Run-time parallelization of irregular DOACROSS loops[1]

J. Thulasiraman[2] V. P. Krothapalli[3] M. Giesbrecht[4]

Technical Report No. 95/03

Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2, Canada.

June 5, 1995

[1]Research was supported in part by Natural Sciences and Engineering Research Council of Canada research grants OGP0155376 and OGP0138482. This work has been submitted for acceptance as a paper at IRREGULAR '95, Workshop on Parallel algorithms for irregularly structured problems, September 4-6, 1995, Lyon, France.

[2]Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada, R3T 2N2, phone: (204)474-8827, fax:(204)269-9178, email: `tram@cs.umanitoba.ca`

[3]Currently at Informix Software Inc., 4100 Bohannon Drive, Menlo Park, CA 94025, USA, email: `prasadk@informix.com`

[4]Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada, R3T 2N2, phone: (204)474-8838, fax:(204)269-9178, email: `mwg@cs.umanitoba.ca`

Contents

Abstract

Dependencies between iterations of an irregular DOACROSS loop cannot always be determined at compile-time because these dependencies may be dependent upon input data which is known only at run-time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, it is necessary to perform run-time analysis. In this paper, we present a new algorithm to parallelize these loops at run-time. The proposed algorithm handles all types of data dependencies without requiring any special architectural support in the multiprocessor. Our scheme has an inspector which builds the iteration schedule and an executor which uses the schedule to execute the various iterations. Our approach does not require any special synchronization operations during the inspector stage and the executor can be implemented with or without synchronization support. It allows overlap among dependent iterations and requires very little inter-processor communication. Further, the schedule formed by the inspector can be reused across loop invocations. We evaluate our algorithm with parametrized loops running on a 8-processor UMA (Uniform Memory Access) shared memory multiprocessor. The results show significant speedups over the serial code with the full overhead of run-time analysis. Furthermore, compared to an older scheme with the same generality, our scheme has a consistent performance (i.e., performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and ensures good speedup during the executor stage.

1 Introduction

The objective of the work presented here is to effectively schedule the iterations in a given loop without violating the inter-iteration dependencies, in order to exploit maximum parallelism. To achieve maximal parallelism, the various dependencies between loop iterations have to be analyzed. The dependencies possible between any two statements S1 and S2 in sequence can be classified as:

1. A *flow* or true dependency, if the statement S2 uses a variable assigned by the statement S1.
2. An *anti-dependency*, when S2 assigns a variable used by the statement S1.
3. An *output-dependency*, when statements S1 and S2 assign the same variable.

Loop-level parellelism is an important source of speedup in multiprocessors that exploit medium-grain parallelism. To parallelize a loop, it is necessary to identify the data dependence relations between loop iterations via dependence analysis [1]. If the loop does not have any dependencies across iterations, then it can run in parallel. Otherwise, it has to be run serially or, if parallelism is to be exploited, proper synchronization features have to be inserted in the code to ensure that the memory accesses are performed in the correct sequence. The latter type of loop is called a DOACROSS loop.

Some DOACROSS loops can be transformed into fully parallel loops. For example, if the difference in iteration numbers between dependent iterations (called the dependence distance) is known at compile-time, techniques like loop-alignment, loop skewing, or dependence uniformization [10, 16, 3, 14] can be used to run the loop in parallel. But, for example, when array elements are accessed via subscripted subscripts, it is not possible to resolve the dependencies between the iterations of the loop structure at compile-time, since the values of the subscripted subscripts are known only at run-time. In these cases, the compiler cannot prove the independence of iterations. Therefore, even if the loop turns out to be parallel in nature, compile-time schemes fail to parallelize the loop.

/ Index arrays I1 and I2 contain values in the range 1 to N. There are M iterations in the DOACROSS loop. Array A contains N elements. */*

do $i = 1$ *to* M

$\quad\cdots$

S_p: $A[I1[i]] = \cdots$

$\quad\cdots$

S_q: $\cdots = A[I2[i]] + \cdots$

$\quad\cdots$

enddo

Figure 1: A generic case of an irregular DOACROSS loop.

In the above loop, the entries in the index arrays *I1* and *I2* have no restriction and are not available until run-time. Each array can have duplicated values, which implies that all three kinds of data dependencies, namely flow (Read-after-Write), anti- (Write-after-Read) and output- (Write-after-Write) dependencies, could occur between instances of statements S_p and S_q. For example, if $I1[1] = I2[3] = I1[4] =$

$I1[5] = x$, then we have the following dependencies: $S_p[1] \xrightarrow{flow} S_q[3] \xrightarrow{anti-} S_p[4] \xrightarrow{output-} S_p[5]$. These accesses form a *dependence (or access) chain.* The entries in the arrays *I1* and *I2* are assumed to remain constant during the execution of one invocation of the loop, although they are allowed to change outside the loop.

For loops not amenable to compile-time parallelization, we can still perform the dependence analysis at run-time and maybe execute the loop in parallel. This approach is called *run-time parallelization.* Much previous research has been done to design effective run-time parallelization algorithms [2, 4, 5, 6, 8, 9, 12, 17]. The main differences among the schemes proposed are the types of dependence patterns that are handled and the required system or architecture support. The key to success in these schemes is to minimize the time spent on dependency analysis and on process synchronization. Indeed, if too much time is spent on these overheads, it is better to run the loop serially. In general, run-time parallelization schemes have two stages, namely *inspector* and the *executor* [17, 6, 12, 2]. The inspector determines the dependence relations among the data accesses. The executor uses this information to execute the iterations in parallel in an order that preserves the dependences.

