Study the impact of using Lock-free buffer to communicate the DOACROSS loop iterations

Esraa H.A. Alwan
Computer science
College of science for women -Babylon University
wsci.israa.hadi@uobabylon.edu.iq
Isr.phd@gmail.com

Abstract

Communication dependency overhead becomes the biggest obstacle that facing the parallelizing loops containing loop-carried dependencies such us DOACROSS loop. Although of a substantial researches had been devoted to this field, the problem still far from solved.

This work introduces a FastForward circular lock-free queue algorithm to communicate the dependency between DOACROSS loop iterations. Instead of giving each iteration of DOACROSS loop to thread as in the original methods, group of iterations will be given to each thread. So to ensure correct results, the dependence between threads must be respected and for parallelism to be effective, the overhead on core-to-core communication must be as low as possible.

Experimental results are implemented on Intel Core i7 processor that has 4GB RAM running SUSE operating system show performance improvements of the proposed DOACROSS approach. An evaluation of this technique on four programs with a range of dependence patterns lead to ≈ 0.9 speed up.

Keywords: DOACROSS, Lock-free buffer, Parallelizing techniques, Multicore

الخلاصة

الكلفة العالية لنقل المعلومات بين المعالجات في الحاسبات المتعددة المعالجات هي واحدة من اكبر المشاكل التي تواجه تنفيذ الدورات ذات الاعتمادية بصورة متوازية . على الرغم من الجهود الكبيره التي بذلت من في هذا المجال الا ان المشكله مازالت بعيده عن الحل. هذا البحث يدرس تأثير استخدام خوارزمية الطابور الدائري بدون القفل لنقل المعلومات بيت المعالجات. لضمان صحة النتائج فان الاعتمادية بين التكرارات الخاصه بالزوارق يجب ان تراعى بالاضافة الى ان كلفة نقل المعلومات يجب ان تكون قليله قدر الامكان. النتائج التي تم الحصول عليها من التجارب تم تنفيذها على حاسبة من نوع Core i7 تنفذ نظام تشغيل سوري ان التقنيه المقترحة قامت بتحسين الأداء وأدت الى تحسين وقت التقيد بنسبه 0.9

الكلمات المفتاحية : الخزان المؤقت بدون مفتاح, تقنيات النفيذ المتوازي, الحاسبات المتعددة المعالجات

I. Introduction

Raising the sequential application performance required many improvements in both commodity hardware systems and compiler optimization techniques. A range of microarchitectural techniques that have been used to enhance the performance of single thread applications in a highly effective way. These techniques include the superscalar issue, out-of-order execution, on-chip caching, and deep pipelines supported by sophisticated branch predictors (Spracklen and Abraham,2005; Rangan,2008).

A new strategy has been introduced by processor designers which involved steadily increasing in the number of transistors, through which many cores are placed in one chip to replicate the performance of a single faster processor.

Multicore systems have become a dominant feature in computer architecture. Chips with 4, 8, and 16 cores are available now and higher core counts are promised. Unfortunately single-threaded legacy application does not achieve better performance

when it is executed on multicore system. However the level of gain becomes higher with software techniques which are parallelizing the sequential application to get better performance (Matthew Bridges,2008; Adve,2010).

Parallelizing single threaded application by converting it to multi-threaded application that can be used most of the cores in multi-core architecture become the most important current research topics. Traditional parallelizing techniques such as DOALL and DOACROSS work through distribution the loop iterations among many cores (Matthew Bridges,2008).While the DOALL techniques improve the program performance when applied in the numerical field, the overhead of communicating the dependency between threads in DOACROSS technique is so large and can be negated any advantage due to the lack of hardware support for inter-core communication.

In this paper, we study the impact of using a Fastforward lock-free buffer method on DOACROSS technique. Fastforward circular lock free buffer has been used to communicate the dependency between thread without any lock where the producer and consumer can reach the queue concurrently. Unlike the a lock-based approach, such as pthread mutex lock which can negate any benefit from the parallelizing due to their overhead. The rest of this paper is structured as follows: the next section (II) introduces the background for DOACROSS technique, then section III explain how can be using the Fastforword lock free buffer to parallelize the DOACROSS technique. Section IV shows the implementation of the proposed method. V presents some experimental results from the application of the manual transformation. Finally in section VI, we survey related work and conclude .

