Time-Stamping Algorithms For Parallelization of Loops at Run-Time *

Chengzhong Xu and Vipin Chaudhary
Department of Electrical and Computer Engineering
Wayne State University, MI 48202
Email: {czxu, vchaud}@ece.eng.wayne.edu

Abstract

In this paper, we present two new run-time algorithms for the parallelization of loops that have indirect access patterns. The algorithms can handle any type of loop-carried dependencies. They follow the INSPECTOR/EXECUTOR *scheme and improve upon previous algorithms with the same generality by allowing concurrent reads of the same location and by increasing the overlap of dependent iterations. The algorithms are based on time-stamping rules and implemented using multithreading tools. The experimental results on an SMP server with four processors show that our schemes are efficient and outperform their competitors consistently in all test cases. The difference between the two proposed algorithms is that one allows partially concurrent reads without causing extra overhead in its inspector, while the other allows fully concurrent reads at a slight overhead in the dependence analysis. The algorithm allowing fully concurrent reads obtains up to an* 80% *improvement over its competitor.*

1 Introduction

Automatic parallelization is a key enabling technique for parallel computing. Of particular importance in this area is *loop parallelization* that exploits inherent parallelism inherent in a loop, partitions the loop code and data across processors and orchestrates the parallel execution of the loop [1]. Current parallelizing compilers demonstrate their effectiveness for loops that have no cross-iteration dependences or have only uniform dependences. They have limitations in parallelization of loops that have complex or statically insufficiently defined access patterns. As an example, Figure 1 presents a loop exhibiting indirect access patterns for elements of a data array X. In the case that its index arrays u and v are functions of the input data, the loop is unable to be parallelized by current parallelizing compilers because its available parallelism cannot be determined statically. For the parallelization of loops with indirect access patterns, compile-time analysis must be complemented by run-time techniques capable of automatically exploiting parallelism at run-time.

for ($i = 0; i < N; i = i + 1$)
  $X[u[i]]$ = F($X[v[i]], \ldots$)

Figure 1: A general form of loops with indirect access patterns

Loops with indirect access patterns appear frequently in scientific and engineering applications. A recent empirical study by Shen, Li and Yew [12] on more than 1000 scientific and engineering routines (100,000 code lines) showed that 47.44% of the one-dimensional array references and 44.91% of the two-dimensional array references were nonlinear functions with respect to the loop indices, and that 15.22% of nonlinear subscripts in one-dimensional arrays and 2.76% nonlinear subscripts in two-dimensional arrays are due to the presence of indirect access arrays. Examples include SPICE for circuit simulation, CHARMM and DISCOVER for molecular dynamics simulation of organic systems, and FIDAP for modeling complex fluid flows [2].

In the past, many run-time parallelization algorithms have been developed for different types of loops on both shared-memory and distributed-memory machines [4, 10, 6]. Most of the algorithms follow a so-called INSPECTOR/EXECUTOR approach. With this approach, a loop under consideration is transformed at compile-time into an *inspector* routine and an *executor* routine. At run-time, the inspector examines loop-carried dependencies between the statements in the loop body; the executor performs the actual loop operations in parallel based on the dependence information exploited by the inspector. The key to success with this approach is how to shorten the time spent on dependence analyses without losing valuable parallelism and how to reduce the synchronization overhead in the executor. An alternative to the INSPECTOR/EXECUTOR approach is

*This work was sponsored in part by NSF MIP-9309489, US Army Contract DAEA-32-93-D-004, and Ford Motor Company Grant #0000952185

a speculative execution scheme that was recently proposed by Rauchwerger, Amato and Padua [9]. In the speculative execution scheme, the target loop is first handled as a doall regardless of its inherent parallelism degree. If a subsequent test at run-time finds that the loop was not fully parallel, the whole computation is then rolled back and executed sequentially. Although the speculative execution yields good results when the loop is in fact executable as a doall, it fails in most applications that have partially parallel loops.

Following the Inspector/Executor approach, this paper proposes two new parallel algorithms for the run-time parallelization of loops that have indirect access patterns. The algorithms are able to handle any type of loop-carried dependencies. They improve upon previous algorithms with the same generality by allowing concurrent reads of the same array element in different iterations, and by increasing the overlap of dependent iterations. The algorithms are based on time-stamping rules and implemented using multithreading tools. The experimental results on a SUN SPARCserver 630MP show that both algorithms are efficient and outperform their competitors in all test cases. The differences between the two new algorithms are that one algorithm allows partial concurrent reads without causing extra overhead in its inspector, while the other allows fully concurrent reads at a slightly higher cost in the dependence analyses. Although our algorithms are evaluated in centralized shared memory machines, they are applicable to distributed-shared-memory systems as well.