In this paper, we describe and evaluate a new algorithm for the run-time parallelization of DOACROSS loops. We follow the inspector and executor approach. In our scheme, the *inspector* inspects the various dependencies existing between the iterations and constructs an iteration schedule. This involves reordering the iterations. The entire set of iterations is partitioned into subsets called wavefronts. A wavefront is a grouping of the iterations which can be executed concurrently. A specific wavefront contains a collection of iterations which have no dependencies among them. Each one of the iterations in a loop is assigned to a specific wavefront based on its position in the dependence hierarchy. The wavefronts are executed sequentially whereas the iterations contained in a wavefront are executed concurrently. Since the wavefront number *WF[i]* for iteration i is one plus the maximum of the wavefront numbers of the iterations upon which it depends, iteration i is assigned a wavefront number such that all the iterations which it may depend upon are in wavefronts which would have completed execution by the time it is scheduled. All the loop iterations are assigned specific wavefront numbers.

The *executor* is a transformed version of the source code loop structure. This carries out the actual execution of iterations according to the schedule produced by the inspector. It uses the wavefront information produced by the inspector to schedule the iterations. It starts executing the iterations clustered in the first wavefront concurrently by scheduling them on multiple processors. It then does the same with the iterations clustered in the next wavefront and so on until it reaches the last wavefront. Thus, the iterations within a particular wavefront are executed concurrently whereas successive wavefronts are executed in a strictly serial fashion.

Our scheme handles all types of data dependencies without requiring any special architectural support. It does not require any special synchronization operation during the inspector stage and the executor can be implemented with or without synchronization support. It allows overlap among dependent iterations and requires very little inter-processor communication. The effectiveness of our scheme is evaluated via measurements of parametrized loops on a 8-processor UMA (Uniform Memory Access) shared memory multiprocessor. The results show significant speedups over the serial code with the full overhead of run-time analysis. Furthermore, compared to an older scheme with the same generality, our scheme has a consistent performance (ie., the performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and ensures good speedup during the executor stage.

We organize this paper as follows. Section 2 presents the previous work on run-time parallelization; Section 3 explains our scheme in detail; Section 4 describes our experimental methodology; and Section 4.2 presents the performance results. Section 5 concludes our study.

2 Previous work

Zhu and Yew [17] propose an algorithm which handles all types of inter-iteration dependencies discussed above. Their algorithm also has a fully parallel inspector and executor. However, its inspector cannot be reused across multiple invocations of the

DOACROSS loop because the inspector and the executor are tightly coupled. As well, it requires many memory accesses, which reduces the overall speedup. Midkiff and Padua [9] improve Zhu and Yew's scheme by allowing concurrent reads to the same array element by several iterations. To do so, each data element requires more than one key field. However, their algorithm, like Zhu and Yew's, requires high communication. Other less general schemes have also been proposed [11].

Krothapalli and Sadayappan [5, 4] suggest a method to remove anti- and output-dependencies by analyzing the reference pattern generated and by using extra copies of the variables. Their algorithm is parallel in nature but requires an atomic *Fetch&Add* operation. The additional storage space required is proportional to the number of references. Krothapalli and Sadayappan [6, 4] describe a technique called pre-synchronized scheduling for dynamically scheduling the iterations of a DOACROSS loop. They further show how to use their scheme on vector processors. Their inspector requires the run-time generation of the Iteration Space Dependency Graph (ISDG) (which can be done in parallel), which introduces run-time overhead. Saltz *et al.*'s [12] approach does not take care of anti- and output- dependencies at the inspector stage and therefore cannot handle all the data dependencies at the inspector stage.

Leung and Zoharjan [8] suggest two techniques to parallelize the wavefront computation during the inspector stage. They are:

Sectioning:

The M iterations are partitioned into P sections, where P is the number of processors. Each section is allocated to one processor. Thus, the wavefront computation goes on in a parallel fashion. Then, the wavefronts computed in each section are grouped together and numbered sequentially. The shortcoming of sectioning is that it produces more wavefronts when compared to the sequential execution of the inspector. This does not affect the overall efficiency as long as the number of processors available for computation is far less than the number of iterations per wavefront. However, as the number of processors is sufficiently high to handle the iterations in a given wavefront, then sectioning is not preferred, because it produces more wavefronts which leads to reduced parallelism among the iterations.

Bootstrapping:

This aims to improve upon sectioning by reducing the total number of wavefronts generated. Sectioning is applied on the source loop and a valid schedule is obtained. Then the schedule obtained is applied on the inspector loop in a wavefront-by-wavefront fashion. This generates fewer wavefronts due to the optimization involved during the second phase.

2.1 Chen *et al.*'s scheme

The recent work by Chen *et al.* [2] handles all types of dependencies at the inspector stage and the inspector results can be reused across multiple invocations of the same loop structure if the dependencies do not change from one invocation of the DOACROSS loop to another. During the inspector stage, all the dependence information is gathered and stored in a table called the *Ticket* table. The inspector stage is divided into two parts, namely the local and the global inspector phase.

