International Journal of Engineering & Technology, 7 (4.6) (2018) 150-157

SPC

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Runtime Parallelization of Static and Dynamic Irregular Array of Array References

Parwat Singh Anjana[a], N. Naga Maruthi[a,]*, Sagar Gujjunoori[a], Madhu Oruganti[b],

a *Department of Information Technology*
Vardhaman College of Engineering
Shamshabad, Hyderabad, Telangana (501218), India
amananjana@gmail.com, maruthi.reddys@gmail.com, sagar.g@vardhaman.org,
b *Department of Electronics and Communication Engineering*
Sreenidhi Institute of Science and Technology
Hyderabad, Telangana (501301), India
**Corresponding author E-mail: oruganti.madhu@gmail.com,*

Abstract

The advancement of computer systems such as multi-core and multiprocessor systems resulted in much faster computing than earlier. However, the efficient utilization of these rich computing resources is still an emerging area. For efficient utilization of computing resources, many optimization techniques have been developed, some techniques at compile time and some at runtime. When all the information required for parallel execution is known at compile time, then optimization compilers can reasonably parallelize a sequential program. However, optimization compiler fails when it encounters compile time unknowns in the program. A conventional solution for such problem can be performing parallelization at runtime. In this article, we propose three different solutions to parallelize a loop having an irregularity in the array of array references, with and without dependencies. More specifically, we introduce runtime check based parallelization technique for the static irregular references without dependency, Inspector-Executor based parallelization technique for static irregular references with dependencies, and finally Speculative parallelization technique (BitTLS) for dynamic irregular references with dependencies. For pro ling the runtime information, shared and private data structures are used. To detect the dependencies between footprints and for synchronization of threads at runtime, we use bit level operations. A window based scheduling policy is employed to schedule the iterations to the threads.

Keywords:*Irregular References, Runtime-Check Parallelization, Inspector-Executor Parallelization, Speculative Parallelization,Computation, Operation, Footprint*

1. Introduction

In modern systems, parallelism is ubiquitously present in hardware: desk-tops and laptops have had multi-core processors for years and mobile de-vices are gaining access to multi-core processors like the ARM Cortex-A9 [3]. Parallelism has thus clearly moved beyond its traditional bastion of scientific domains. Due to the ubiquity of parallelism, many more applications are being parallelized than just the traditional scientific computing. While researchers have had years to understand and optimize for the parallelism patterns of dense matrix algorithms, used in high performance scientific c computing, the parallelism patterns in other types of irregular algorithms are not well researched. We de ne irregular algorithms as those that rely heavily on pointer-based data structures such as the graphs, trees or he linked lists [21]. These data structures are primarily used in many applications since the data is sparse and such data structures allow compact representation of large data. Examples of such data structures are found in many games engines and in the benchmarks such as STAMP[13], Lone Star or PARSEC [18] for example. An important characteristic of these algorithms is that the exact elements, and therefore memory location, they access are heavily data-dependent and are unknown until runtime. Moreover, unlike matrices, such data structures are highly dynamic and keep changing at runtime in terms of their sizes and shapes. This cripples potential compiler static analysis such as the traditional data dependence analysis used to efficiently parallelizing the dense matrix computations by checking the array indices. On the other hand, the performance demands of these algorithms are very high and must be met through parallelism.

The topic of parallelizing irregular, pointer based software is receiving a lot of attention. Different techniques approach the problem in different ways. Dynamic Parallel Java (DPJ) [16] is a language that provides a type and effect system that is verifiable at compile time allowing the compiler to make strong assertions about the effects of a particular operation on the data. This approach however, needs a programmer torewrite his code in the new language which is its downside. Work by Reid et al. [22] is similar work which also extends the type system by using a notion of static ownership of data. Each data object is owned by a logical owner (a parent class, a global owner known as "world" for global objects or any other object in the scope) and an effect system is also introduced for functions. The Galois programming model [11], [12] also takes a data-centric semantic approach to detecting conflicts. In the Galois model, the programmer defines whether or not two operations can commute: if they can, the two operations

can operate in parallel and if they cannot, the two operations mustbe serialized. Again in this work, the programmer must first understand thealgebraic commutativity property of the operations which may be difficult.Another approach that has been explored is the dynamic detection of parallelism such as Rinard'sJade [19]. This work dynamicallyannotates data structures and dynamically evaluates conditions to determine the potential overlap of operations. Work on serializer sets by Allen et al. [10] also seeks to dynamically extract parallelism by having the user specify serializers which correspond to operations that conflict with one another and therefore must be executed sequentially. As against these approaches, our approach plans to discover the parallelism by sampling references at runtime and then verifying the patterns are repetitive.

