's larger number of processing elements are on single chip. That led to multi-core architecture and parallelization. At the end researcher summarize that the polyhedral model for transformation provides a powerful basis for the system, and recent advances have made it feasible to use with non-toy codes and also researcher work with programmers to automate parallelization to achieve efficient and effective parallelization.

B) Parallelization in JAVA

The paper proposes and evaluates an approach for automatic parallelization which uses traces as units of parallel work. A trace is a sequence of unique basic blocks which are executed in sequential order during the execution of a program. A trace collection system is used to monitor a program's execution and generate traces based on it. It starts recording a trace when occurrences of certain events exceed a specific threshold. Once the traces are collected, they are used for optimization.

An offline feedback directed system is used which monitors the execution of a program and collects information that is used to optimize the program. It requires two executions as preliminary and primary. The preliminary execution is the one in which the information is collected. The primary execution is the execution of the program after it is optimized using this information. Offline feedback systems analyze collected information more thoroughly since they do not compete with the executing program for resources in contrast with online systems which require only one execution of a program but have to compete for resources.

Using traces as automatic parallelization offer benefits firstly traces are based on a binary representation of a program without the need of examine the source code. Secondly traces include loop iterations and methods as units of parallel work exhibiting both data and task level parallelism. Thirdly traces are collected by keeping track of program execution and are relatively simple to identify. Collecting traces, extracting and packaging traces, selecting the traces that are to execute in parallel, scheduling the selected traces to execute on multiple processors and executing the scheduled traces are five steps to execute trace in parallel. Each step has its own challenges. First challenge is to collect traces efficiently, secondly to package traces into a form that can execute in parallel.

Third challenge is to determine the likely successors of trace; fourth challenge is to effectively distribute traces among multiple processors while the last challenge is to provide a mechanism for executing traces on multiple processors. A single threaded program is transformed by extracting its traces and packaging them into methods that are suitable for parallel execution. The transformation performs six steps for each trace that exists in a method: two to extract individual traces and four to package them. An infrastructure is created that extracts packages and executes traces. When a program's frequently executed methods are optimized, infrastructure is called and transforms the methods by extracting traces from these methods and packaging the traces in their own methods. Methods are compiled one at a time and they cannot share intermediate representation data such as instructions and variables.

The overall improvements in performance of the applications are measured by techniques using speedup. Speedup is the ratio of the sequential execution time to the execution time of the parallel version of the program. Experimental evaluation indicates that system effectively parallelizes a number of programs that exhibit data level parallelism. The geometric mean of the speedups on four processors is 2.03, and the best speedup is 2.76. Results indicate that grouping of traces is essential to good performance. Thus, this indicates that trace based parallelization is promising and can be realized efficiently.

C) A Compile-time Cost Model for Automatic OpenMP Decoupled Software Pipelining Parallelization

In the paper "A Compile-time Cost Model for Automatic OpenMP Decoupled Software Pipelining Parallelization" it is proposed to use pipeline parallelism in ordinary programs that cannot be dealt with by traditional techniques. Here a compile-time cost model for automatic parallelization is used for profit estimate by extending the existing cost model in Open64 loop nest optimizer (LNO). The researchers improved this DSWP model to increase the efficiency of parallelization. Researcher said that we evaluate our cost model with loops containing complex memory access patterns and control flow structure but Load balance is necessary for parallelization otherwise this is not effective.

This algorithm is not restricted by CPU architecture and hardware support it means that this algorithm is not platform limited. This algorithm has two overheads one is scheduling problem and the other is bad load balance. Keeping load balancing among parallel threads is a key problem in achieving performance. OpenMP is also implemented by DSWP transformation. In this paper existing model Open64 model is used to partition threads, this is also used in program transformation that also improve the automatic parallelization process. In this paper researchers include cost model.

By experience they improve this model by including this cost. This model is applicable and efficient. "Performing a profit analysis both accurately and efficiently is very hard, since whether or not a parallel program is profitable relies on many factors, including its available parallelism and the manner in which it is exploited, compiler optimizations, runtime support, data layout, operating system noise, and workload balancing and so on. Many compilers and runtime libraries have an internal cost model that helps evaluate compiler transformations, guides the compiler in its optimization process and helps achieve load balancing". It can be further improved by extending the processor model and Cache model for multicore platforms.

V. CONCLUSION

A number of factors were found that affect the performance of the automatic parallelization. First is for programmer has to write the program in such a way that is easily divisible into multiple independent parts that are easily parallelized automatically but this tough job. Migration overhead also included in this system because in multicore system program threads are move from one processor to other. Main purpose is to increase the speed of execution of the program that is achieved by increasing the clock frequency and also increased by cores of processors. We increase performance by parallelization. Solution is to utilize the ubiquitous commodity of the modern world-The Multicore Processors. But it is a great challenge to convert single thread into multiple threads. This clearly shows that the performance of the program developed for this research has increased if it is executed in a multi-threaded environment. The important thing to note here is that the overhead that occurs does have a significant effect on the execution time. The execution time should have increased two times as the first PC had an 8 thread CPU to perform the execution as compared to this PC with only 4 threads. But when we calculate the results more precisely then this result are unpredictable in some cases but we improve these results in future by using different techniques of automatic parallelization.

REFERENCES

[1] Bøegh, Jørgen, Stefano Depanfilis, Barbara Kitchenham, and Alberto Pasquini. "A method for software quality planning, control, and evaluation." *IEEE software* 2 (1999): 69-77.

[2] Huo, Ming, June Verner, Liming Zhu, and Muhammad Ali Babar. "Software quality and agile methods." In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pp. 520-525. IEEE, 2004.

[3] Boehm, Barry W. "A spiral model of software development and enhancement." *Computer* 21, no. 5 (1988): 61-72.

[4] Mnkandla, Ernest, and Barry Dwolatzky. "Defining agile software quality assurance." In *Software Engineering Advances, International Conference on*, pp. 36-36. IEEE, 2006.

[5] Royce, Walker. "CMM vs. CMMI: From conventional to modern software management." *The Rational Edge* (2002): 2-9.

[6] Hong, G. Y., and T. N. Goh. "Six Sigma in software quality." *The TQM Magazine* 15, no. 6 (2003): 364-373.

[7] Manzoni, Lisandra V., and Roberto T. Price. "Identifying extensions required by RUP (rational unified process) to comply with CMM (capability maturity model) levels 2 and 3." *Software Engineering, IEEE Transactions on* 29, no. 2 (2003): 181-192.

[8] Ince, Darrel C. *Introduction to software quality assurance and its implementation*. McGraw-Hill, Inc., 1995.

[9] Rowlingson, Robert, and Richard Winsborrow. "A comparison of the Payment Card Industry data security standard with ISO17799." *Computer Fraud & Security* 2006, no. 3 (2006): 16-19.

[10] Mathur, Aditya P. "Performance, effectiveness, and reliability issues in software testing." In *Computer Software and Applications Conference, 1991. COMPSAC'91, Proceedings of the Fifteenth Annual International*, pp. 604-605. IEEE, 1991.