II. Background

DOACROSS is the most popular Cyclic Multi-Threading (CMT) transformation. This technique tries to execute the loop body in parallel even in the exist the cross dependency between it is iterations. This technique works similarly to DOALL, see figure 1, by giving each iteration to a thread and these threads are executed on multi-core in round-robin fashion. In contrast with the DOALL technique, however, there are data and control dependencies crossing loop iteration boundaries.

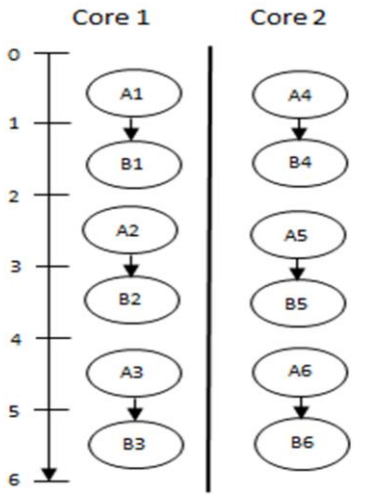
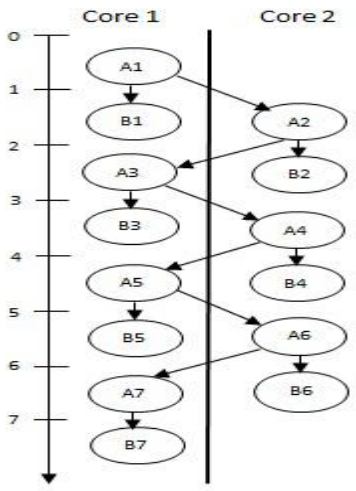


Figure 1: DOALL technique. Adapted from (Raman,2009)

Therefore, to get correct results the dependency should be respected and the synchronization should be in the right order so as to ensure that the following iteration receives the correct value (Chen,2010; Rajamony,1997). With the increased number of cores integrated on the same die, the communication latencies between them become more significant, with their values ranging from a few tens of cycles to even a few hundreds. As a result, this leads to reduced performance of CMT, for unlike with Independent Multi-Thread (IMT) the increased number of threads in the system does not always lead to a linear increase in performance. Figure 2 shows an example of this technique. DOACROSS schedules each loop iteration on an alternate thread and communicates the dependency from thread to thread in a cyclic fashion due to the pointer chasing in statement A. Set of odd iteration will be given to the first thread while the rest will be given to the second one (Unnikrishnan et al.,2012;Zhong,2001). The total time to execute the DOACROSS parallelized loop equals:

$$L_{par} = \frac{n}{m} * (L_i + SC) \qquad (1)$$

where *n* represents the number of loop iterations, *m* the number of available threads, *iter*1, *iter*2, *iter*3, *iter*4 represent loop iterations, *Li* the execution time for one loop iteration



A : while(ptr = ptr->next)
B : ptr->data += 1;

Figure 2: DOACROSS techniques and code example .Adapted from (Raman,2009)

and *SC* the stall cycle. Figure 3 shows the synchronization and associated stalls that happen during the execution of the DOACROSS technique. C represents the consumer point and P the producer point, whilst t2 is the execution time between a consumer point and producer point in the same thread. Depending on this figure the *SC* between *iter*1(*i*) and *iter*4(*i*+*m*) is equal to

$(t2_1 + CL1) + (t2_2 + CL2) + (t2_3 + CL3)$
Or

$$m * (t2 + CL)$$

Where, CL is the communication latency between two threads. Finally, the *speedup* that will be gained from this technique is:

$$Speedup = \frac{n * L_i}{L\ par} \qquad (2)$$

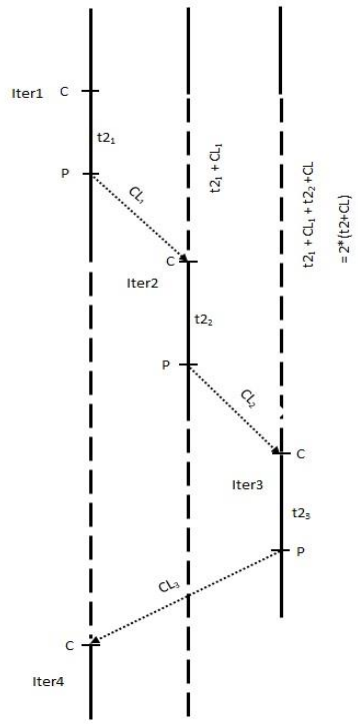


Figure 3: This figure shows the synchronization and associated stalls in DOACROSS. Adapted from (Raman,2009)

III. Proposed method

In order to improve the communication latency, fast synchronization and communication mechanisms are necessary. Using lock-based approach, such as a pthread mutex lock to communicate the dependency is impracticable and can negate any benefit from using DOACROSS technique. Moreover, many of the drawbacks can be associated with lock-based approaches such as deadlocks and livelocks. Unlike lock-based approach, the synchronization with lock-free one is achieved through high-level tools rather than mutex locks(John et al.,2008).

In this work , FastForward circular lock-free queue algorithm has been introduced to communicate the dependency between DOACROSS iteration. Instead of giving each iteration of DOACROSS technique to thread as in the original methods, group of iterations will be given to each thread. So to ensure correct results, the dependence between threads must be respected and for parallelism to be effective, the overhead on core-to-core communication must be as low as possible. The FastForward circular lock-free queue algorithm uses the entry buffer itself to determine the state of the circular queue where both control variables (i.e., head and tail) are thread local. Using this algorithm the overhead of accessing the control variables is reduced (Chen,2010). As a result, the producer and consumer can access the queue concurrently, via the enqueue and dequeue operations, which makes it possible for them to operate independently as long as there is at least one data element in the queue. Two communication primitives producer and consumer are inserted in the source and the destination of the dependency. These

communication pairs are inserted statically where each producer feeds one consumer and each consumer is fed by one producer. Figure 4 illustrates the enqueuer and dequeuer functions

```
1 enqueue_nonblock ( data )
2 {
3 if ( NULL != buffer [ head ]) {
4 return 1;
5 }
6 buffer [head] = data ;
7 head = NEXT ( head );
8 return 0;
9 }
```

Enqueue Function

```
1 dequeue_nonblock ( data )
2 {
3 data = buffer [ tail ];
4 if ( NULL == data ) {
5 return 1;
6 }
7 if ( NULL == data )&&( flag ==100) {
8 return 2;
9 }
10 buffer [ tail ] = NULL ;
11 tail = NEXT ( tail );
12 return 0;
13 }
```

Dequeue Function

Figure 4: Enqueue and dequeue function

In order to determine the source and the destination of dependencies between loop iterations, the loop body needs to inspect. These arguments denote the data that will go in the communication buffers. The destination of a dependency appears in the loop iterations and hence where the data must be retrieved in order for iterations to work correctly.

IV. Experimental Result

This section discusses the results that have been obtained from the manual implementation of the proposed method. The performance of DOACROSS using fastforward lock free buffer has been studied. Experiments were implemented on an Intel Core i7 that has 4GB of RAM running SUSE operating system(see table 1).

Table 1 : Platform details

Processor	**Intel® Core™ i7 CPU**
Processor speed	**2.93GHz**
Processor Configuration	**1 CPU, 4 Cores, 2 Threads per Core**
L1d Cache size	**32 k**
L1i Cache size	**32 k**
L12 Cache size	**256 k**
L13 Cache size	**8192 k**
RAM	**4 GB**
Operating System	**SUSE**
Compiler	**GCC 4.8**

All programs were manually parallelized and compiled with GCC 4.8 compiler. Several programs have been used as case studies. Experiments were carried using a set composed of four programs. These programs/loops were selected because they have different aspects of the dependency. Some loops are well balanced and others are imbalanced, some loops have high trip count / loop-visits others not, etc. Be able to produce speedups (or at least not hurt performance) under these situations is important when one consider using these algorithms. Next we look at the sample programs in more detail and look at the results of the transformation process. For all these below figures the columns represent the execution time in ms (millisecond) where the row represents the number of threads.

