A Conceptual Paper on Exploitation of Parallelism for Existing Code Base

Pradip S. Devan[#1], R. K. Kamat[#2]
#1 Department of Computer Science, Shivaji University, Kolhapur – 416 004 E-mail: pradip.devan@gmail.com
#2 Department of Electronics, Shivaji University, Kolhapur – 416 004 E-mail: raj_kamat@yahoo.com

ABSTRACT:

Business demands for better computing power because the cost of hardware is declining day by day. Therefore, existing sequential software are either required to convert to a parallel equivalent and should be optimized, or a new software base must be written. The factors outlined in this paper are analyzed the current business demands and need of parallelism of existing sequential source code. To address these requirements, we reviewed the ongoing research in parallelization and we conclude some solution approaches.

Key words: Complier; Parallelization; computing etc.

INTRODUCTION:

Many of the industries had followed traditional sequential programming as per hardware demand i.e. for single core. An algorithm is constructed and implemented as a serial stream of instructions; only one instruction may execute at a time i.e. instructions execution proceeds sequentially one by one [1-4]. The demand for computational power has grown beyond what the present uniprocessor technology can offer; therefore parallel programming seems to be the most logical way to meet the demand. Some parallel languages such as SISAL [5] and PCN [6] have found little favor with application programmers; however industries prefer to use their traditional sequential programs rather than learning a completely new language only for parallel programming.

Performance scaling across processor was relied on increasing uniprocessor clock frequency previously for all programs; so significant code changes were not required to improve software performance. However, clock frequency scaling has hit the power wall; and hence processor manufacturers are now using the counting growth in transistor to place multiple cores [7]. Machines with four or more cores are already in market and it could scale up to 1000-core+ in the next decade. Many new languages have been proposed to encourage development of multi-threaded applications and to relieve the burden of writing parallel programs [8, 9, 10]. However these languages are offering a good and easier parallel programming for the new application which can be written by same languages but not for the existing codes which are written in single thread. Manual conversion of the existing single threaded application into multi-threaded is very tedious/difficult job. It increases the complexity and software development cost because of rewriting legacy code; programmer's training, efforts to avoid the deadlocks, race conditions and other problems associated with parallel programming.

Auto-parallelization techniques extract threads for multi-threaded application from single-threaded one without any programmer's interventions. This process doesn't have any programmers' involvement except the preparation of new build and it doesn't suffer additional cost or any complexity. Hence these techniques are preferred over manual parallelization. Auto-parallelizing compilers that take as input sequential code and produce parallel code as output have been studied in the scientific computing domain for many years. However these techniques have not proven successful enough for broad adoption. Many research techniques (such as Cilk, OpenMP) are focusing on changes or extensions to existing programming languages to enable easier specification of parallelism [8, 9, 10]. Unfortunately, these approaches do not alleviate all problems of parallel programming.

MOTIVATION:

At present, business demands for better computing power because the cost of hardware is declining day by day. Therefore, existing sequential software are either required to convert to a parallel equivalent and should be optimized, or a new software base must be written. However, both options require a skilled developer in dependence analysis. Converting these software in multithreaded for parallel computation increases the complexity and cost involved in software development due to rewriting legacy code, efforts to avoid race conditions, deadlocks and other problems associated with parallel programming. Parallel programming is a difficult and expensive process. While tools have been developed to assist with parallelization, which still require the user with a deep knowledge of parallelization--a topic not within the grasp of most users who could benefit from parallelizing, especially when the programs at issue are inherently parallel.

RESEARCH PROBLEM:

To manage power consumption and heat dissipation, the clock speed for each "core" in the processor is slower than previous generations. For example, an earlier processor with a single core that ran at 3.0 Ghz is being replaced with a dual or quad-core processor with each core running in the neighborhood of 2.6 Ghz. In this case, though each core is a bit slower; total processing will be increase. Increasing performance will continue to come from more cores and not faster. Hence the single-threaded applications will give poor performance on multi-core server as compared to previous generation i.e. these additional cores often provide no benefit. A large percentage of mission-critical enterprise applications will not run "auto magically" faster on multi-core servers. In fact, most of application will run slow.