The key to parallelizing any computation or operations therefore is understanding the disjointedness between their data footprints: two computations with disjoint data footprints can be run in parallel regardless of their read/write status. Also, two computations that are not disjoint but have read-only behavior can also be parallelized. Our approach in this article is to define a computation as being composed of i) an operation and ii) a footprint as shown in Figure 1. An operation can be a primitive operation or memory operation(read/Write), it depends on the context in which it is used. For example, it can be an insert, search or delete operation, and it can be a memory read/write operation. A data footprint can be defined as memory references referred during the execution of computation by a thread. The key question, therefore, is whether we can determine data footprints efficiently. For solving the problem of data footprint determination, we first classify two different classes of references: static and dynamic. Then propose solutions that we would investigate one by one. Our primary focus will be on the loop having the irregular array of array references. Our proposed techniques reduce the memory requirement for profiling the information at runtime. We propose to use bit level computations to provide the thread synchronization and to detect the dependencies. By using the bit level computations, we are reducing the memory overhead and improve the performance as computations take less time for bit operations. Apart from this, for load balancing at runtime we used window based scheduling policies to schedule the iterations to the threads.

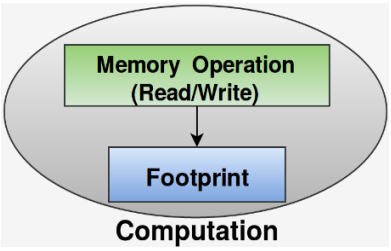

Fig. 1 Computation

2. Background

In compiler assisted automatic parallelization, a compiler is not only responsible for analysis and identification of parallel sections in the program but also responsible for generating the parallel code. The step of analyzing and determining the parallel section is one of the most crucial tasks because a compiler needs to prove that certain memory references accessed by different computations are disjoint and do not cause any dependence violation. Optimization compiler fails when it encounters the compile time unknowns. In most of the cases, the compile time unknowns depend on input data (input from a _le/command line/keyboard). A compile time unknown can be a memory reference (indices/pointers), loop bound, access pattern, or stride (WHILE loop). In this section, first, we understand the basic terminology which will be referred more frequently in the rest of the article.

2.1 Reference/Footprint

A reference is a value that allows a program to access particular data item stored in the memory indirectly [7]**.** Footprint gives all sort of memory aspect i.e. amount of main memory that a program uses or references while running [4]**.** In general, the meaning of memory footprint depends on the context in which it is being used. For our context, memory reference is considered as a memory footprint on which operations are being carried out during the program execution, and we use them interchangeably in the rest of the article.

2.2 Regular vs. Irregular Programs

Rauchwerger et al**.,** [17] identify two major classes of the programs based on the memory accesses, regular and irregular. In a regular program, a well-behaved analytical function can describe memory accesses and control flows statically at compile time. When a program does not depend on input data, memory accesses can be easily analyzed and extracted by the current state of the art optimization compilers. While on the other hand in an irregular program, memory access depends on the input data, which consequently cannot be predicted statically at compile time. Further, we identified two subclasses of irregular programs based on how subscripts of an array are resolved at runtime. When memory references are defined at the beginning of the execution and after the initial setup, they do not change. Such programs belong to the class of static irregular programs. When memory accesses are computation dependent and change from one execution phase to another, then they belong to the class of dynamic irregular programs.

As shown in Example 2.2.1, static irregular references will be fixed, as there will be no modification to the value stored in the array (B[]) used as an indirection for other array (A[]). Whereas, in dynamic irregular references as shown in Example 2.2.2 the value stored in the array (B[]) changes with execution. Therefore, the memory accesses become computation dependent.

```
for(i = 0; i < N; i ++){
……
    S1:    A[ B[i] ] = .....
                ……
}
```

Example 2.2.1: Static Irregular References

```
for(i = 0; i<N ; i++) {
    ……
S1: A[ B[i] ] = ....
S2: B[ i ] = f ( C[i] );
    ……
}
```

Example 2.2.2: Dynamic Irregular References

2.3 Runtime Checks

Runtime Checks are also known as guards or dynamic memory disambiguation, is short pieces of code that the compiler will generate to resolve ambiguity at runtime. Depending on the outcome of the runtime check, a sequential or parallel version of the originalloop is executed [9]**.** Checks can be as simple as single if-statements which take O(1) time[14]**,**[15] or can be a function call or a loop which takes O(n)/O(nlogn) time [20]**.** Sometime checks may have significant overhead and require O(n2) time. In the loop level parallelization using runtime check, a loop can be either executed in parallel or sequential it depends on the outcome of the runtime check. Compiler developers are looking towards adding runtime checks into the compiler to parallelize the loop

with a small number of dependencies and a high percentage of parallelism [9].

3. Methodology

Many runtime techniques have been implemented to parallelize programs with the statically unknown array of array references. In this article, we targeted the loops with static unknown array of array references. First, we distinguish two different classes of static unknown references, and then we propose different solutions to parallelize them. We classify the two classes as static and dynamic. In the static case, compile time unknown is known at runtime and is fixed. For such case, without executing the original loop, we enumerate the indices using runtime checks. Note that this could cause some additional over-head at runtime, but the benefits outweigh the cost. An Inspector-Executor technique is implemented by runtime checks to detect and resolve the dependencies. Runtime checks are generated at the compile time and using these checks we enumerate static unknown footprints/references at runtime and if they do not overlap, we parallelize them using Compiler Directives. When indices/footprints overlap, a parallel Inspector loop is generated to determine the dependencies and to prepare concurrent schedule, an Executor executes the loop parallelly using schedule created by Inspector. While on the other hand when a loop has dynamic irregular references Speculative parallelization is followed. Here loop iterations are executed parallel, and dependencies are determined and resolved at runtime. Loop iterations are re-executed in the case of dependence violation. Efficiently determining the disjointedness of thefootprint is one of the objectives to be solved. We classify the disjointednessof computations based on operation and footprint as shown in Table 1.