The rest of the paper is organized as follows. Section 2 provides an overview of the INSPECTOR/EXECUTOR technique and a brief review of previous work. Section 3 presents an improved algorithm based on a recent work by Chen, Torrellas, and Yew. Section 4 develops the algorithm further into a new algorithm that allows fully concurrent reads. Section 5 describes our implementation details and experimental results. Section 6 concludes the paper with remarks on future work.

2 INSPECTOR/EXECUTOR Scheme and Previous Work

The basic idea of the INSPECTOR/EXECUTOR scheme is for the compiler to generate an inspector and an executor for a loop to be parallelized at run-time. The inspector identifies cross-iteration dependencies and produces a parallel execution schedule. The executor uses this schedule to perform the actual operations of the loop. The INSPECTOR/EXECUTOR scheme provides a run-time parallelization framework, and leaves strategies for dependence analysis and scheduling unspecified. The scheme can also be restructured to decouple the scheduling function from the inspector and to merge it with the executor. The scheduling function can even be extracted to serve as a stand-alone routine between the inspector and the executor. There are many run-time parallelization algorithms belonging to the INSPECTOR/EXECUTOR scheme. They differ from each other mainly in their structures and strategies used in each routine, in addition to the type of target loops considered.

Pioneering work on using the INSPECTOR/EXECUTOR scheme for run-time parallelization is due to Saltz and his colleagues [11]. They considered loops without output dependencies (*i.e.* the indexing function used in the assignments of the loop body is an identity function), and proposed an effective DOALL INSPECTOR/EXECUTOR scheme. Its inspector partitions the set of iterations into a number of subsets, called wavefronts, that maintain cross-iteration flow dependencies. Iterations within the same wavefront can be executed concurrently, but those in different wavefronts must be processed in order. The executor of the DOALL scheme enforces anti-flow dependencies during the execution of iterations in the same wavefront. The DOALL INSPECTOR/EXECUTOR scheme has been shown to be effective in many real applications. It is applicable, however, only to those loops without output dependencies. The basic scheme was recently generalized by Leung and Zahorjan to allow general cross-iteration dependencies as shown in Figure 1[7]. In their algorithm, the inspector generates a wavefront-based schedule, maintains output and anti-flow dependencies as well as flow dependencies; the executor simply performs the loop operations according to the wavefronts of iterations.

Note that the inspector of the above scheme is sequential. It requires time commensurate with that of a serial loop execution. Parallelization of the inspector loop was also investigated by Saltz, et al. [11] and Leung and Zahorjan [6]. Their techniques respect flow dependencies, and ignore anti-flow and output dependencies. Most recently, Rauchwerger, Amato and Padua presented a parallel inspector algorithm for a general form of loops [9]. They extracted the function of scheduling and explicitly presented an inspector/scheduler/executor scheme. Both their inspector and scheduler are parallel.

DOALL INSPECTOR/EXECUTOR schemes assume a loop iteration as the basic scheduling unit in the inspector and the basic synchronization object in the executor. An alternative to the scheme is DOACROSS INSPECTOR/EXECUTOR parallelization techniques which assume a memory reference of the loop body as the basic unit of scheduling and synchronization. Processors running the executor are assigned iterations in a wrapped manner and each spin-waits as needed for operations that are necessary for its execution. An early study of DOACROSS run-time parallelization techniques was conducted by Zhu and Yew [13]. They proposed a scheme that integrates the functions of dependence analysis and scheduling into a single executor.

Later, the scheme was improved by Midkiff and Padua to allow concurrent reads of the same array element by several iterations [8]. Even though the integrated scheme allows concurrent analysis of cross-iteration dependencies, tight coupling of the dependence analysis and the executor incurs high synchronization overhead in the executor. Most recently, Chen, Torrellas and Yew developed the DOACROSS technique by decoupling the function of the dependence analysis from the executor [4]. Separation of inspector and executor not only reduces synchronization overhead in the executor, but also provides possibility of reusing the dependence information developed in the inspector across multiple invocations of the same loop. Their inspector is parallel, but sacrifices concurrent reads of the same array element. One of the contributions of this paper is to improve Chen, Torrellas and Yew's algorithm to allow concurrent reads.