During the local inspector phase, the entire iteration space is divided into different sections and distributed among the available processors. The number of sections is equal to the number of available processors. Each processor records a sequence number in the *Ticket* table for each one of the iteration dependent references in each iteration in its own section. However, it does not assign any sequence number to the first reference in any dependence chain in its section. The sequence number for the first reference in each dependence chain in each one of the sections will be assigned during the global inspector phase. Furthermore, during the local inspector phase, each processor constructs a linked list called the *Unresolved list* with nodes corresponding to the *Ticket* table entries in its section which have not been assigned sequence numbers. Each node contains the index of the shared data structure element, the corresponding entry in the *Ticket table* and a transition sequence number. The transition sequence number is a possible sequence number for the first reference to the same element of the shared data structure in the other following sections. During the global inspector phase, the unfilled entries in each section of the *Ticket* table are filled by their corresponding processors by communicating with other processors using the *Unresolved list* maintained by each one of the processors.

The executor makes use of the dependence information recorded in the *Ticket* table to execute each one one of the iterations. It makes use of a shared variable *Ready*, for each one of the iteration dependent accesses. During any stage of execution, if the value of the *Ready* variable corresponding to any one of the iteration dependent accesses does not equal its corresponding *Ticket* table entry, then the particular iteration dependent access *busy-waits*. The executor can either be *pre-scheduled* or *self-scheduled*.

In their scheme, the synchronization operations during the inspector are implemented using *Cedar* synchronization primitives (like *Cedar_sync*), which have hardware support and give relatively better performance when compared to simple *lock* and *unlock* synchronization primitives. To implement *Cedar* synchronization primitives on other multiprocessing machines requires some architectural support. During the inspector stage, building the *Ticket* table is very expensive, because it requires inter-processor communication. Furthermore, during the executor stage accesses to iteration dependent variables *busy-wait*. Since the iteration space to processor mapping is not done with a global picture of the dependence hierarchy, it leads to longer *busy-waits* during the executor stage. As well, their algorithm serializes reads with input dependencies, which introduces pseudo-dependencies among truly concurrent iterations.

3 A new algorithm for run-time parallelization

In order to exploit the latent parallelism in the irregular DOACROSS loop (Figure 1), we follow the inspector and executor approach. During the inspector stage, shadow memory locations (variables) are created for each one of the iteration dependent variables. The shadow variables register the iteration in which a particular iteration dependent variable was last read and/or updated. Using the shadowed information pertaining to each iteration dependent variable, the various dependencies between iterations are analyzed, and an appropriate wavefront number is allocated to each one of the iterations. The proposed dependence analysis takes care of all kinds of data dependencies.

```
/* Index arrays I1 and I2 contain values in the range 1 to N. There are M iterations
in the loop. There are N elements in the arrays LastW and LastR. There are M
elements in the array WF. L is an array of pointers (size M) to linked lists. */
WF[:] = 0; L[:] = NULL; LastW[:] = 0; LastR[:] = 0;
do i = 1 to M
    mywf=0;
    if (LastW[I1[i]] ≠ 0) /* output- dependency */
        { mywf = max(WF[LastW[I1[i]]],mywf); }
    if (LastR[I1[i]] ≠ 0) /* anti- dependency */
        { mywf = max(WF[LastR[I1[i]]],mywf); }
    if (LastW[I2[i]] ≠ 0) /* flow dependency */
        { mywf = max(WF[LastW[I2[i]]],mywf); }
    WF[i] = mywf + 1; /* wavefront allocation */
    Add_node(L[WF[i]],i); /* place iteration i at appropriate level */
    LastW[I1[i]] = i; /* update the shadow variables */
    if (WF[LastR[I2[i]]] < WF[i]) { LastR[I2[i]] = i; }
enddo
```

Figure 2: The Inspector algorithm.

In the above inspector algorithm, the arrays *LastW* and *LastR* serve as shadow memory variables for the shared array A. *LastW* records the iteration during which a particular shared array element *A[k]* was last modified. Similarly, *LastR* records the iteration during which a particular shared array element *A[k]* was last read. For each one of the shared array references *A[k]* in a particular iteration, the algorithm finds out the iterations in which it was last modified and last read. This is accomplished using the information (iteration numbers) recorded in the shadow arrays *LastW* and *LastR*. Then the algorithm allots an appropriate wavefront number to the particular iteration. This way an iteration is allotted a wavefront number such that it is numerically greater than the wavefront numbers of the iterations upon which it depends. Further, the inspector builds an array L of linked lists, where *L[j]* contains a list of those iterations in wavefront *j*. This is used during the executor stage to schedule iterations onto different processors. The linked list holds the iteration numbers based on their position in the dependence hierarchy. For example, an iteration with a wavefront number *four* will be placed in the fourth level in the array of linked lists. In the worst case (i.e serial loop) there will be M (total number of iterations) levels.