Most enterprise applications are not programmed to be multi-threaded. A single-threaded application cannot take advantage of the additional cores in the multi-core processor without sacrificing ordered processing

We aim to develop a methodology as a part of compiler capable of automatically generating parallel specification starting from a sequential computing intensive (loop) code specification.

BACKGROUND OF RESEARCH WORK:

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions. These instructions are executed on a central processing unit. A single-threaded application cannot take advantage of the additional cores in the multi-core processor without sacrificing ordered processing. Only one instruction may execute at a time. Once the instruction is finished, the next can be executed [13]. On the other hand, parallel programming software uses multiple processing elements simultaneously to solve a problem. This is accomplished by dividing the program into independent parts so that these all independent parts execute by different processing element simultaneously. However this is a difficult and expensive process. Though, some tools have been developed to assist parallelization; users require deep knowledge of parallelization.

Moore's law stated that the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately after two years. The period often quoted as "18 months" is due to Intel executive David House, who predicted that period for a doubling in chip performance. It means that processing capacity doubles every 18 month. Current trend is showing up in all market will always demand for faster, lighter, more compatible, and overall better hardware/software. Multi-core computing systems are delivering performance with increasing transistor densities; however this potential cannot be realized unless the application has been well parallelized

SIGNIFICANCE OF RESEARCH WORK:

As the demand for computational power has grown beyond what the present uniprocessor technology can offer, parallel computing seems to be the most logical way to meet the demand. In parallel programming, the computational speed is increased by using multiple processing elements (PE's) to cooperatively solve a single problem.

The compilers will have the ability to analyze the tasks that can be safely and efficiently executed in parallel. Automatic parallelization can result in shorter execution times. The lines of code and development cost is comparatively high; hence the ultimately goal of auto-parallelization is saves potential cost.

It also relieve the programmer from -

- Searching for loops that are good candidates for parallel execution
- Performing dataflow analysis to verify correct parallel execution
- Adding parallel compiler directives manually

ONGOING RESEARCH IN PARALLELIZATION:

Most research compilers consider FORTRAN programs only for automatic parallelization. FORTRAN programs are simpler to analyze as compared to C/C++ programs. Typical examples are:

1) Vienna Fortran compiler

2) Paradigm compiler

3) Polaris compiler

4) SUIF compiler

They have focused on changes or extension of existing programming languages to enables easier specification of parallelism [8, 9, 10] i.e. they tried parallelism in commutative fashion by removing serialization. OpenMP, provides automatically schedule parallelism for performance but provide little help to achieving correctness. Unfortunately, these approaches do not alleviate all problems of parallel programming

Memory transaction [11] could be an alternate technique for specific paradigm, specialized languages such as streamlt [12] can handle the correctness issues and schedule the parallelism for good performance.

However many programs and data structures are not fit in this techniques; so must be written manually from scratch to take advantage.

Many researchers groups are working on the development of automatic parallelization from different points of views. There are several well-known research groups involved in the development and improvement of parallel compilers such as:

- OpenMP Group [14]
- HPFF [15]
- Rouge Wave [16]
- SUIF [17]
- MPI Group [18]
- Intel [19]
- The portland group [20]

PARALLEL PROGRAMMIMG APPROACHES:

There are basically two main approaches for parallel programming:

i) Implicit parallelism:

This is common approach to parallelization of sequential code called as parallelizing compilers. This approach allows programmers to write their programs without any concern about the exploitation of parallelism and has been followed by parallel languages and parallelizing compilers. Exploitation of parallelism is instead automatically performed by the compiler and/or the runtime system and hence user cannot control the scheduling of calculations and/or the placement of data. In this approach parallelism is transparent to the programmer, maintaining the complexity of software development at the same level of standard sequential programming.