Table 1 Computation Classification

	Mem. Operation	Footprint	Dependence	Parallelizable
Computations	Not-Disjoint	Disjoint	No	Yes
	Disjoint	Disjoint	No	Yes
	Disjoint	Not-Disjoint	RAW/WAR	No
	Not-Disjoint (Read)	Not-Disjoint	No	Yes
	Not-Disjoint (Write)	Not-Disjoint	WAW	No

3.1 Disjointedness of Computations

When an array is accessed, the primitive operations can be an insert, search, and delete. However, here both an insert and a delete operation is quite expensive. Apart from this, in most of the array based programs, the size will be fixed and memory update operations are performed on the array. Therefore we define the computation as being composed of memory operation (read/write), and the array reference/footprint. We considered the array reference as a footprint for the classification.

3.1.1 Disjoint Footprints (1st and 2nd Row)

If footprints are disjoint, regardless of operation, two computations can always be parallelizable. Operations on different memory footprints will never cause any dependence. As shown in Figure 2, the Computation – 1=T 1 performing read/write, let on a[1], and Computation – 2=T 2 is performing read/ write, let on a[2] and no dependence between T1 and T2; therefore, they can be parallelized.

3.1.2 Non-Disjoint Footprints and Disjoint Operations (3rd Row)

In this case, two computations will always be dependent because of non-disjoint footprint and disjoint operations, which leads to WAR or WAR dependencies.

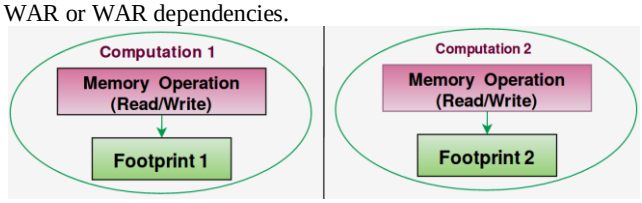

Fig. 2 Disjoint Footprints.

As shown in Figure 3, T1 performing a read on a[1], and T2 a write, hence Write after Read (WAR) dependence from T1 to T2. Or a Read after Write (RAW) dependency from T1 to T2 when T1 write on a[1], and T2 reads from a[1]. Such computations (T1, T2) will never be disjoint and cannot be parallelized without resolving the dependencies.

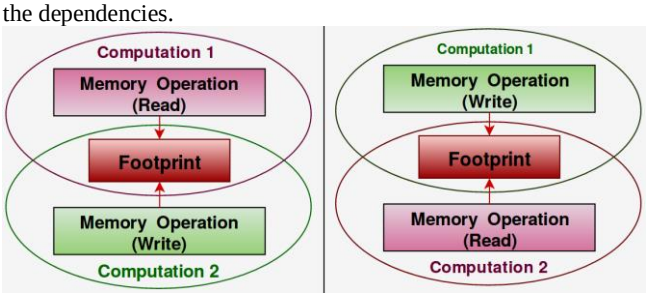

Fig. 3 Non-Disjoint Footprints and Disjoint Memory Operations.

3.1.3 Non-Disjoint Footprints and Operations

This is the case in which both footprints and memory operations will be non-disjoint. There are two sub-cases: -

1.Read – only Operation (4th Row): - For read-only operation, there is no dependence violation as it does not cause any violation for two computations. As shown in Figure 4, T1 and T2 can be parallelized as both are having read only status on a[1].

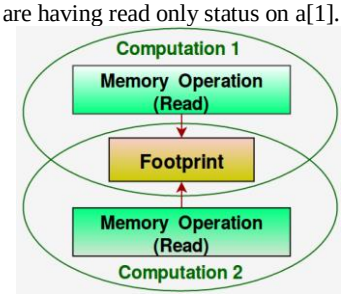

Fig. 4 Non-Disjoint Footprints with Read Operations.

2. Write - only Operation (5th Row): -Write operation on the same footprint by different computations will always cause output dependence (WAW).Therefore computations cannot be parallelized. As shown in Fig. 5, let T1 and T2 both write to a[1] then they cannot be parallelized as both are writing to the same memory address (footprint) a[1].

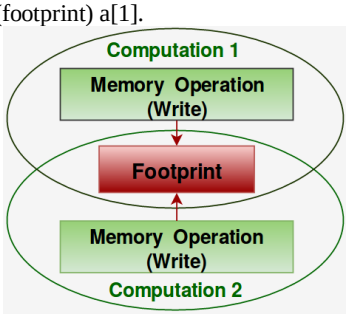

Fig. 5 Non-Disjoint Footprints with Write Operations.