DOACROSS INSPECTOR/EXECUTOR parallelization techniques provide potential to exploiting fine-grained parallelism across loops. Fine-grained parallelism does not necessarily lead to overall performance gains without an efficient implementation of the executor.

3 A Time-Stamping Algorithm Allowing Partially Concurrent Reads

This section presents a parallel DOACROSS INSPECTOR/EXECUTOR algorithm (PCR, for short) along the lines of Chen-Torrellas-Yew's work. The algorithm allows partially concurrent reads without incurring any extra overhead in the inspector. The next section presents a new algorithm (FCR, for short) that allows fully concurrent reads.

Consider the general form of loops in Figure 1. It defines a two dimensional iteration-reference space. The inspector of the algorithm examines the memory references in a loop and constructs a dependence chain for each data array element in the iteration-reference space. Each reference in a dependence chain is assigned a *stamp*, indicating its earliest access time relative to the other references in the chain. A reference can be activated if and only if the preceding references are finished. The executor schedules references to a chain through a *logical clock*. At a certain time, only those references whose stamps are equal to or larger than the time are allowed to proceed. Dependence chains are associated with clocks ticking at different speeds.

3.1 Serial Inspector and Parallel Executor

We assume stamps of references as discrete integers. The stamps are stored in a two-dimensional array $stamp$. Let (i, j) indicate the j^{th} access of the i^{th} iteration. $stamp[i][j]$ represents the stamp of reference (i, j). The stamping rules of the inspector algorithm are as follows.

(S1) References at the beginning of dependence chains are assigned one.

(S2) For a reference (i, j) in a dependence chain,

- **(S2.1)** if the reference is a write operation, $stamp[i][j]$ is set to the total number of accesses ahead in the chain plus one;
- **(S2.2)** if the reference is a read operation and its immediate predecessor (say at (m, n)) is also a read operation, $stamp[i][j]$ is set to $stamp[m][n]$;
- **(S2.3)** if the reference (i, j) is a read and its immediate predecessor (m, n) is a write, $stamp[i][j]$ is set to the total number of references ahead in the chain plus one.

Assume the indirect arrays are

$$u = [15, 5, 5, 14, 10, 14, 12, 11, 3, 12, 4, 8, 3, 10, 10, 3]$$
$$v = [3, 13, 10, 15, 0, 8, 10, 10, 1, 10, 10, 15, 3, 15, 11, 0].$$

Applying the above stamping rules to the target loop, we obtain the stamped dependence chains labeled by array elements as shown in Figure 2. All references to the same array element form a chain.

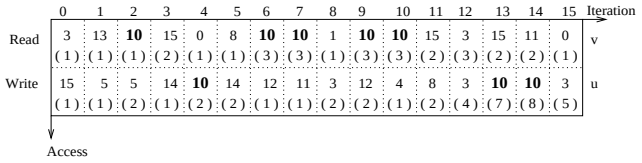

Figure 2: Sequentially constructed dependence chains labeled by array elements. The numbers in parentheses are the stamps of references.

From this figure, it can be seen that the stamp difference between any two directly connected references is one except for pairs of write-after-read and read-after-read references. In pairs of read-after-read, both reads have an equivalent stamp. In pairs of write-after-read, their difference is always the size of the read group minus one.

In the executor, we define a logical clock for each dependence chain. We index dependence chains by the indices of the data elements. Let $time[k]$ represent the current clock time of a chain k. We set up the following clocking rules in the executor corresponding to the inspector's stamping rules.

(C1) Initially, all clocks ($time[k]$) are set to one.

(C2) A reference (i, j) in dependence chain k is triggered if $stamp[i][j] \leq time[k]$.

(C3) For a clock associated with chain k, $time[k]$ is incremented by one if a reference in the dependence chain k have been activated.

It is easy to show that the above INSPECTOR/EXECUTOR algorithm respects all dependencies and allows concurrent reads. Look at the dependence chain 10. From the stamps of references, it is clear that the four read operations in iterations 6, 7, 9 and 10 can proceed simultaneously once the write in iteration 4 is done. The write operation in iteration 13 cannot be activated until all four reads are finished.

3.2 Parallel Inspector and Parallel Executor

The above algorithm builds the stamp table sequentially. Building dependence chains that reflect all types of dependencies is a very time-consuming process. It requires an examination of all references at least once in the loop. A basic parallel strategy is to partition the entire iteration space into a number of regions. Each region, comprising a number of consecutive iterations, is assigned to a different processor. Each processor establishes its local dependence chains by examining the references in its local region. Processors then exchange information about their local dependence chains and finally connect them into complete chains.