The ultimate goal of this approach is to relieve the programmer from the parallelizing tasks. However, this is still very hard to achieve and much beyond the scope of current compiler technology. This approach could be divided in two parts as:

a) **Automatic Parallelization:** Compiler would accept the sequential code and produce efficient parallel object code without any (or very little) annotation by programmer

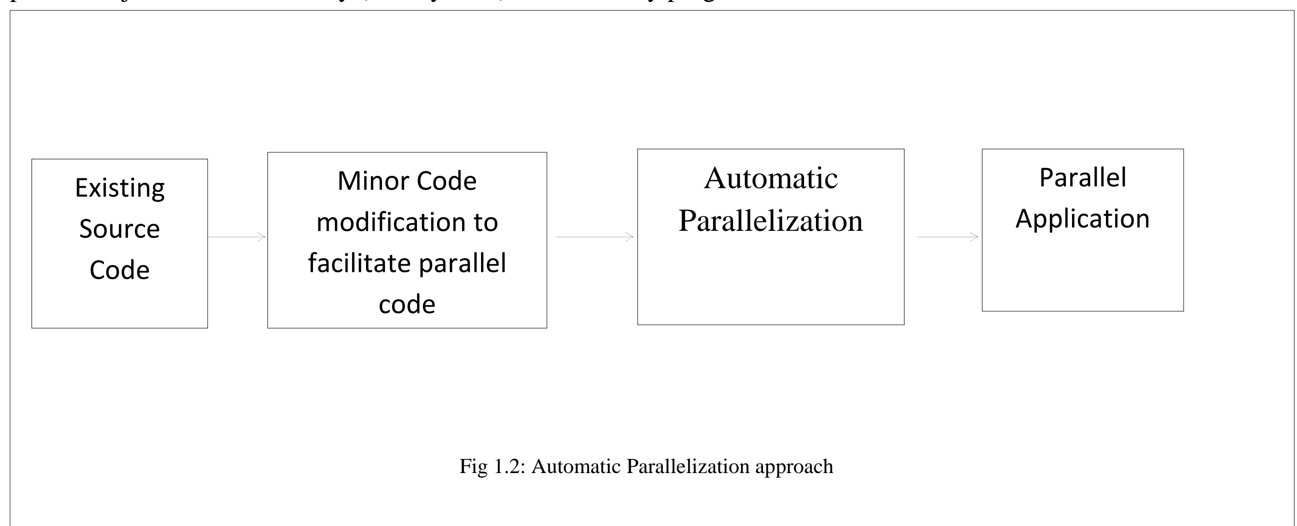

Fig 1.2: Automatic Parallelization approach

b) **Semi-Automatic Parallelization:** Programmer generally uses a pragma or similar notation to provide hints to compiler about parallelism can be done for the code snippet.

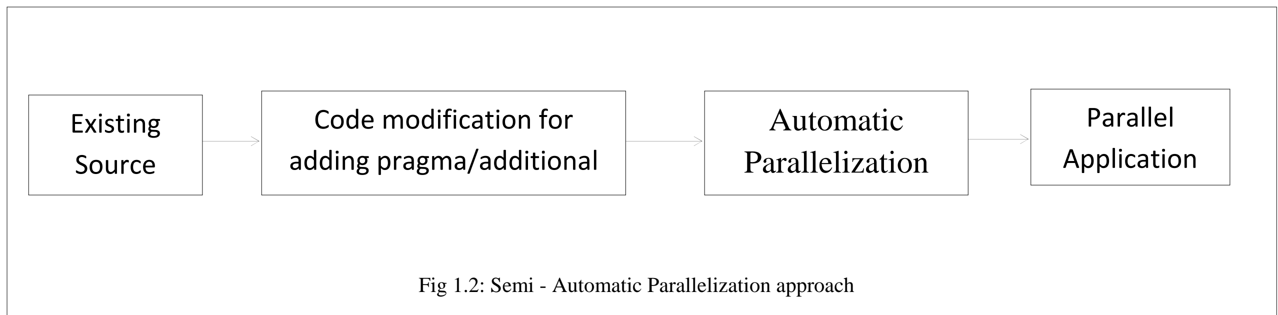

Fig 1.2: Semi - Automatic Parallelization approach

ii) Explicit parallelism:

In this approach, the programmer is responsible for most of parallelization efforts such as fork/join primitives, semaphores etc. This allows the flexibility i.e. any form of parallel control can be implemented. However it increases additional burden to handle complexity of decomposition, mapping tasks to processor. This approach completely based on the user's judgment that how parallelism exploited for particular application. Mainly parallelizing libraries and rewriting the existing sequential code into parallel are considered in this approach.

a) Parallel Libraries: This is most straightforward approach to parallelizing an application and useful when the existing parallel libraries match well with the needs of the application. The main idea behind this approach is to encapsulate parallel code which shares multiple applications can be implemented in very efficient way into parallel library. Such library can be reused by several codes. For example parallel implementation of some mathematical routine can be used in several codes.