As most of the execution time is spent in the loop [1], a lot of research has been carried out for automatically loop level parallelization [23]. We are also targeting the loop for automatic parallelization. In proposed technique with the help of Runtime Checks loops are directly parallelized at an early stage, and runtime technique is employed when a loop cannot parallelize directly. Following are the few benefits of Runtime Checks based technique:

1. Loops that could not parallelize statically at compile time because of compile time unknowns can be quickly parallelized using Runtime Checking ifthe loop has no dependencies.

2. It will determine the parallelism at an early stage (fine grain level) and help in reducing the overhead associated with runtime technique to parallelize a loop.

3. When Runtime Check identifies any dependence instead of directly executing the loop in sequence, if a runtime parallelization technique is employed at coarse grain level then it may give significant benefits.

As we discussed earlier, a program may have different types of irregularity in memory footprint/references, and based on the kind of irregularity different techniques can be employed for parallelization. We classified three cases as:

1. A loop with static irregular references with no dependencies.
2. A loop with static irregular references with dependencies.
3. A loop with dynamic irregular references with dependencies.

We are using Runtime Check based technique for case 1 and case 2, and Speculation for case 3. All cases and respective parallelization approach areshown in the Figure 6.

As shown in the Figure 6, explanation of the three cases asmentioned earlier are as follows:

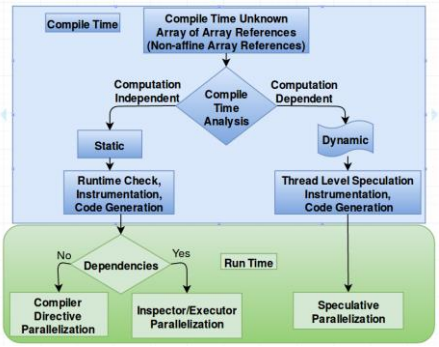

Fig. 6 Classification of Statically Unknown Irregular References and Parallelization Strategies.

1. In the first case, computations have static irregular disjoint footprints. Therefore, computations can be directly parallelized. Runtime checks are used to determine the disjointedness of footprints, and if there is no dependency parallelize the loop directly using Compiler Directives. Here, we are using Runtime Checks to enumerate the indices to determine whether footprints are disjointed or not. Classification Case 3.1.1.

2. The case when computations have static irregular non-disjoint footprints. For such a case, checks are used to determine disjointedness of the computation at an early stage. The Inspector-Executor technique is used if computations are not disjoint. In Inspector-Executor, all disjointed computations are grouped together using Inspector phase and ordered basedon the sequential semantics of the program. While at Executor phase all disjoint groups of computations will be executed in parallel. Classification Case 3.1.2 with Static Footprints.

3. When footprints are dynamically irregular and computation dependent, Runtime Check based technique will not work. Because even though using Checks indices (footprint) can be enumerated, but during execution they may change. Therefore, to parallelize such computation, Speculative runtime technique is needed. In this case, we introduced Speculation based technique similar to MINITLS [23]. We did some optimization to reduce memory overhead and to improve the performance. Classification Case3.1.2; 3.1.3.2 with Dynamic Footprints.

In next section, we will explain how loops with static and dynamic memory footprints can be parallelized using Checks, or by using Speculative runtime technique. In the rest of the article, for A[B[]] array A[] is called referring array, B[] is called referred array, and elements of the array B[] givesthe footprints for array A[] where read/write operations will be carried out during loop execution. The term footprint, reference, and index is used interchangeably, and all are same. We also consider one iteration of the loop as one computation. With the initial assumption that there will be no dependencies caused by any other memory operations (other then unknown array of array references). It means that read/write operations other than unknown array of array references are independent and do not cause any dependence.

4. Parallelization: Case Study

We group the parallelization process into two categories Static Irregular References Parallelization using Runtime Check, and Dynamic Irregular References Parallelization using Speculation. In the Static process we are using Runtime Checks which further divided into two types Compiler Directive Based Parallelization and Inspector-Executor Based Parallelization, and in the Dynamic process, we are using Speculation.

4.1 Case1: Static Memory References Parallelization using Runtime Check

When memory footprints/references are static, Runtime Check is inserted into the loop at compile time, and two parallel version of the loop is generated as shown in Example 4.1.1. A first parallel version using Compiler Directive and second parallel version using Inspector-Executor. During loop execution, first, Runtime Check function will be executed to determine the overlapping indices of the array (i.e. determines the disjointedness of footprints). The compiler directive based parallel code is executed when Runtime Check detects no overlapping of indices. But in the case when any overlapping is detected Inspector-Executor based parallel code will be executed. This solution works with a single read or writes indirection references inside a loop. It fails to parallelize a loop with dynamic references or when each iteration of a loop is having multiple reads and write references on same footprint (non-disjoint footprints with multiple read-write references).