The dependency information thus recorded in the form of wavefronts can be reused for several invocations of the DOACROSS loop, provided the dependencies do not change. The primary advantage of wavefront generation is that it leads better load balance among processors because the wavefronts carry a global picture of the dependence hierarchy which leads to efficient scheduling of iterations onto processors during the executor stage. The inspector in the new scheme does not require any *atomic synchronization operations* or *barriers* during the inspector stage.

In the aforementioned inspector algorithm, the entire wavefront computation is done sequentially, which defeats the very purpose of parallelization. In order to parallelize the wavefront computation, *sectioning* [8] is performed on the inspector algorithm.

```
/* Index arrays I1 and I2 contain values in the range 1 to N. There are M iterations
in the loop. There are N elements in the arrays LastW and LastR. Arrays LastR and
LastW are local to the procedure. There are M elements in the array WF. L is an
array of pointers (size M) to linked lists. piece is the size of each section. P is the
number of processors. MAXW[:] has P elements. WF[:], MAXW[:] and L[:] are
shared data structures. They are initialized to zero before the light-weight threads are
activated. Thread id values range from 0 to P-1. */
piece = M/P; Myid = get_myid(); /* get my thread ID */
start = Myid*piece; /* get my share */
if (Myid == (P-1)) { last = M; } else { last = start+piece; }
Max = 0; LastW[:] = 0; LastR[:] = 0;
do i = (start+1) to last
    mywf=0;
    if (LastW[I1[i]] ≠ 0) /* output- dependency */
        { mywf = max(WF[LastW[I1[i]]],mywf); }
    if (LastR[I1[i]] ≠ 0) /* anti- dependency */
        { mywf = max(WF[LastR[I1[i]]],mywf); }
    if (LastW[I2[i]] ≠ 0) /* flow dependency */
        { mywf = max(WF[LastW[I2[i]]],mywf); }
    WF[i] = mywf + 1; /* wavefront allocation */
    LastW[I1[i]] = i; /* update the shadow variables */
    if (WF[LastR[I2[i]]] < WF[i]) { LastR[I2[i]] = i; }
    Max = max(Max,WF[i]); /* record largest wavefront in this section */
enddo
if (Myid == 0) { MAXW[Myid+1] = Max; }
else
```

```
    while(MAXW[Myid] == 0); /* Wait for previous section to complete */
    sum = MAXW[Myid]; /* Get largest wavefront upto previous section */
    do i = (start+1) to last
        { WF[i] = WF[i] + sum; } /* Update wavefront numbers in this section */
    MAXW[Myid+1] = sum + Max;
endif
do i = (start+1) to last /* place the iteration i at the appropriate level */
    { Add_node(L[WF[i]],i); }
```

Figure 3: The Parallel Inspector algorithm.

In the parallel inspector, the entire range of iterations is partitioned into consecutive sub-ranges, called *sections*. Each section is assigned to a different processor. Each processor computes a valid parallel schedule for the iterations in its section by applying the sequential inspector algorithm and ignoring any dependencies on iterations outside of its section. After all the processors have finished, we have a schedule for each section. Every such schedule, called a *sub-schedule*, is a mapping from the iterations in the corresponding section to the wavefronts of that section. The overall schedule is formed by concatenating the sub-schedules in the order of the sections. This is done by making processor i communicate the maximum wavefront up to its section to the next processor $i+1$ in sequence until the final section. In the above parallel inspector algorithm, the shared array *MAXW* serves to communicate the largest wavefront up to a particular section to the next section.

Iterations	1	2	3	4	5	6	7	8	9
Access1 (*write*)	1	2	2	1	5	9	7	8	11
Access2 (*read*)	2	9	6	8	9	1	12	10	12
Wavefront No.	1	2	3	2	1	3	1	3	1

Table 1: Sample run of the inspector on a DOACROSS loop with 9 iterations and 2 hot references per iteration.

The assignment of iterations onto processors during the executor stage can be either *pre-scheduled* or *self-scheduled*. In *pre-scheduling*, iterations are mapped onto processors statically, usually in a cyclic fashion. In *self-scheduling*, iteration to processor mapping is done at run-time, where the processors dynamically pick the

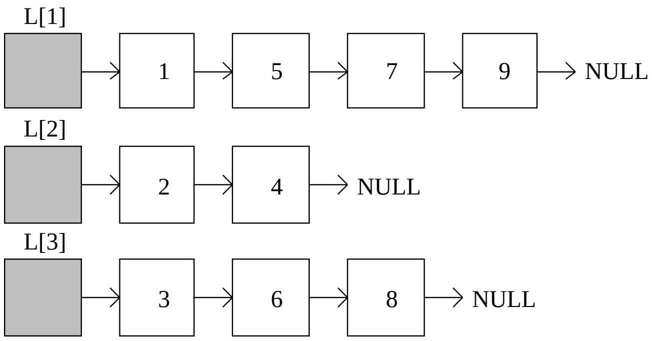

Figure 4: The array of linked lists generated during the sample run.

available iterations either one at a time or in chunks (*chunk scheduling*). To achieve better load balance among the processors, *guided self-scheduling (GSS)* is usually employed, where the free processors pick *(N/P)* of the remaining iterations, where N is the number of remaining iterations to be processed and P is the number of available processors.