To apply the algorithm to the parallel construction of the time table, one key issue is how to stamp the references in a dependence chain across regions, reflecting all dependencies and allowing independent references to be performed in parallel. Since all processors (except the first) have no knowledge about the references in regions ahead, they are unable to stamp their local references in a local chain without the assignment of its head. Suppose there are four processors that are cooperatively building a stamp table like Figure 2. Each processor examines four consecutive iterations. We label processors participating in the parallelization by their region indices. Consider references in region 3 to memory location 3. Since processor 3 does not know whether there are dependent references in regions from 0 through 2 and what their stamps are, it is unable to stamp local references $(12, 0)$, $(12, 1)$ and $(15, 1)$.

To allow processors to continue with the examination of other references in their local regions in parallel, Chen *et al.* [4] proposed a *conservative* approach for an inspector to assign a conservative number to the second reference of a local chain and leave the first one to be decided in a subsequent global analysis. By this conservative approach, processor 3 temporarily assigns 24 plus 1 to the reference $(12, 0)$, assuming all 24 accesses in regions from 0 to 2 are in the same dependence chain. The extra one is due to the reference $(12, 0)$. It results in a stamp of 26 in the subsequent reference $(15, 1)$, as shown in Figure 3. The negative sign of a stamp indicates the stamp is temporarily recorded for the calculations of subsequent references' stamps.

Borrowing the idea of conservative approach, we set up one more stamping rule that assigns references in a local read group the same stamp.

(S3) For a reference (i, j) in iteration region r,

- **(S3.1)** if the reference is at the beginning of a local chain, $stamp[i][j]$ is set to $-(n + 1)$, where n is the total number of accesses in the regions from 0 through $r - 1$;
- **(S3.2)** if the head reference is a read, all subsequent consecutive reads are set to $-(n + 1)$.

Applying the above stamping rule, together with the rules of the serial inspector, on local dependence chains, we obtain partially stamped dependence chains as presented in Figure 3. There are three partially stamped dependence chains associated with array elements 3, 10, and 15 in Figure 3. The dependence chains of other elements are omitted for clarity.

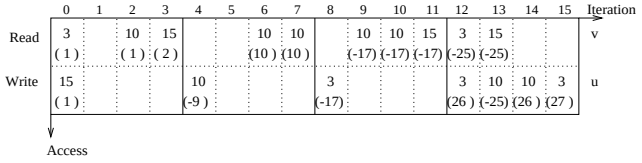

Figure 3: A partially stamped dependence chain labeled by array elements. Numbers in parentheses are stamps of references.

Using the conservative approach, most of the stamp table can be constructed in parallel. Upon completion of the local analysis, processors communicate with each other to determine the stamps of undecided references in the stamp table. Processor 2 sets $stamp[8][1]$ to 2 after communicating with processor 0 (processor 1 marks no reference to the same location). At the same time, processor 3 communicates with processor 2, but gets an undecided stamp on the reference $(8, 1)$, and hence assigns another conservative number, 16 plus 1, to reference $(12, 0)$, assuming all accesses in regions 0 and 1 are in the same dependence chain. The extra one is due to the total number of dependent references in region 2. Note that the communications from processor 3 to processor 2 and from processor 2 to processor 1 are in parallel. Processor 2 can provide processor 3 only the number of references in the local region till the end of the communication with processor 0.

Generally, processors communicate with processors ahead in the global analysis phase to determine their undecided references using the following rules.

(G1) For an undecided write reference (i, j),

- **(G1.1)** if its predecessor, say (m, n) in region r, has a stamp t ($t > 0$) $stamp[i][j]$ is set to t;
- **(G1.2)** if its predecessor (m, n) in region r is UNDECIDED, $stamp[i][j]$ is set to the sum of the total

number of accesses in regions from 0 to $r-1$ and the total number of references in the same chain in the region r plus one.

(G2) For an undecided read reference (i, j),

(G2.1) if its predecessor, say (m, n) in region r is a write, $stamp[i][j]$ is set in a way similar to rule (G1);

(G2.2) if its predecessor (m, n) is a read,

(G2.2.1) if the reference (m, n) is in the same region, $stamp[i][j]$ is set to $stamp[m][n]$;

(G2.2.2) if the reference (m, n) is in a different region r,

(G2.2.2.1) $stamp[i][j]$ is set to the sum of the total number of accesses in regions from 0 to $r-1$ and the total number of relevant references in the region r plus one;

(G2.2.2.2) $stamp[i][j]$ is set to $stamp[m][n]$ plus the number of reads in the read group of reference (m, n).