4.1.1 General Concept

In this discussion, we concentrated on loops having single static indirect read/write reference to the array data structure i.e. loops with static irregular memory references. During the Runtime Checking process, a striped version of the original loop is generated and executed in parallel based on window size selected for parallel execution. Every memory location in user array guarded by a lock bit in profiling data structure and only one thread is allowed to obtain a lock form a footprint only once. Thread gets the lock for a footprint using atomic instruction such as Compare and Swap (CAS**)** [8] or Test and Set Lock (TSL) [2]**.** Work of Runtime Check is limited to detection of disjointedness of footprints. At any point in time whena non-disjoint footprint is detected Runtime Checking will immediately returns that there is a conflict. Checking process is stopped by sending a signal to all

active thread. Here, using Runtime Check, we are trying to determine whether any indirection of user array A[] is overlapping or not. If they overlap, Inspector-Executor based code will be executed as shown in Example 4.1.1.

```
read(A[], B[])
if (static references)
{
/*Runtime Check*/
dependence = runtime check(B[])
if(dependence == 0)
{
        #pragma omp parallel for
        for(i= 0; i< N; i+ +)
{
        A[ B[i] ] = .....
}
}
else
{
        Ins_Exe_forall(A[];B[];N);
}
}
```

Example 4.1.1: Static Irregular Reference Parallelization using Runtime Check

In Example 4.1.1, "runtime_check(B[])" is a function call which determines the "dependence" using parallel threads at runtime. If the Runtime Check return "no dependence i.e. (0)" then "#pragma omp parallel for" directive is used to parallelize the loop, otherwise "Ins_Exe_forall(A[], B[], N)" function is called for parallel execution.

4.1.2 Data Structure for Profiling

As shown in Figure 7, a new profiling data structure is needed to perform Runtime Checking to determine disjointedness of footprints (indices). In this technique, profiling data structure known as Bit Comp[N] is used to create a shadow of the referred array elements i.e. shadow of elements of array B[]. Bit_Comp[N] is a new Boolean array of bits of size N which initially initialized to 0. A hash function H = hash (B[*i*]) is used to map the referred array elements to the profiling data structure. Every Thread (T*i*) running in parallel obtain the lock on respective elements of array B[] by performing an automatic operation (TSL) on index H (H = hash (B[*i*])) of Bit_Comp[H]. At any point in time, if a Thread (T*i*) fails to perform TSL operation on respective Bit_Comp[H] it indicates that footprints are non-disjoint and Inspector-Executor loop will be invoked for parallel execution. On the other side when

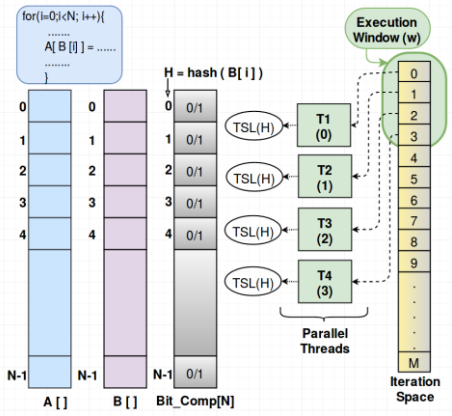

Fig. 7 Runtime Check Technique

all the iteration of the Runtime Check loop is over, and Thread successfully did their task there is no dependencies and loop will be parallelly executed using Compiler Directive based parallel code.

4.1.3 Scheduling/Load Balancing

For scheduling the iterations of the iteration space, we have used a window based scheduling technique [6] where window size determines the number of parallel threads. After completion of all the iteration within the window, the window will slide to next iterations untilall the iteration not over. Iterations within the window are allocated to the active threads based on a one-to-one mapping. For Runtime Check, profiling data structure will be initialized only once that is at the beginning of runtime checking process. While in Inspector-Executor based technique profiling data structure is re-initialized every time window slides.

4.1.4 Compiler Directive Based Parallelization

When Runtime Check returns no overlapping of indices that means all the footprints are disjoint. Then Compiler Directive based parallel code is executed. All scheduling and parallelization are done automatically. One example can be by using OpenMP[5] directives the only thingis that system should have OpenMP compiler, and no dependencies between loop iterations.

4.1.5 Inspector-Executor Based Parallelization

In this technique, an original loop is stripped down into two loops as Inspector loop and Executor loop where dependencies are determined parallelly at Inspector phase; if the indirection arrays do not change between invocations of the loop, then footprints will be static. Therefore schedule generated by the Inspector loop will be executed by the Executor.

As far as we know, there are no parallel Inspector-Executor approaches that are capable of exploiting parallelism between a consecutive write or consecutive read. We introduce new techniques where an original loop is stripped down in two loops: one parallel loop for an Inspector and another parallel loop for an Executor phase. We followed window based scheduling of the iterations to the threads, so further dependencies within the iterations of the window will be determined easily and do not cause waiting overhead for threads. In the Executor loop, parallel groups are cyclically distributed among the threads, and each thread goes on with the execution as dependencies are fulfilled. A parallel Inspector determined the dependencies within the window (set of iterations) prepare a parallel schedule and Executor executes different independent iteration groups parallelly. This process will continue until all the iteration of the loop is not over. Here one important note is that we are assuming that there is only one array of array references, and there are no other accesses to the referencing array A[] and referred array B[]. The methodology is evaluated for loops having static irregular read or write references with different memory access patterns and computational workloads. As for static case we are targeting the loop having a single array of array reference with a store statement, it makes our parallel implementation simpler to deal with the loops having static footprints.