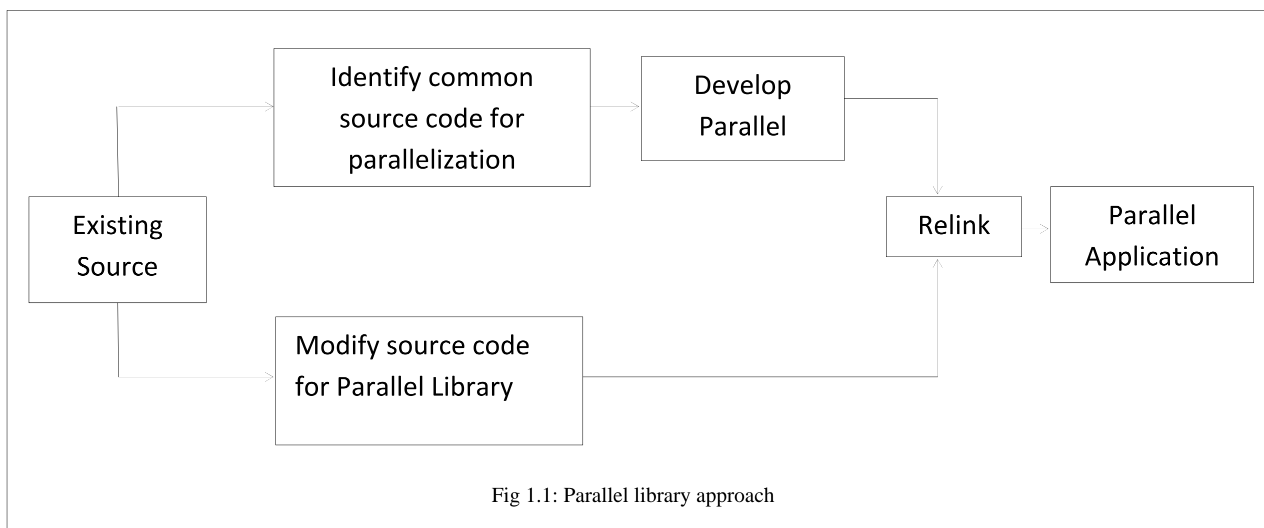

Fig 1.1: Parallel library approach

b) Rewrite Existing Code to Expose Parallelism: In this approach, programmer has to write a parallel application from the very beginning; which gives more freedom to choose programming language and the programming mode. However, rewriting existing sequential code base to parallel is very difficult job for programmer as it increases the complexity. Also industries are not ready for this approach as its increases the software cost because of rewriting legacy code, programmer's training etc.