Figure 4 shows the complete dependence chains associated with array elements 3, 10 and 15.

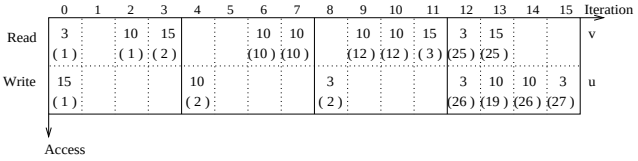

Figure 4: A fully stamped dependence chain labeled by array elements. Numbers in parentheses are stamps of references.

Accordingly, the third clocking rule of the executor in Section 3.1 is modified as follows.

(C3) For a clock associated with a dependence chain k,

(C3.1) $time[k]$ is incremented by one if a reference (i, j) in region r and belonging to chain k is activated, and $stamp[i][j] \leq n$, where n is the total number of accesses from region 0 to region $r-1$;

(C3.2) $time[k]$ is set to $n+2$ if an activated reference (i, j) satisfies $stamp[i][j] > n$.

For example, consider references in the dependence chain on element 10 in Figure 4. The reference $(2, 0)$ first triggers the reference $(4, 1)$. Activation of reference $(4, 1)$ will set $time[10]$ to 10 because $stamp[4][1] = 2$, which is less than 8, the total number of accesses in the first region. There are two concurrent reads $(6, 0)$ and $(7, 0)$ in region 1 and two concurrent reads $(9, 0)$ and$(10, 0)$ in region 2. One of the reads in region 2 will set $time[10]$ to 18, and the other increments $time[10]$ to 19. The write $(13, 1)$ is then triggered. The last is the write $(14, 1)$.

Note that this parallel inspector algorithm only allows consecutive reads in the same region to be performed in parallel. Read operations in different regions have to be performed sequentially even though they are consecutive in the final dependence chains. In the dependence chain on element 10 in Figure 4, for example, the reads $(9, 0)$ and $(10, 0)$ are activated after reads $(6, 0)$ and $(7, 0)$. We are able to assign reads $(9, 0)$ and $(10, 0)$ the same stamp as reads $(6, 0)$ and $(7, 0)$, and assign the reference $(13, 1)$ a stamp of 14. This dependence chain, however, will destroy the anti-flow dependencies from $(6, 0)$ and $(7, 0)$ to $(14, 0)$ in the executor if reference $(9, 0)$ or $(10, 0)$ starts earlier than one of the reads in region 1.

4 A Time-Stamping Algorithm Allowing Fully Concurrent Reads

This section presents a new algorithm that allows fully concurrent reads. The basic idea is to use real numbers to represent clock time in the synchronization of references in a dependence chain. Write operations and read groups can each be regarded as a macro-reference. For a write reference or the first read operation in a read group in a dependence chain, the inspector stamps the reference with the total number of macro-references ahead. Other accesses in a read group are assigned the same stamp as the first read. Correspondingly, in the executor, the clock of a dependence chain is incremented by one time unit on a write reference and by a *fraction* of a time unit on a read operation. The magnitude of an increment on a read operation is the reciprocal of its read group size.

Figure 5 presents sequentially stamped dependence chains. In addition to the stamp, each read reference is also associated with an extra data item recording its read group size. In an implementation, the variable for read group size can be combined with the variable for stamp. For simplicity of presentation, however, they are declared as two separate integers. Look at the dependence chain on element 10. The reference $(4, 1)$ triggers four subsequent reads: $(6, 0)$, $(7, 0)$, $(9, 0)$ and $(10, 0)$ simultaneously. Activation of each of these reads increments the clock time by $1/4$. After all of them are finished, the clock time reaches 4, which in turn activates the reference $(13, 1)$. Due to space limitations, readers are referred to [14] for the details of the algorithm.

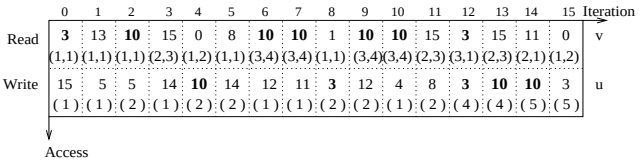

Figure 5: Sequentially constructed dependence chains labeled by array elements. Numbers in parentheses are stamps of references.