During the parallel Inspector phase accesses to the user data structure A[] is recorded into a new shared data structure known as *DefUseMatrix*. *DefUseMatrix* is a 2-D array of structure which consists three fields: *ReadSet*, *WriteSet*, and *Group*. A *BitComp* array is used to get the exclusive ownership of footprints to record the access in *DefUseMatrix*. Synchronizationbetween multiple threads has been done by using CAS operation. Parallel Thread records the access into *DefUseMatrix*, and once all thread is done for a window size, their groups are determined. Each independent group is then scheduled to the different parallel thread for parallel execution at Executor phase. Once execution of current window is over the window will slide to next iteration set. Unlike Runtime

Check technique whenever window slides, *DefUseMatrix* and *BitComp* data structure will be reinitialized.

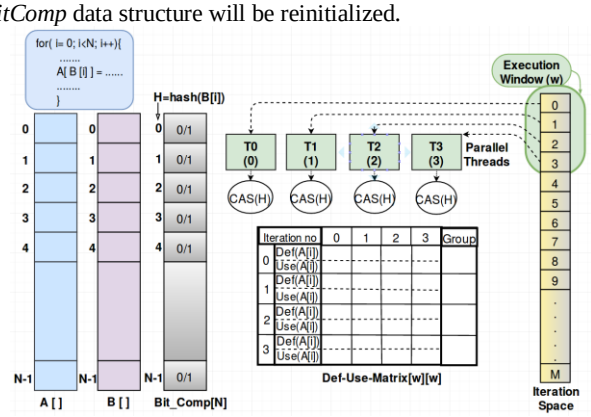

Fig. 8 Parallel Inspector Parallel Executor Technique

4.2 Case2: Dynamic Memory Refrences Parallelization using Speculation (BitTLS)

In this section, to deal with multiple dynamic non-disjoint/disjoint read and write references on nondisjoint footprint, we introduce a new speculative parallelization technique BitTLS which is based on MiniTLS**[Yiapanis et al(2013)Yiapanis, Rosas-Ham, Brown, and Luj_an].**We modified the shadow data structure used in MiniTLS to improve the performance. We are using lazy version management for data instead of an eagerversion as used in MiniTLS. The motivation behind using lazy version is thateven it serialize the commit, re-execution overhead will be decreased by using value forward mechanism which will provide correction instead of re-execution. Our technique not only gives better performance but also reduce the comparison overhead required to acquire a lock on a footprint of Profiling data structure (Prof DS[]) when already a writer thread is an owner for that footprint. An extra bit is introduced to reduce the comparison overhead and to improve
the performance. This additional bit is known as "WriteBit". Other than this, instead of direct roll back we introduced correction mechanism which furtherimproves the miss speculation overhead.

4.2.1 Data Structure for Profiling

A four thread configuration of BitTLS speculative parallelization technique is shown in Figure 9. There are three data structures for profiling, one is shared by all the threads and other two are private to each thread. This data structures are array of structures.

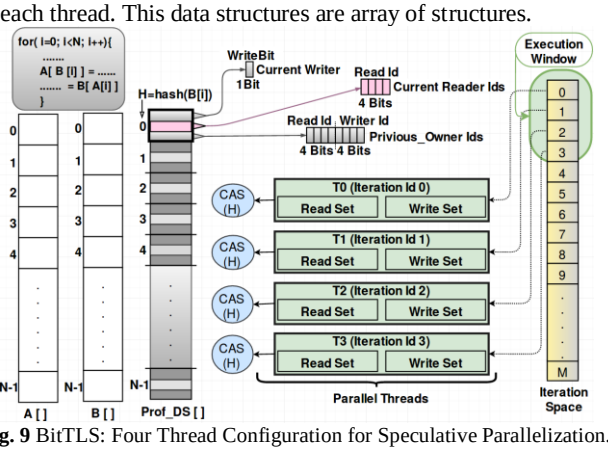

Fig. 9 BitTLS: Four Thread Configuration for Speculative Parallelization.

1. Shared Data Structure

A shared data structure known as Prof DS is shown in Figure 9. Prof DS[] is used to map all indirection indices of array A[] (i.e. Elements of array B[]).Prof DS is an array of integers which consists of three fields: *CurrentWriter, CurrentReaderId, PreviousOwner*. Bit location in bit sequences represents the Thread-Id and order of speculation of threads. A less speculative thread is represented by a less significant bit and a most speculative thread by a most significant bit. We used one-two bytes Integer for four thread configuration. Where first eight bits are used for *PreviousOwner (WriterId* and *ReaderId*four bit each) and next four bits are used for *CurrentReader*, next one bit for *CurrentWriter*, and remaining bits left unused. Like this, for four thread configuration, we are using thirteen bit and allowing multiple readers simultaneously for a read operation. On the other hand, in MiniTLS to allow multiple readers simultaneously sixteen bits are required. As well in MiniTLS, if only twelve bit is used then the parallel read will be not supported.