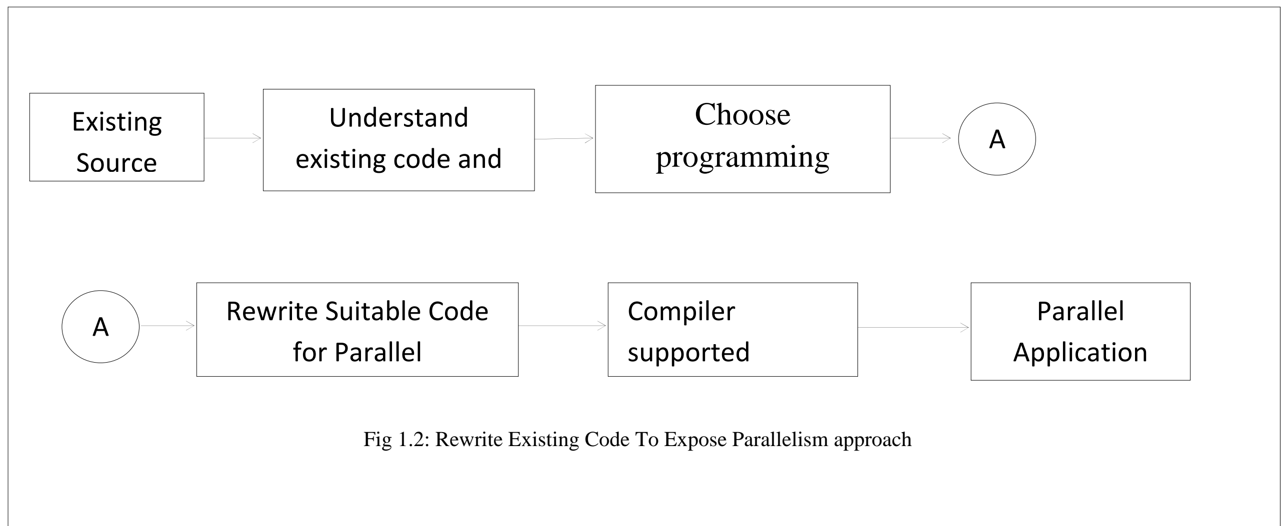

Fig 1.2: Rewrite Existing Code To Expose Parallelism approach

CONCLUSION:

The existing large numbers of sequential software are needed to be convert parallel equivalent software to meet today's business demands. After considering complexity and cost involved in software development due to rewiring legacy code; the auto-parallelization or semi-auto parallelization techniques are necessary to exploit parallelism. Unfortunately, these techniques have yet to extract sufficient parallelism.

In this paper we have described approaches for parallelizing the existing sequential source code and ongoing research on same. Many advances have been achieved in parallel software development but there is still considerable work needed in order to effectively exploit the computational power of new generation CPU/GPU. Most of the research work is happening in the area of semi and auto parallelization due to its cost benefits. The outcome of these research works are slowly adopted by few of the mainline and commercial compilers.

REFERENCES:

[1] Barney, Blaise. "Introduction to Parallel Computing", Lawrence Livermore National Laboratory. http://www.llnl.gov/computing/tutorials/parallel_comp/
[2] Foster, "Designing and Building Parallel Programs". http://www-unix.mcs.anl.gov/dbpp/
[3] A. J. van der Steen, and J. Dongarra. "Overview of Recent Supercomputers". www.phys.uu.nl/~steen/web03/overview.html
[4] F. Corbera, R. Asenjo and E. Zapata. "Accurate Shape Analysis for Recursive Data Stuctures" Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, 2017 (2001) 1-15
[5] J. Feo, D. Cann, and R. Oldehoeft. A Report on the SISAL Language Project. Journal of Parallel and Distributed Computing, vol 10, pages 349-366, 1990.
[6] I. Foster and S. Tuecke. Parallel Programming with PCN. Technical Report ANL-91/32, Argonne National Laboratory, Argonne, December 1991.
[7] T. N. Mudge. Power: A first-class architectural design constraint. IEEE Computer, 34(4):52–58, 2001.
[8] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In Conference on Programming Language Design and Implementation, 2006.
[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In Conference on Programming Language Design and Implementation, 1998.
[10] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J.Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communicationexposed architectures. In International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.
[11] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C.Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In Conference on Programming Language Design and Implementation, 2006
[12] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J.Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communicationexposed architectures. In International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.
[13] [13] Foster, "Designing and Building Parallel Programs". http://www-unix.mcs.anl.gov/dbpp/ [15] A. Grama, A. Gupta, G. Karypis, and V. Kumar. "Introduction to Parallel Computing". http://www-users.cs.umn.edu/~karypis/parbook/
[14] OpenMP Group : http://openmp.org/wp/
[15] HPFF : http://hpff.rice.edu/
[16] Rouge Wave: http://www.roguewave.com/
[17] SUIF : http://suif.stanford.edu/
[18] MPI Group : ttp://www.mcs.anl.gov/research/projects/mpi/
[19] Intel:http://software.intel.com/sites/products/documentation/studio/composer/en-us/2009/compiler_c/optaps/common/optaps_qpar_par.htm
[20] The portland group : http://www.pgroup.com/products/pgicdk.htm, http://www.cs.cmu.edu/~scandal/research-groups.html