Loop.c An artificial program for the purpose of demonstrating the effectiveness of DOACROSS technique. First, we execute the loop body transformed by DOACROSS. The loop iterations distribute among cores. At the first time we distribute the loop iterations between two cores(30 iterations each). Then the loop iterations distributed among four cores (15 iterations each). Figure 5 shows the enhancement in the execution time with 2 and 4 threads. No more thread have been presented (no more splitting) that because the performance start to degrade. In this program the location of dependency is at the beginning of the loop body with heavy computation inside loop. For this reason the enhancement in the execution time has been achieved.

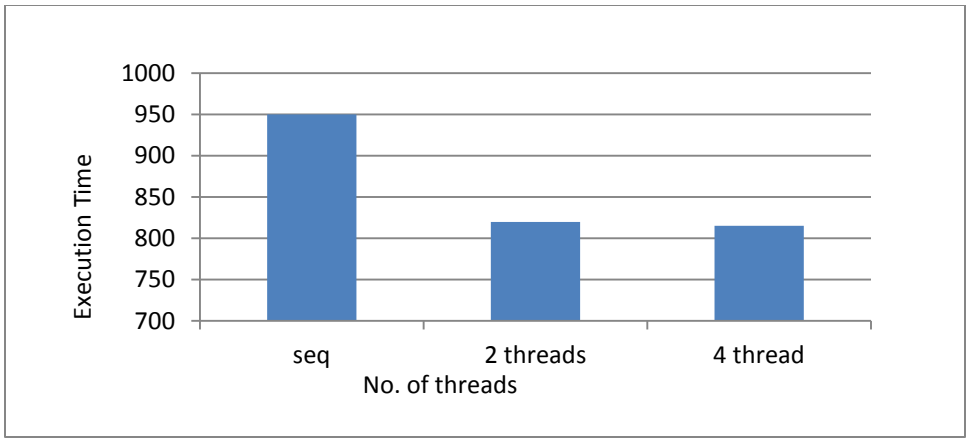


Figure 5 : loop.c program

Linkedlist.c This program is another artificial program. The common feature is the traversal of a linked list of linked lists (in contrast to the use of arrays as in the other examples). This allows us to demonstrate the cost of adding a buffer to the program. Two parameters affect the workload, namely the length of the first level list and the length of the second level list. The loop iterations distribute among cores. At the first time we splitting the loop iterations between two cores (10 iterations each). Then the loop iterations have been splitted among four cores (5 iterations each). Figure 6 shows the enhancement in the execution time with 2 and 4 threads. Same as before with the increasing number of cores (more splitting) the performance starts to degrade. Similarly the location of dependency is at the beginning of the loop body with heavy computation inside it.

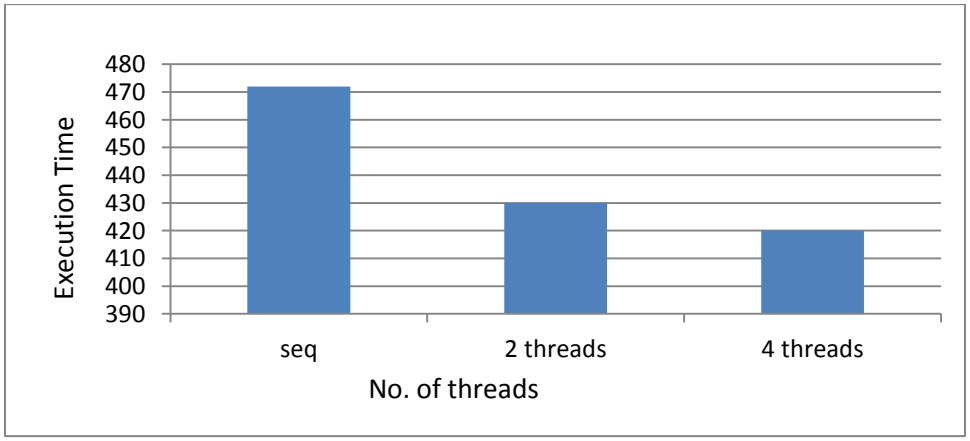


Figure 6: Linkedlist.c program

Code-1.c This program (Tao,1997). presents the boundary value problem with the shooting method. The Runge-Kutta and secant methods are used. Some change has been made to this program to make it appropriate for this work. At the first time the loop iterations have been splitted between two cores (50 iterations each). Then more splitting has been done among four cores (25 iterations each) as illustrate in the figure 7. For this program the location of dependency in the middle of the loop body .

