IJCSE International Journal of Computer Sciences and Engineering Open Access
Survey Paper Vol.-7, Issue-7, July 2019 E-ISSN: 2347-2693

Techniques of Parallelization : A Survey

Vijay Kumar[1*], Alka Singh[2]

[1,2]Department of Computer Science K.N.I.T, Sultanpur, UP, India

Corresponding Author: Vijay Kumar_vijaykumar5391@gmail.com, Tel.: 9559142142

DOI: https://doi.org/10.26438/ijcse/v7i7.150154 | Available online at: www.ijcseonline.org

Accepted: 10/Jul/2019, Published: 31/Jul/2019

Abstract— Parallel computing enables us to utilize hardware resources efficiently and to solve computationally intensive problems by dividing them into sub-problem using a shared-memory approach and solving them simultaneously. Emerging technologies are based on parallel computing as it involves complex simulations of real-world situations which are extremely computation-intensive and time-taking as well. Parallel programming is gaining significance due to the limitations of the hardware. Researchers are trying to enhance memory and bus speed to match the processor's speed. Generating parallel code requires skill and a particular technique of parallelization. There are several parallelization techniques amongst which one needs to be shrewdly chosen for a particular task and architecture. A brief survey of existing parallelization procedures is provided through this paper. New hybrid techniques are required to be developed combining technical and architectural benefits two or more parallel models. A thorough revision of traditional parallelization techniques is required to derive new techniques.

Keywords— Shared Memory; Parallel programming; Parallelization techniques.

I. INTRODUCTION

An exceptional move towards parallel processing from sequential computing is essentially due to Von Neumann bottleneck, where latency being significant problem. A slow growth had been observed in rate of clock speed of the processors in the recent years. Hence to improve the performance of an application parallel processing is the best alternative. Parallel computing provides solution to large problem by using multiple cores or CPUs concurrently, thereby saving execution time. Moore's law has provided a significant impact in software and hardware industry. For the proficient effect of the hardware, the multi-threaded or multi-process programming must be composed. Hence computing demands parallelism in future scope. Composing parallel code manually is a time taking procedure. Hence to achieve tremendous performance and functionally precise parallel programming, automatic parallelization is essential. It changes sequential code to parallel code for the reason to make simultaneously utilization of the multicore architecture. This parallel code can be used to run on multicore system [1].

II. TYPES OF PARALLELISM

A. Hardware:

1) *single processor:*
   - VLIW
   - Pipelining
   - Superscalar
2) *SIMD, Vector processors, GPUs*
3) *Multiprocessor:*
   - Symmetric shared memory
   - Distributed-memory
   - Multi-cores
4) *Multicomputer /clusters.*
5) *Classes of Parallel Computers*
   - Cluster
   - Multiprocessor
   - Multicore Computing
   - Network Processor

B. Parallelism in Software

- Bit level partition
- Instruction level parallelism
- Task-level parallelism
- Data parallelism
- Transaction level parallelism

1. *Parallelism in Applications:*

a) *At Instruction level:*
   - Several instructions executes from the same instruction stream in concurrent fashion.

b) *At task level:*
   - Concurrently executing multiple threads from the same application.
   - Compiler or user generated whereas managed by compiler and hardware.
   - Confined in practice by:

-communication/synchronization overhead.
-Algorithmic Attributes.

c) *At data level:*
- Single stream instruction concurrently drives numerous data.
- Limited by irregular data handling patterns and by memory bandwidth.

d) *At transaction level:*
- Several Threads from distinct transactions can be executed concurrently.
- Limited by concurrency overheads.

e) *Task Partitioning Algorithms*
- Analysis Scheme
- Construction Scheme

f) *Task Mapping Algorithms*
- Graph Theoretic Algorithms
- Mathematical Programming
- Heuristic Algorithm

III. PARALLELIZATION TECHNIQUES

Parallelization process involves*:*

1. Complex programs are partitioned to task and mapping it to different processors based on partitioning and mapping algorithms.
2. Synchronization and communication is maintained among processors.
3. Loop parallelization and Dependency analysis are the most significant parallelization techniques [1].

A. Loop Parallelization

Loop parallelization is of greater significance since 90% of the execution time is mostly due to loops in the code. Distribution of loop elements into chunks and allotting to various processors will lessen the latency. In loop parallelization, parallelizing nested loop efficiently is a challenging task [1].

B. Dependency Analysis

Dependency analysis is extremely significant in parallelization process. For parallel execution of two statements there should be no dependency between them. Hence the removal of these dependencies is important to make code parallelizable. Through complex analysis these dependencies between statements are identified [1].

C. Full Replication

One basic method for keeping away from race conditions is to recreate the reduction object and make one duplicate for each thread. The duplicate for each thread should be introduced at the beginning. Each thread basically updates its very own duplicate, consequently keeping away from any race conditions. After the local reduction has been performed utilizing every data items on a specific node, the updates made in every one of the duplicates are combined. The other four procedures depend on locking [2]. The memory format for these four methods is appeared in Figure 1.