5 Performance Evaluation

This section presents our experimental results obtained on a SUN SPARCserver 630MP, using Solaris thread tools. The system is a 4-Sparc-40MHz shared memory multiprocessor. All programs were written in C and the Solaris thread libraries, and compiled by *gcc* with the $O2$ optimization option.

5.1 Implementation

Three algorithms were implemented in this experiment: the PCR algorithm allowing partially concurrent reads, the FCR algorithms allowing fully concurrent reads and the Chen-Torrellas-Yew algorithm (CTY, for short). All algorithms were implemented as three routines: a local_inspector, a global_inspector, and an executor. They are separated by barrier synchronization operations.

All algorithms were programmed in the Single-Program-Multiple-Data (SPMD) paradigm. At the beginning, threads were created in a bound mode so that each thread is bounded into a different CPU and runs to completion. In the local and global inspectors, each thread is assigned a block of consecutive iterations. Threads share a global table: $stamp$. Threads in the local inspector work on non-overlapped fragments of the table and hence run in parallel. Threads in the global inspector can also proceed without synchronization, even though they require communication with each other to determine undecided element stamps. In the implementation, each thread in the local inspector records the first and the last references of each local dependence chain in two auxiliary arrays: *head* and *tail*. In the global inspector, threads update the stamp table through communicating their read-only head and tail arrays. Since no table elements are updated by more than two threads, all threads can also run in parallel.

In the executor, iterations are assigned to threads in a cyclic way. Threads are synchronized by mutual exclusion locks. For accessing an element at location k in the execution of a reference (i, j), a thread first needs to acquire a mutual exclusion lock. After obtaining the lock, it checks whether the reference is ready for execution. After accessing the data, it updates the clock time and broadcasts the new time to threads that are blocked on the element. If more than two threads are blocked on the element at the time, they will compete again for the mutual exclusion lock after their release.

5.2 Results

Performance of a run-time parallelization algorithm is dependent on a number of factors. One is the structure of the target loops. This experiment considered two kinds of loop structures: Single-Read-Single-Write (SRSW) and Multiple-Reads-Single-Write (MRSW), as shown in Figure 6. The SRSW structure refers to a loop which comprises a sequence of interleaved reads and writes in its loop body. The MRSW structure refers to a loop that begins with a sequence of reads and ends with a write operation in its loop body.

SRSW structure:

```
for (i = 0; i < N; i++) {
    for(j = 0; j < A; j++)
        if (odd(j))     dummy = X[u[i][j]];
        else            X[u[i][j]] = dummy;
}
```