Static or *pre-scheduling* of iterations does not lead to efficient iteration to processor mapping. Further it may lead to poor load balance among the processors. Since the position of iterations in the dependence hierarchy is known only at run-time, *self-scheduling* ensures efficient iteration space to processor mapping and better load balance among the processors.

```
/* T is a shared pointer to a node in the array of linked lists L. Level is a shared
variable holding the current wavefront level. T = NULL and Level = 0 initially. LP
is a local pointer to a node in the array of linked lists L. Thread id values range from
0 to P-1. */

Myid = get_myid(); /* get my thread id */
while(1)
    lock(); /* prevents all other accesses to shared memory */
    if (T) T = T→next; /* gets next iteration in current level */
    else /* go to the next level */
        if ((Level+1) == (M+1))
            { unlock(); return; } /* all levels completed */
        if (L[Level+1] == NULL)
            { unlock(); return; } /* all iterations completed */
        unlock(); barrier(); /* synchronization point */
        if(Myid == 0)
            { lock(); Level = Level+1; T = L[Level]; } /* go to next level */
```

```
        barrier(); /* to prevent race condition */
        if (Myid ≠ 0) continue;
    endif
    if (T == NULL) { unlock(); continue; }
    LP = T; /* pick up the iteration to be processed */
    unlock();
    ...
    A[I1[LP→iteration]] = ··· /* process the iteration */
    ...
    ··· = A[I2[LP→iteration]] + ···
    ...
endwhile
```

Figure 5: *Self-scheduled* executor with *barrier synchronization.*

Avoiding inter-wavefront barriers

In *barrier sychronization*, a *barrier* condition has to be met whenever all the iterations in a particular wavefront have been scheduled. In other words, processing of iterations corresponding to wavefront *i+1* should not begin before all the iterations corresponding to wavefront *i* have been completed, because an iteration in wavefront *i+1* may depend upon some of the iterations in wavefront *i*. *The number of barriers required is proportional to the total number of wavefronts.* This obviously reduces the processor utilization as processors tend to idle at the barrier, waiting for other processors to complete. One way to get over the problem of under-utilization of processors is to avoid barriers between wavefronts and to schedule iterations onto free processors irrespective of the wavefront number. But this may override the dependence hierarchy. In order to preserve the dependence structure, a *busy-wait* mechanism may be employed, where the free processors are allowed to process iterations corresponding to different wavefronts, but with the constraint that all accesses to iteration dependent variables *busy-wait*.

In the *busy-wait* mechanism, iterations corresponding to different wavefronts can be scheduled onto processors simultaneously, but accesses to iteration dependent variables *busy-wait* until all previous accesses to those variables have been completed. This involves maintaining a shared variable *Ready*, for each one of the iteration dependent variables to enforce busy-wait. The *busy-wait* mechanism improves the load-balance among the processors and optimizes processor utilization. So, whenever a processor becomes free, instead of waiting for other processors to finish their work corresponding to a specific wavefront, it proceeds with the iterations in the next wavefront. Whenever an iteration dependent variable is accessed, it *busy-waits* until all accesses to the specific variable in preceding wavefronts are over, and then it proceeds. The *busy-wait* gives better performance because of the fact that a particular iteration in a given wavefront may not depend upon all the iterations in the previous wavefront. Thus, *busy-wait* allows overlap of dependent iterations by shifting the dependence granularity from iteration level to the access level. It gives better performance over *barrier synchronization* particularly when the dependence chains are very long; otherwise *barrier synchronization* performs better due to the run-time overheads involved in *busy-wait* mechanism.

The *busy-wait* mechanism needs some data structures to be set up in order to sequence the different accesses. It uses a two-dimensional shared array *sequence table* and another shared array *Ready* to sequence the different accesses in each iteration. Setting up of the *sequence table* has to be done sequentially.

```
/* S_table is a two-dimensional shared array (sequence table). It contains entries
relating to iteration dependent accesses in all the iterations. Ready is a shared array
with N elements. LP is a local pointer to a node in the array of linked lists L. */

Ready[:] = 0;
do j = 1 to Maximum_wavefront_number
    LP = L[j]; /* get the first iteration in level i */
    while LP ≠ NULL /* for each one of the iterations in level i */
        i = LP→iteration; /* get the iteration number */
        /* fill S-table entry with appropriate sequence number */
        S_table[i][1] = Ready[I1[i]];
        Ready[I1[i]] = Ready[I1[i]] + 1; /* increment the sequence number */
        S_table[i][2] = Ready[I2[i]];
        Ready[I2[i]] = Ready[I2[i]] + 1;
        LP = LP→next; /* get the next iteration */
    endwhile
enddo
```

Figure 6: Setting up the *sequence table*.