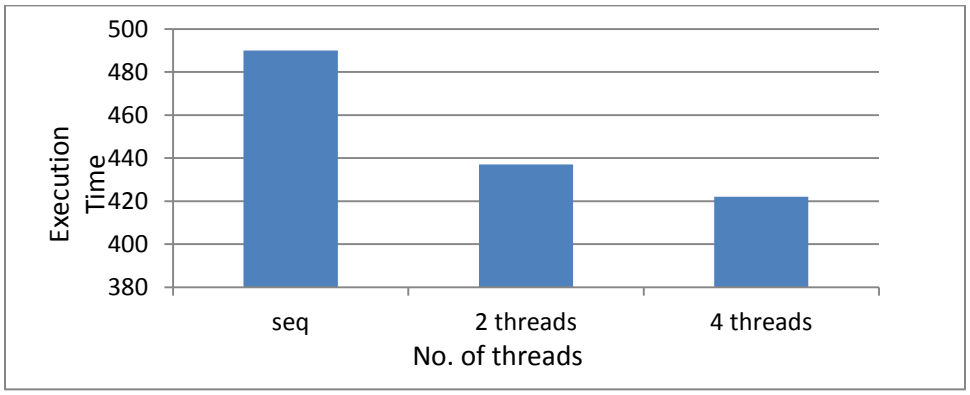


Figure 7: Code-1.c program

Sor.c This program solves Laplace's equation on the unit square in finite difference form using SOR iteration (Its illustrate how the location of dependency (at the end of the loop body in this program) and trivial computation inside the loop body can negate any benefit from using DOACROSS technique. The execution time of the sequential program with 5000 loop iterations is 3.358 ms while for the parallel one is 4.100 ms.

Table 2 shows the details about the programs that were parallelized. These columns identifies the programs name, the number of iterations for the whole program , the execution time for sequential program and the last two columns represent the execution time for the parallel program with different number of threads(two and four).

Table 2: The experiment result

Program name	Iteration	ExeSeq	Exe2threads	Exe4threads
Loop.c	60	950	820	815
Linkedlist.c	20	472	430	420
Code-1.c	100	490	437	422
Sor.c	3.358	4.100		

It is also shows the improvement in the execution time for the parallel program with two and four threads for the first and second compared with the sequential one. According to the DOACROSS behavior, we do not expect a significant reduction in the execution time for the parallel one. The iteration in the first thread should be finished before the second

thread start to execute. The overlap between threads is very small and it depends on how long the execution time for the rest of loop body it will take.
Another issue is the amount of dependency between loop iterations. All the above programs have very limited dependency between loop iterations (just one value). The execution time will be increased with more than one value communicate between loop iterations (due to the multiple call to the enqueue and dequeue functions). And the situation will get worse when the dependency locates in the middle or at the end of loop body with no heavy computation inside the loop.

V. Related work

The first person introduced the DOACROSS technique in 1986 was Ron Cytron to parallelize cross-iteration dependent loops(Chen,2010). Then considerable amount of researches have been done to improve DOACROSS performance.

Ding and Den present an effective algorithm to determine the redundant synchronization in the multidimensional DOACROSS loop. Nonuniformity of boundary iteration in multidimensional iteration space has been addressed in this algorithm. Moreover, they present an effective condition to identify redundant synchronization that do not have nonuniformity problem(Chen,1999).

Priya et al. automatically parallelizing the DOACROSS loop for manycore-SMP system. The proposed method bounds the number of synchronization variables with worker thread (processor core) by using compiler and runtime optimization that is called dependency folding. Moreover, efficient cost analysis has been presented to determine the worth of DOACROSS parallelization (Unnikrishnan et al.,2012).