D. Full Locking

One clear solution to avoiding race conditions is to relate one lock with every element in the reduction object. After the data processing task, a thread requires to obtain the lock linked with the element in the reduction object it needs to update. As in the apriori mining algorithm, for each candidate, there is a lock connected with the count, which is required to be procured before updating that count. If a candidate's count is required to be updated by two threads, one should wait for the other to release the lock first. A large number of candidates are considered during any iteration in, so the probability of one thread to wait for another thread is extremely small [2][3]. Supporting too many locks results in three kinds of overheads:

1) Large number of locks requires large memory.
2) Overhead due to cache misses.
3) Overhead due to false sharing [3].

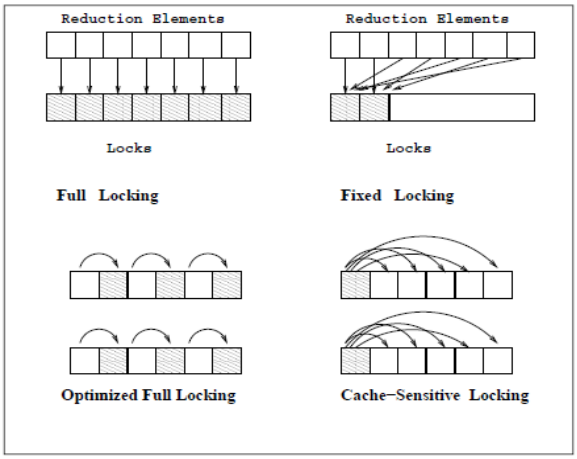

Figure 1. Locking Schemes with their memory layout.

E. Optimized Full Locks

The next scheme we describe is optimized full locks. As illustrated in figure 1, to overcome cache misses related to full locking scheme optimized full locking scheme allocate a reduction element and the related lock in continuous memory addresses. By proper alignment and padding, it can be assured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity cache miss. The possibility of false sharing is also reduced. This is because there are fewer elements (or locks) in each cache block. This scheme does not reduce the total memory requirements [2].

F. Fixed Locking

The fixed locking scheme is projected to diminish the memory overheads required by locks in full locking and

optimized full locking schemes. A fixed number of locks are used in fixed locking schemes. The number of locks chosen is a parameter to this scheme. If the number of locks is '*Lc*,' then the element '*Ie'* in the reduction object is assigned to the lock *Ie mod Lc*. Two locks are used in figure 1[2].

G. Cache-Sensitive Locking

This technique collaborates with the ideas from fixed locking and optimized full locking. Suppose an N byte cache block and a K byte reduction element. For a particular cache block, a single lock is used for all reduction elements in that block. Consequently, every cache block will have 1 lock and N/K-1 reduction elements.

This scheme results in lower memory requirements than the full locking and optimized full locking schemes. Cache-sensitive locking reduces each of three types of overhead associated with full locking.

Table 1. Tradeoff among the Techniques

	Full Replication	Optimized full Locks	Cache Sensitive Locks	Full locks	Fixed locks
Memory Requirement	Very high	High	Low	High	Low
Parallelism	Very high	High	Medium	High	low
Locking Overhead	None	Medium	High	Medium	High
Cache Misses	Low	Medium	Low	High	medium
False Sharing	None	Yes	None	Yes	Yes
Merging Costs	Yes	None	None	None	None

The problem of false sharing is also reduced because there is only one lock per cache block.

H. Reduction Parallelization Techniques

The occurrence of loop dependence pattern in loops which apply the reduction process. Reduction occurs at the core of large number of applications and algorithms –scientific and otherwise [4].

Reduction parallelism requires two tasks:

1) Reduction operation parallelization.

2) *Recognizing the reduction variable:* Techniques involved in recognition of reduction variable:

a) Static reduction recognition: Recognition of reductions statement is carried out at the time of compilation by syntactic pattern matching the iterative statements with generic reduction template and then performing data dependency inspection of the variable in order to assure that it is not used anywhere else except reduction statement [5].

b) Run-time reduction validation: In cases where compile-time dependency analysis is not possible then in that case reduction has to be validated at run-time [6][7][8].

I. AUTOMATIC PARALLELIZATION

A semi-automatic parallelization system SUPERB that is being developed at the University of Bonn within the framework of the SUPRENUM project [9]. It is designed to merge MIMD and SIMD [10] techniques. It's applicable for large-scale technical computing. [11][12][13][14].

J. CLUSTER PARALLELISM

Basing on explicit and implicit cluster parallelism, a model of an expert system to develop a "smart" (intellectual) compiler. The present research looks into quantitative structural associations between clusters and, accordingly, microprocessors in Embedded Computing Systems and defines routes of data transfer inside a cluster or between different clusters in the suggested smart compilation, based on analysis of algorithms [15].

K. Rank-Level Parallelism in DRAM

The hierarchically organized DRAM system maintains Channel-Rank-Bank structure. Hierarchical connection comprises of channel to ranks (multiple ranks), and each ranks to multiple banks connections hence facilitating parallelism in DRAM shown in Fig.2 [16].