Iterations	1	2	3	4	5	6	7	8	9
Sequence No. (Access1)	0	2	3	1	0	2	0	1	0
Sequence No. (Access2)	0	1	0	0	0	2	1	0	0

Table 2: *Sequence table* with sequence numbers for the different accesses in each iteration in the sample run.

```
/* T is a shared pointer to a node in the array of linked lists L. Level is a shared
variable holding the current wavefront level. T = NULL and Level = 0 initially.
S_table is a two-dimensional shared array. Ready is a shared array with N elements.
LP is a local pointer to a node in the array of linked lists L. Thread id values range
from 0 to P-1. */

Myid = get_myid(); /* get my thread id */
while(1)
    lock(); /* prevents all other accesses to the shared memory */
    if (T) T = T→next; /* gets next iteration in current level */
    else /* go to the next level */
        if ((Level+1) == (M+1))
            { unlock(); return; } /* all levels completed */
        if (L[Level+1] == NULL)
            { unlock(); return; } /* all iterations completed */
        Level = Level + 1; /* go to next level */
        T = L[Level];
    endif
    if (T == NULL) { unlock(); continue; }
    LP = T; /* pick up the iteration to be processed */
    unlock();
    i = LP→iteration; /* get the iteration number */
    ...
    while( Ready[I1[i]] ≠ S_table[i][1] ); /* busy_wait */
    A[I1[i]] = ···
    ...
    while( Ready[I2[i]] ≠ S_table[i][2] ); /* busy_wait */
    ··· = A[I2[i]] + ···
    ...
endwhile
```

Figure 7: *Self-scheduled* executor with *busy-wait*.

4 The experimental system

Our experiments are timing runs on a 33 MHz 8-processor SGI shared memory multiprocessor. The machine has 4 CPU boards with 2 CPU's on each board. First level cache hits costs 1 processor cycle; first level misses or second level hits costs 3 processor cycles; second level misses costs 57 cycles to bring 64 bytes. The machine is run in single-user mode for all experiments.

4.1 Workloads

The performance of our algorithm depends on some parameters of the loop that we try to parallelize. Table 3 shows what these parameters are (Column 2) and what they measure (Column 3). In our analysis we use a loop as in Figure 8 as workload and try to cover as wide a range of values as possible for these parameters. Column 4 shows the range of values chosen. Array A has $N*r$ elements. The *Hfrac* and *Hsize* parameters are used to simulate different dependence patterns. Each access has a probability *Hfrac* to be a *hot* reference. The hot section is a portion *Hsize* of array A. Loops with long dependence chains and therefore low parallelism have high *Hfrac* and low *Hsize*. In the following sections, a *Mostly-Serial Loop* is one with (*Hsize, Hfrac*) = (10%, 90%), a *Mixed Loop* is one with (*Hsize, Hfrac*) = (50%, 50%) and a *Mostly-Parallel Loop* is one with (*Hsize, Hfrac*) = (90%, 10%).

Param.	Explanation	What We Want to Measure	Values
N	Iteration Count	Problem size scalability	1600, 3200, 6400, 12800,25600
Hfrac	Hot spot fraction	Consistency over dependence patterns	10%, 50%, 90%
Hsize	Hot spot size		0.1%, 10%, 50%, 90%
W	Iteration grain size	Consistency over grain size	40μs (1320 cycles), 160μs (5281 cycles), 640μs (21122 cycles)
r	# references per iteration	Consistency over number of accesses	1, 2, 4, 8

Table 3: Loop parameters used in the experiments. The cycles in column 4 are 30 *ns* processor cycles.

/ There are N iterations and r accesses per iteration. INDEX contains the actual array references. tmp1 and tmp2 are dummy variables. */*

```
do i = 1,N,1
      do j = 1,r,1
            if(odd(j)) tmp1 = A[INDEX[i*r+j]];
            else A[INDEX[i*r+j]] = tmp2;
            do j = 1,W,1
                  dummy loop simulating useful work
            enddo
      enddo
enddo
```

Figure 8: Loops used as experimental workload.

4.2 Evaluation

In this section, we analyze our experimental results. First, we show the speedup of our scheme over serial execution. Then we compare our scheme with Chen *et al.*'s scheme. A *pre-scheduled* form of Chen *et al.*'s executor and a *self-scheduled* form of our executor with *barrier* synchronization is used for the comparisons. In Chen *et al.*'s scheme both *pre-scheduled* executor and *self-scheduled* executor give very similar performance.

4.3 Performance comparison with serial execution

To determine the speedup of our algorithm, we consider two situations: one in which it performs both inspection and execution (Table 4) and another in which it only performs the execution (Table 5). The latter corresponds to the case where we reuse the wavefront information gathered during previous inspection of the loop. We use 8 processors in both the experiments, and vary the number of iterations, the iteration grain size and the number of references per iteration. Overall, we obtain good speedups: up to 7 when inspector and executor are considered, and up to 8 when only the executor is considered. Obviously, the best results occur when the size of the loop body (W) is large and the number of accesses (r) is low. Table 4 and Table 5 also show that, as expected, the performance is better when the dependence chains are short; that is why the performance of the *Mostly-Parallel Loop* is better than the other two loops.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.20	1.33	1.30	1.44	1.29
	4	1.45	1.78	1.67	2.06	1.86
	2	1.33	1.94	2.52	2.40	2.63
	1	2.14	2.31	2.77	2.67	2.88
160 μs	8	3.00	3.18	3.13	3.49	3.47
	4	3.60	3.81	4.14	4.41	4.36
	2	3.81	4.44	4.75	4.75	4.98
	1	4.29	4.65	5.24	5.22	5.44
640 μs	8	5.02	5.43	5.75	5.79	5.85
	4	5.74	6.01	5.34	6.50	6.46
	2	5.83	6.30	6.53	6.81	6.91
	1	6.13	6.39	6.75	6.96	7.09