2.Private Data Structure

Apart from a shared data structure in BitTLS, every thread will have their own private data structure, implemented using the array of structures to profile the accesses and updates to user data structure. Every thread will have a *ReadSet* and a *WriteSet*. Read set is used to store last read value in shared data structure.Likewise, *WriteSet* is used to store the value which current thread want to update in user data structure. A hash function H(H = hash (B[i])) is used to store value at respective location in *ReadSet* and*WriteSet*. Since we are using lazy version management for data and conflict detection thread-local data structure will contain the most updated valuewhich will be passed to a more speculative thread using value-speculation.

4.2.2 Operations

In any speculative parallelization technique order of speculation is very crucial as facilitates the commit, correction, and rollbacking decision in a view to maintain program sequential semantics. As we are following window based speculative scheduling of the loop iteration thread that executes the first iteration of the window will be less speculative and thread that executes the last iteration of the window will be most speculative. As well in our technique, lazy version management is used, so we are not bothered about Write-after-Write (WAW) and Write-After-Read (WAR) conflicts and only need to deal with Read-After-Write (RAW) conflicts.

1. Speculative Read

Whenever a thread wants to perform a read operation on A[B[*i*]], It have to obtain the ownership of respective footprint/index (H = hash(B[*i*]]) in Prof DS[] data structure by performing an atomic operation (i.e. CAS). Steps are follow to get the ownership: first, check that *WriterBit* is set or not. If set, it means that location is write protected and another thread is a write owner of that footprint hence no thread is allowed to get ownership until ownership is released. When write bit is*notset*, and current thread wants to perform a read operation, then it has to set *ReaderId*. After this process thread will check whether any less speculative thread previously performed a Write operation on that footprint. For this thread need to check *PreviousOwner-WriteId* field. In case if any less speculative thread performed Write operation on that footprint then current thread will read the value from nearest less speculative thread local *WriteSet* data structure. Like this Value-Forwarding is achieved in our technique. Remember we are using lazy version management technique to manage the shared data so updates will be stored in thread local data structure, and when all iteration of current threads committed shared data structure is modified using thread local data structure. Before releasing ownership of footprint thread have to set corresponding bit in *ReaderId* of *PreviousOwner* field. When the same thread wants to carry out a read operation to the same location multiple time, *ReaderId* of that thread will be unchanged. At the same time when one thread is currently read owner of that footprint another thread also allow to read by setting there respective bit in the *ReaderId*.

2. Speculative Write

For a writer thread, the flow of checking is as follows: first, identify that Writer Bit is set or not. If set, it means another thread is a write owner of that location and to obtain ownership thread have to wait by using spin locking mechanism. If write bit is *notset*, then that thread have to check whether any other thread is read owner of that location or not. If *writeBit* is *notset.* and no read owner of that footprint, current thread has to set *WriteBit*. Then perform a write operation, modify thread-local data structure, and before releasing ownership of footprint thread have to set corresponding bit in respective WriterId of *PreviousOwner* field. When the same thread wants to carry out a write operation multiple time to the same location, *WriterId* of that thread will be unchanged in the *PreviousOwner* field.

4.2.3 Scheduling/Load Balancing

There are two primary scheduling policies in the literature as static and dynamic scheduling to schedule the iterations of the loop to speculative threads. In a static scheduling, a fixed number of iterations are scheduled at beginning, while in dynamic scheduling, iterations are scheduled at runtime in fixed chunk size. We have used sliding window based scheduling[6] to schedule the iterations to the speculative threads, where window size will decide the number of active thread for parallel execution. There is one to one mapping between iteration to threads where thread id determines the speculation order of the threads. During the execution, once all the iteration withina window is completed the window will slide. Whenever window slides all shared, and private data structure of every thread will be re-initialized, once the modification is updated in the user data structure.

4.2.4 Dependence Detection and Commit

In BitTLS, a lazy conflict detection mechanism is implemented, in which a thread have to wait until the commit of a less speculative thread. Furthermore, the MiniTLS system employs a conflict correction mechanism known as value forwarding to overcome the waste of execution time because of re-execution.It means a thread that identifies a conflict must delays immediately any speculative execution, and starts a correction mechanism by sending a signal to all more speculative threads. In which value from the less speculative thread will be forwarded to the more speculative threads for the correction. To determine thread that operated on that location for reading and write operation *PreviousOwner* bits of shared data structure is used. To record the commit operations of the thread unused 4 bits of shared data structure can be used. When a thread issue a read operation during speculative execution and detects that a less speculative thread performed a write operation on that location,then thread will read the value from previously less speculative thread local *WriteSet*if thread status is uncommitted. While when thread status is committed, current thread has to read from user data structure. With the help of value-forwarding, RAW conflicts will be resolved. In the case when a more speculative thread performed a write on that location thread will read from user data structure by following this process WAR conflict will be resolved.