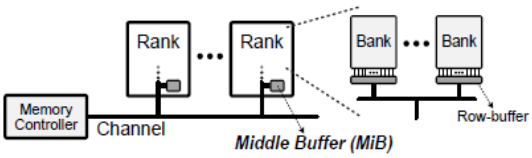

Figure 2. Rank level parallelization

L. Memory-Level Parallelism

Problem due to time consuming memory accesses degrades system performance. Memory level parallelism enhances performance through concurrent memory accesses [17][18].

M. Hybrid Approach for Parallelization

Analyzing the parallel segments in the code by performing block-level analysis, function level analysis, and instruction-level analysis.These parallel segments are executed in parallel over different computers. Hybrid Approach comprises of block level analysis and functional level analysis [19][20][21].

IV. CONCLUSIONS

Program analysis and transformation strategies have been developed in order to attain the proper parallelism. A program calculus is now emerging that allows the formal analysis of these transformations as well as the development of new powerful transformations. Although it is not clear

how close we are to that goal, it is clear is that we are not there yet and that our research effort must continue because of the great impact that effective parallelizers are bound to have on the ordinary users acceptance of parallel machines.

REFERENCES

[1] Prema, S.,and R. Jehadeesan,"*Analysis of Parallelization Techniques and Tools,"* International Journal of Information and Computation Technology 3 (2013): 471-478.

[2] Jin, Ruoming, and Gagan Agrawal,"*Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance*" Proceedings of the 2002 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2002.

[3] John L. Hennessy, and David A. Patterson, "*Computer Architecture: A Quantitative Approach*"Morgan Kaufmann,Inc.,San Francisco,2nd edition,1996.

[4] Yu, Hao, and Lawrence Rauchwerger, "*Adaptive reduction parallelization techniques*" ACM International Conference on Supercomputing 25th Anniversary Volume. ACM, 2014.

[5] Chapman, Barbara, and Hans Zima, "*Supercompilers for parallel and vector computers*" Addison-wesley, 1990.

[6] Zhang, Ye, Lawrence Rauchwerger, and Josep Torrellas,*"Hardware for speculative run-time parallelization in distributed shared-memory multiprocessor*." Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture. IEEE, 1998.

[7] Rauchwerger, Lawrence, and David A. Padua,*"The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization"* IEEE Transactions on Parallel and Distributed Systems 10.2 (1999): 160-180.

[8] Yu, Hao, and L. Rauchwerger, *"Run-time parallelization overhead reduction techniques*" Proc. of the 9th Int. Conf. on Compiler Construction, Berlin, Germany. 2000.

[9] Banerjee, Utpal, et al, "*Automatic program parallelization"* Proceedings of the IEEE 81.2 (1993): 211-243.

[10] P.M. Behr, W.K. Giloi and H. Mfihlenbein, "*SUPRENUM: The German supercomputer architecture—Rationaleand concepts, Proc"* 1986 International Conference on Parallel Processing (1986).

[11] MJ. Flynn, *"Some computer organizations and their effectiveness"*, IEEE Trans. Computer. 21 (9) (1972) 948-960.

[12] U. Trottenberg, *"SUPRENUM--an MIMD multiprocessor system for multi-level scientific computing"*, in: W. H~indler et al., eds., CONPAR 86. Conference on Algorithms and Hardware for Parallel Processing, Lecture Notes in Computer Science 237 (Springer, Berlin, 1986) 48-52.

[13] H.P. Zima, H.-J. Bast, M. Gemdt, P.J. Hoppen, *"Semi-automatic parallelization of Fortran programs"*, in: W. Hgmdler et al., eds., CONPAR 86. Conference on Algorithms and Hardware for Parallel Processing, Lecture Notes in Computer Science 237 (Springer, Berlin, 1986) 287-294.

[14] H.P. Zima, H.-J. Bast, M. Gerndt, PJ. Hoppen, *"SUPERB: The SUPRENUM Parallelizer Bonn"*, Research Report SUPRENUM 861203, Bonn University, (1986).

[15] Ruchkin, Vladimir, et al, *"Frame model of a compiler of cluster parallelism for embedded computing systems"*,2017 6th Mediterranean Conference on Embedded Computing (MECO). IEEE, 2017.

[16] Shin, Wongyu, et al, *"Rank-Level Parallelism in DRAM"*,IEEE Transactions on Computers 66.7 (2017): 1274-1280.

[17] Liu, De-feng, Guo-teng Pan, and Lun-guo Xie, *"Understanding how memory-level parallelism affects the processors performance"*,2011 IEEE 3rd International Conference on Communication Software and Networks. IEEE, 2011.

[18] Cheng, Shaoyi, et al, *"Exploiting memory-level parallelism in reconfigurable accelerators"*, 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2012.

[19] Kumar, K. Ashwin, et al, *"Hybrid approach for parallelization of sequential code with function level and block level parallelization"*, International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06). IEEE, 2006.

[20] Gasper, Pete, et al, *"Automatic parallelization of sequential C code"*, Midwest Instruction and Computing Symposium, Duluth, MN, USA. 2003.

[21] Wall, D,*"Limits of instruction level parallelism. David W. Wall, Limits of Instruction Level Parallelism"* Proc. 4th ASPLOS. 1991.