MRMW structure:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < A - 1; j++)
        dummy = X[u[i][j]];
    X[u[i][A-1]] = dummy;
}
```

Figure 6: Single-Read-Single-Write (SRSW) versus Multiple-Read-Single-Write (MRSW) loop structures.

Another major factor affecting the overall performance is memory access patterns defined by the index array $u[i][j]$. Two memory access patterns were considered: uniform and non-uniform access patterns. A uniform access pattern (UNIFORM, for short) was generated by assuming all array elements have the same probability of being accessed by a memory reference. A non-uniform access pattern (NUNIFORM, for short) was generated by assuming that 90% of references are to 10% of array elements. Non-uniform access patterns reflect hot spots in memory accesses and result in long dependence chains.

Figure 7 presents the overall time of the PCR, FCR and CTW algorithms spent on the parallel execution of loops ranging from 64 to 4096 iterations in a loop. Assume each iteration has 4 memory accesses ($A = 4$), and each access incurs a loop of dummy operations (about $50\mu s$ computational workload). The delay due to the dummy loop inside a program could be greatly reduced by cache systems during the sequential execution of the program. For fairness in

comparison, we set the sequential execution time of the program to be the multiplication of the total number of accesses by $50\mu s$.

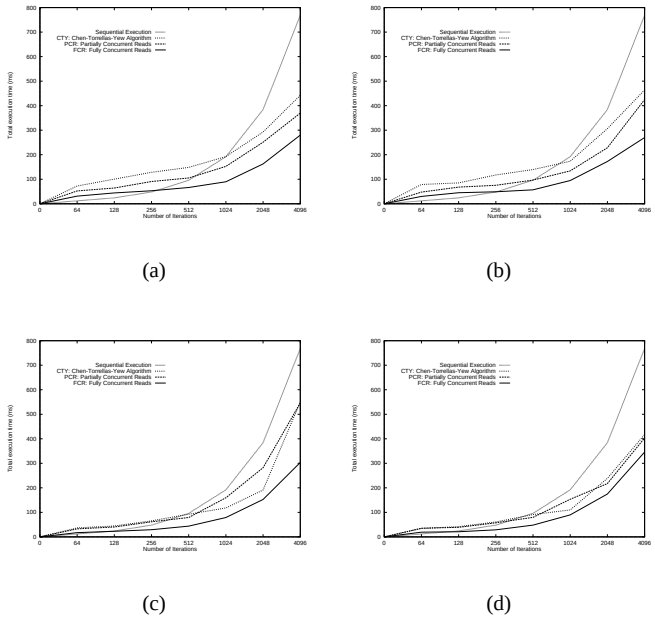

Figure 7: Parallelization of loops (a) with the non-uniform access pattern and the SRSW structure; (b) with the non-uniform access pattern and the MRSW structure; (c) with the uniform access pattern and the SRSW structure; (d) with the uniform access pattern and the MRSW structure

Overall, these four figures show that run-time parallelization is a viable approach to accelerating the execution of loops that are large and can not be handled by compile-time parallelization. All three algorithms we considered in this experiments all show improvements over serial code for large loops. The larger a loop is, the greater their improvement. Of the three algorithms, the FCR algorithm performs best in all test cases. It gains up to an 80% improvement over the CTY algorithm for loops with uniform and non-uniform access patterns. The PCR algorithm outperforms the CTY algorithm for loops with non-uniform access patterns only. For loops with uniform access patterns, they have similar results. These figures also show that the FCR and the PCR algorithms gain few benefits from the MRSW loop structure even though the MRSW structure has many more reads than the SRSW structure. It is because serial execution of the memory accesses in an iteration prevents the FCR and the PCR algorithms from exploiting more cross-iteration parallelism.

As expected, run-time parallelization techniques do not necessarily speed up the execution of loops. Benefits from the parallel execution of small loops could be outweighed by the parallelization overheads. From these figures, it can be seen that each parallelization algorithm has a break-even point in the axis of iteration number, from which parallel execution starts to defeat serial code. The smaller the break-even point, the smaller the parallelization overheads. In all test cases, the plots of the algorithms indicate that the FCR algorithm has the smallest break-even point. The break-even point is as small as 128 for loops with uniform access patterns. For loops with non-uniform access patterns, the turning point of the FCR algorithm increases to 256, because of the hot spot in memory accesses.

The overhead of a parallelization algorithm is mainly due to the dependence analysis in the local and global inspectors and synchronization of references in the executor. Table 1 presents overhead profiles of the three algorithms for the parallelization of different sizes of loops. Compared to the CTY algorithm, it is clear that the FCR algorithm reduces the time spent on the executor significantly at slightly more overheads in its local and global inspectors. The FCR algorithm lends itself well to the parallelization of loops that are contained inside sequential loops. If the dependence analysis can be reused across multiple loop invocations, the FCR algorithm can achieve up to a 3.7 times speedup over serial codes for loops with uniform access patterns, and up to 3.4 times speedup for loops with non-uniform access patterns.

6 Conclusions