5. Conclusion and Future Work

In this article, we have focused on how to use different parallelization technique for various type of irregularity in an array of array references. We used Runtime Check based analysis that is performed at runtime to determine the loops having a statically unknown static array of array references without any dependencies. We mainly focused on using Runtime Checks to determine the indices overlapping (footprint disjointedness) then based on the result of Check different execution path will be followed. When check return no dependencies compiler directive based parallelization has been performed and in the case of dependencies Inspector-Executor based parallelization. Further, we proposed a speculative technique BitTLS to parallelize a loop having a dynamic irregularity in the array of array references. We focused on reducing the size of profiling data structures and on reducing the number of computations required for providing the synchronization between parallel executions. In all this three techniques we have performed bit-level operations for thread synchronization and window based scheduling to schedule loop iterations to the threads.

Our future work will be divided into three primary objectives. First objective focus on determining the effectiveness of our proposed methodology on the set of C benchmark. Then in the second objective, we plan to do a comprehensive investigation of the speedup achieved. Finally, we will extend our work to develop new parallelization techniques in the presence of pointer-intensive programs that make use of dynamic data structures such as Linked List, Trees and Graphs.

Acknowledgements

Authors acknowledge Department of Science & Technology, Government of India for financial support (Reference no. SR/WOS-A/ET-27/2014(G)) under Women Scientist Scheme-A (WOS-A) to carry out this research work.

References

[1] Aho AV, Sethi R, Ullman JD (1986) Compilers, Principles, Techniques. Addison wesley Boston

[2] Anderson TE (1990) The performance of spin lock alternatives for shared money multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1(1):6-16

[3] ARM-Cortex (2011) a9 processor. URL: http://wwwarmcom/products/processors/cortex-a/cortex-a9php[Accessed 15 May 2017]

[4] Contributors W (2017) Memory footprint. https://en.wikipedia.org/w/index.php?title=Memory_footprint&oldid=769055905, online; accessed 19-August-2017

[5]Dagum L, Menon R (1998) Openmp: an industry standard apifor shared-memory programming. IEEE computational science and engineering 5(1):46-55

[6] Dang F, Yu H, Rauchwerger L (2001) Ther-lrpd test: Speculative parallelization of partially parallel loops. In: Parallel and Distributed Processing Symposium, Proceedings International, IPDPS 2002, Abstracts and CD-ROM, IEEE, pp 10-pp

[7] Felicie AL (2012) Reference (Computer Science). Salve Regina University

[8] Harris TL, Fraser K, Pratt IA (2002) A practical multi-word compare-and-swap operation. In: International Symposium on Distributed Computing, Springer, pp 265-279

[9] Huang D, Steffan JG (2011) Programmer-assisted automatic parallelization. In: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp., pp 84-98

[10] M D Allen SS, Sohi GS (2010) Serialization sets: a dynamic dependence-based parallel execution model. In: PPoPP '09, New York, NY, USA, ACM,pp 85-96

[11] M Kulkarni BWGRKB K Pingali, Chew LP (2007) Optimistic parallelism requires abstractions. In: PLDI '07, pp 211-222

[12] M Mendez-Lojo DPXSMAHMKMB D Nguyen, Pingali K (2010) Structure-driven optimization for amorphous data-parallel programs. In: PPoPP '10, New York, NY, USA, ACM

[13] Minh CC, Chung J, Kozyrakis C, Olukotun K (2008) Stamp: Stanford transactional applications for multi-processing. In: IISWC, IEEE Computer Society, pp 35-46

[14] Mock M (2004) Why programmer-specified aliasing is a bad idea. In: Latin American Conference on Compiler Science

[15] Nicolau A (1989) Run-time disambiguation: coping with statically unpredictable dependencies. IEEE Transactions on Computers 38(5):663-678

[16] R L Bocchino VSADDSVASHRKJOPSHS Jr, Vakilian M (2009) A type and effect system for deterministic parallel java. In: OOPSLA'09, NewYork, NY, USA, 2009. ACM, pp 97-116
[17] Rauchwerger L (1998) Run-time parallelization: Its time has come. Parallel Computing 24(3):527-556
[18] Reinders J (2007) Intel Threading Building Blocks
[19] Rinard MC, Lam MS (2010) The design, implementation, and evaluation of jade. In: ACM Trans. Program. Lang. Syst., 20(3), pp 483-545
[20] Rus S, Pennings M, Rauchwerger L (2007) Sensitivity analysis for automatic parallelization on multi-cores. In: Proceedings of the 21st annual international conference on Supercomputing, ACM, pp 263{273
[21] S S Mukherjee MDHJRLAR S D Sharma, Saltz J (1995) Efficient support for irregular applications on distributed-memory machines. In: PPOPP '95, ACM, New York, NY, USA, pp 68-79
[22] W Reid WK, Craik A (2008) Reasoning about inherent parallelism in modern object-oriented languages. In: ACSC '08, Darlinghurst, Australia, Australian Computer Society, Inc., pp 27-36
[23] Yiapanis P, Rosas-Ham D,Brown G, Lujan M (2013) Optimizing software runtime systems for speculative parallelization. ACM Transactions on Architecture and Code Optimization (TACO) 9(4):39