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.29	1.38	1.48	1.25	1.21
	4	1.78	1.68	1.76	1.81	1.91
	2	2.14	2.07	2.33	2.31	2.48
	1	2.14	2.50	2.65	2.86	2.82
160 μs	8	3.15	3.02	3.38	3.42	3.15
	4	3.81	4.21	4.28	4.18	4.16
	2	4.36	4.62	4.84	4.94	5.01
	1	5.00	5.50	5.24	5.45	5.50
640 μs	8	5.74	5.70	5.78	5.79	5.67
	4	6.18	6.17	6.45	6.48	6.56
	2	6.64	6.75	6.81	6.89	7.00
	1	6.46	6.94	7.10	7.03	7.19

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%) N=1600	3200	6400	12800	25600
40 μs	8	1.21	1.25	1.40	1.44	1.43
	4	1.89	2.00	1.91	2.02	2.05
	2	2.30	2.13	2.52	2.60	2.54
	1	3.00	3.10	2.65	2.71	2.72
160 μs	8	3.26	3.05	3.25	3.16	3.02
	4	4.36	4.36	4.44	4.33	4.22
	2	5.00	4.62	5.04	5.03	5.11
	1	5.00	5.22	5.58	5.52	5.60
640 μs	8	5.61	5.95	5.85	5.89	5.96
	4	6.49	6.58	6.54	6.51	6.53
	2	6.64	6.73	6.89	6.89	6.95
	1	7.03	6.94	7.10	7.12	7.17

(c)

Table 4: Speedup of our algorithm over serial execution including *inspector* and *executor* using 8 processors.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%) N=1600	3200	6400	12800	25600
40 μs	8	3.60	5.14	5.46	5.96	6.50
	4	4.00	4.00	5.42	6.19	6.44
	2	3.75	4.43	5.73	5.95	6.41
	1	3.75	4.30	5.55	6.15	6.28
160 μs	8	5.73	6.53	6.94	7.30	7.43
	4	5.55	6.42	6.97	7.30	7.50
	2	6.10	6.32	6.91	7.22	7.49
	1	6.00	6.40	7.09	7.27	7.52
640 μs	8	6.70	7.00	7.37	7.62	7.78
	4	6.89	7.07	6.24	7.63	7.76
	2	6.64	7.04	7.38	7.60	7.79
	1	6.83	7.04	7.43	7.62	7.80

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	4.50	6.00	6.36	6.22	6.36
	4	5.33	6.40	6.50	6.50	6.60
	2	7.50	6.20	6.30	5.68	6.58
	1	5.00	6.00	6.78	6.83	6.81
160 μs	8	7.00	6.89	7.58	7.51	7.43
	4	6.78	7.63	7.18	7.52	7.61
	2	6.78	7.06	7.33	7.56	7.61
	1	7.50	7.56	7.53	7.62	7.70
640 μs	8	7.53	7.67	7.73	7.80	7.81
	4	7.53	7.63	7.75	7.82	7.84
	2	7.47	7.73	7.80	7.82	7.86
	1	7.47	7.60	7.85	7.84	7.87

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	5.67	5.83	6.36	6.81	6.50
	4	5.67	6.40	6.50	6.55	6.72
	2	5.33	6.40	6.30	6.58	6.78
	1	7.50	7.75	6.78	6.42	7.17
160 μs	8	7.75	7.35	7.35	7.58	7.60
	4	7.63	7.63	7.40	7.64	7.66
	2	7.50	7.06	7.56	7.55	7.64
	1	6.67	7.50	7.74	7.62	7.70
640 μs	8	7.53	7.65	7.78	7.88	7.88
	4	7.74	7.74	7.81	7.87	7.87
	2	7.71	7.71	7.84	7.85	7.89
	1	7.71	7.73	7.80	7.88	7.90

(c)

Table 5: Speedup of our algorithm over serial execution including *executor only* using 8 processors.

In summary, the results show that run-time parallelization is a viable approach to speed up loops that cannot be handled by compile-time parallelization. This approach gives good results when (a) both the inspector and executor are considered

(b) executor only is considered. There is no big leap in performance when the inspector results are reused because the inspector stage is very efficient and fast. To see how the speedup varies with the number of processors, we also ran several experiments with the three different types of loops using various number of processors. With up to 8 processors, we obtain almost linear speedups using (1) both inspector and executor and (2) executor only. We expect the speedup to be scalable with a larger numbers of processors.