In this paper, we present two new run-time techniques for the parallelization of loops that have indirect access patterns. Our schemes can handle any type of loop-carried dependencies. They follow the DOACROSS INSPECTOR/EXECUTOR approach and improve upon previous algorithms with the same generality by allowing concurrent reads of the same location and by increasing the overlap of dependent iterations. The algorithms are implemented based on stamping rules and using multithreading tools. The experimental results on an SMP server with four processors show that our schemes are efficient and outperform their competitors consistently in all test cases. The difference between the two proposed algorithms is that one allows partially concurrent reads without causing extra overhead in its inspector, while the other allows fully concurrent reads at a slight overhead in the dependence analysis. The algorithm allowing fully concurrent reads obtains up to an 80% improvement over the Chen-Torrellas-Yew algorithm. Even for loops with long cross-iteration dependent chains, it achieves speedups over the serial code of up to 3 times with the full overhead of run-time analysis, and of 3.7 times if part of the analysis is reused across multiple loop invocations.

Future work includes evaluation of these algorithms for the parallelization of real application codes on large-scale

		UNIFM-SRSW			UNIFM-MRSW			NUNIFM-SRSW			NUNIFM-MRSW		
		LA	GA	EX	LA	GA	EX	LA	GA	EX	LA	GA	EX
128	FCR	2.42	3.51	11.97	2.52	4.90	8.55	2.80	2.31	34.11	3.13	3.58	32.93
	PCR	2.11	2.87	30.00	2.19	2.26	29.45	2.21	1.54	55.54	2.10	2.37	58.29
	CTY	2.17	2.83	34.50	2.21	2.07	32.53	2.40	3.15	90.00	1.55	1.44	77.13
512	FCR	5.85	6.01	26.90	5.67	10.45	26.81	10.15	3.22	48.00	7.47	6.68	37.68
	PCR	7.30	4.08	62.53	5.07	2.54	68.09	4.82	1.21	94.14	4.76	1.56	85.34
	CTY	4.47	2.83	80.28	4.01	3.04	80.08	3.86	3.12	136.32	3.88	2.29	128.27
2048	FCR	18.53	26.14	103.89	19.37	47.81	102.58	2.03	13.96	120.50	28.08	26.53	112.71
	PCR	12.79	7.81	256.40	13.60	10.25	188.60	12.10	3.98	229.91	12.96	5.76	204.80
	CTY	11.66	6.77	167.90	14.73	8.27	210.80	10.77	3.63	273.10	11.01	4.00	286.27

Table 1: Times spent on the local inspector(LA), the global inspector(GA) and the executor (EX)

systems, and comparison of DOACROSS and DOALL techniques for run-time parallelization using multithreading. Implementation of the algorithms on a Cray J916 with 16 processors is already under way. In addition, we are also investigating issues to parallelize loops that have non-uniform cross-iteration dependences and to integrate both compile-time and run-time techniques into the Stanford SUIF parallelizing compilers [3, 5].

Acknowledgements

We would like to thank Dr. Loren Schwiebert, Jialin Ju and Roy Sumit for their valuable comments on an early version of this paper.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler Transformations for High-Performance Computing", ACM Computing Surveys, Vol.26, No.4, Dec. 1994, pages 345–420.

[2] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. "Massively parallel methods for engineering and science problems". *CACM*,37(4): 31-41, April 1994.

[3] V. Chaudhary, C. Xu, S. Roy, J. Ju, V. Sinha and L. Luo, "Design and Evaluation of an Environment APE for Automatic Parallelization of Programs", In *Proc. of 2nd Int. Symp. on Parallel Algorithms and Networks*, IEEE Computer Society, June 1996.

[4] D. K. Chen, P. C. Yew, and J. Torrellas. "An efficient algorithm for the run-time parallelization of doacross loops". In *Proc.of Supercomputing* 1994, pp. 518-527, Nov.1994.

[5] J. Ju and V. Chaudhary. "Unique sets oriented partitioning of nested loops with non-uniform dependences", ICPP96.

[6] S.-T. Leung and J. Zahorjan. "Improving the performance of runtime para- llelization". In *4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 83-91, May 1993.

[7] S.-T. Leung and J. Zahorjan. "Extending the applicability and improving the performance of runtime parallelization", Technical Report, Department of Computer Science, University of Washington, 1995.

[8] S. Midkiff and D. Padua. "Compiler algorithms for synchronization". *IEEE Trans. on Computers*, C-36(12),December 1987.

[9] L. Rauchwerger and D. Padua. "The LRPD test: Speculative run-time paralle- lization of loops with privatization and reduction parallelization". In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June,1995.

[10] L. Rauchwerger, N. M. Amato, and D. A. Padua. "Run-time methods for para- llelizing partially parallel loops". In *Int. Conf. on Supercomputing*, 137–146.

[11] J. Saltz, R. Mirchandaney, and K. Crowley. "Run-time parallelization and scheduling of loops". *IEEE Trans. Comput.*,40(5),May 1991.

[12] Z. Shen, Z. Li, and P. C. Yew, "An empirical study on array subscripts and data dependencies," in *Proc. of ICPP*, pp. II–145 to II–152, 1989.

[13] C. Zhu and P. C. Yew. "A scheme to enforce data dependence on large multi- processor systems". *IEEE Trans. Softw. Eng.*, 13(6):726-739, 1987.

[14] C. Xu and V, Chaudhary, "Efficient algorithms for parallelization of loops at run-time", Tech. Rept. PDCL-96-05-01, Dept. of ECE, Wayne State University, May 1996.