Rajamony and Cox introduce two algorithms to enhance the performance of DOACROSS technique. These algorithms added lesser synchronization to minimize the amount of synchronization that have been added when parallelizing loop carried dependency. The first one uses an interval graph representation of dependency while the second one uses the integer programming to determine the optimal solution (Ramam,2009). Zhong (2001) present a time stamp algorithm to parallelize the DOACROSS loop at runtime. This algorithm exploits the parallelism at fine grained memory reference level through using the INSPECTOR/EXECUTOR scheme. Moreover parallelism among consecutive reads of the same memory element has been exploit which improves the previous algorithm of the same generality (Zhong ,2001).

VI. Conclusion

This work introduces a Fastforward circular lock-free queue algorithm to communicate the dependency between DOACROSS iterations. Instead of giving each iteration of loop to thread as in the original DOACROSS method , group of iterations will be given to each thread. Two communication primitives producer and consumer are inserted in the source and the destination of the dependency. These communication pairs are inserted statically where each producer feeds one consumer and each consumer is fed by one producer.

An evaluation of this technique on four program codes with a range of dependence patterns leads to performance improvements gains on a core-i7 870 machine with 4-core / 8-threads. The results are obtained from an manual implementation that shows the

proposed method can give a factor of up to ≈ 0.9 speed up compared with the original sequential code. More researches need to be done with different program sizes and platforms, explorations for the combination with other techniques.

References

Adve, S. V., Boehm, H.-J. Memory models: a case for rethinking parallel languages and hardware. Commun. ACM, 53(8),pages:90–101.2010.

Chen, W. R., Yang, W., and Hsu, W. C. (2010). A lock-free cache-friendly software queue buffer for decoupled software pipelining. In Computer Symposium (ICS), 2010 International, pages 997–1006. IEEE.

Chen, D. K.,and Yew P.C., *Redundant Synchronization Elimination for DOACROSS Loops*, IEEE Transactions on Parallel and Distributed Systems, v.10 (5). Pages : 477-481, 1999.

John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for efficient pipeline parallelism: a cache-optimized concurrent lockfree queue. In Siddhartha Chatterjee and Michael L. Scott, editors, Principles and Practice of Parallel Programming, pages 43–52. ACM, 2008.

Spracklen L. and Abraham S. G. Chip Multithreading: Opportunities and Challenges, HPCA ,2005.

Matthew Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Code.* PhD thesis, Department of Computer Science, Princeton University, November 2008. Retrieved 20130215 from ftp://ftp.cs.princeton.edu/techreports/2008/ 835.pdf.

Rangan R., Vachharajani, N., Ottoni, G., and August, D. I. (2008). Performance Scalability of Decoupled Software Pipelining. ACM Transactions on Architecture and Code Optimization (TACO), 5(2).

Unnikrishnan P., J. Shirako, K. Barton, S. Chatterjee, R. Silvera, andV. Sarkar. A practical approach to doacross parallelization. In Euro-Par
'12, 2012.

Raman, E. (2009). Parallelization Techniques with Improved Dependence Handling. PhD thesis,Princeton University. Dept. of Computer Science. http://liberty.princeton.edu/Publications/phdthesis_eraman.pdf, retrieved <2013.02.27>.

Rajamony R.; Cox . L. A., Optimally synchronizing DOACROSS loops on shared memory multiprocessors. Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques, pages:214 – 224, 1997.

Vachharajani, N. A. (2008). *Intelleigent Speculation for Pipelined Multithreading*. PhD thesis, Departement of Computer Science, Princeton University. http://liberty.princeton.edu/ Publications/phdthesis_nvachhar.pdf, retrieved <2012.11.08>.

Zhong Xu C. and Chaudhary V. ,Time Stamp Algorithms for Runtime Parallelization of DOACROSS Loops with Dynamic Dependences . IEEE Transactions on Parallel and Distributed Systems., V 12(5),pages: 433 - 450, 2001.

Tao Pang. An Introduction to Computational Physics. Cambridge University Press, 1997. Retrieved 20170213 from http://www.physics. unlv.edu/_pang/cp c.html.