4.4 Performance comparison with Chen *et al.*'s scheme

Table 6 gives the speedup of our scheme over Chen *et al.*'s when both inspector and executor are considered. It indicates superior performance of our algorithm in this case. It is also clear that the performance of our approach does not degrade rapidly as the DOACROSS loop structure becomes more and more parallel in nature. This is largely because our inspector is much faster and efficient, without requiring special synchronization operations and uses very little inter-processor communication.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
40 μs	8	4.60	13.85	37.40	110.56	251.99
	4	1.82	3.83	10.08	32.67	76.97
	2	1.11	1.50	3.28	7.75	21.80
	1	1.00	1.00	1.45	2.07	5.04
160 μs	8	3.71	10.21	26.11	76.44	194.45
	4	1.82	2.88	7.17	19.00	49.40
	2	1.25	1.67	2.41	4.70	11.44
	1	1.14	1.27	1.52	1.87	3.27
640 μs	8	2.46	5.27	13.18	33.94	86.60
	4	1.62	2.14	3.71	8.17	19.83
	2	1.44	1.62	1.90	2.84	5.12
	1	1.41	1.47	1.58	1.77	2.25

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	23.71	98.31	348.4	744.25	1554.58
	4	6.78	17.58	69.22	232.65	614.60
	2	2.86	4.20	12.67	48.04	165.73
	1	1.43	1.75	2.96	8.14	29.93
160 μs	8	16.55	62.61	226.22	576.05	1153.98
	4	4.13	11.86	44.82	142.49	359.50
	2	1.86	2.85	7.22	26.74	86.73
	1	1.33	1.55	1.98	4.38	15.29
640 μs	8	8.62	30.39	100.75	255.22	538.18
	4	2.31	5.09	17.85	57.16	145.60
	2	1.33	1.68	3.23	9.93	31.27
	1	1.03	1.14	1.35	2.05	5.68

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	24.50	95.36	348.70	1984.33	1898.82
	4	6.78	21.50	80.24	349.65	690.24
	2	2.86	4.13	14.00	56.69	177.16
	1	2.20	2.30	2.91	7.98	30.82
160 μs	8	18.47	66.63	229.27	560.09	1142.25
	4	4.79	12.93	49.29	156.82	379.82
	2	2.17	2.88	7.88	29.42	92.64
	1	1.25	1.43	2.07	4.74	16.39
640 μs	8	8.77	33.67	107.56	268.71	584.97
	4	2.41	5.53	19.20	60.29	151.33
	2	1.36	1.69	3.35	10.86	32.70
	1	1.15	1.14	1.33	2.19	5.90

(c)

Table 6: Speedup of our algorithm over Chen et al.'s scheme including *inspector* and *executor* using 8 processors.

Table 7 considers the executor only, which is important if the inspector results are reused across loop invocations. Further, it shows that our executor gives better performance for the *Mostly-Serial* and *Mixed* Loops. This is because our scheme has better iteration space to processor mapping during the executor stage whereas their executor spends processor cycles *busy-waiting*. In the case of *Mostly-Parallel* loops, both the schemes show nearly identical results because there is not much *busy-waiting* overhead involved.

W	r	**Mostly-Serial Loop** ($Hsize$ = 10%, $Hfrac$ = 90%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.43	1.54	1.63	1.75
	4	1.25	1.00	1.42	1.62	1.66
	2	1.25	1.14	1.45	1.57	1.64
	1	1.00	1.14	1.36	1.50	1.54
160 μs	8	1.36	1.58	1.61	1.68	1.73
	4	1.45	1.47	1.60	1.67	1.71
	2	1.40	1.47	1.57	1.64	1.70
	1	1.40	1.47	1.59	1.65	1.70
640 μs	8	1.50	1.55	1.63	1.68	1.71
	4	1.51	1.56	1.62	1.69	1.71
	2	1.47	1.56	1.62	1.72	1.71
	1	1.51	1.54	1.62	1.67	1.72

(a)

W	r	**Mixed Loop** ($Hsize$ = 50%, $Hfrac$ = 50%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.67	1.82	1.78	1.84
	4	1.00	1.20	1.30	1.40	1.43
	2	1.00	1.00	1.10	1.05	1.21
	1	0.67	0.80	1.11	1.06	1.08
160 μs	8	1.00	1.06	1.18	1.15	1.16
	4	0.89	1.06	1.03	1.06	1.09
	2	0.89	0.94	0.97	1.03	1.04
	1	1.00	1.00	1.00	1.00	1.02
640 μs	8	1.00	1.02	1.02	1.03	1.03
	4	1.00	0.98	1.00	1.01	1.02
	2	0.94	0.98	0.99	1.00	1.01
	1	0.94	0.97	0.99	1.00	1.00

(b)

W	r	**Mostly-Parallel Loop** ($Hsize$ = 90%, $Hfrac$ = 10%)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.00	1.09	1.33	1.11
	4	1.00	1.00	1.00	1.15	1.08
	2	0.67	0.80	0.90	1.00	1.03
	1	1.50	1.25	1.00	0.95	1.03
160 μs	8	1.13	1.06	1.03	1.03	1.03
	4	1.00	1.00	1.00	1.02	1.02
	2	1.00	0.94	1.00	1.00	1.00
	1	0.89	0.94	1.00	1.00	1.01
640 μs	8	1.00	1.00	1.01	1.01	1.01
	4	1.00	0.98	1.01	1.01	1.00
	2	1.00	0.98	1.00	1.00	1.01
	1	1.00	0.98	0.98	1.00	1.01

(c)

Table 7: Speedup of our algorithm over Chen et al.'s scheme including *executor only* using 8 processors.

In Figure 9 (a & b), the sensitivity of the inspector (both schemes) to the different types of loops is shown. The columns are normalized to the *Mostly-Serial* case. Results show that our inspector is less sensitive to differences in loop types and gives consistent performance compared to Chen *et al.*'s scheme.

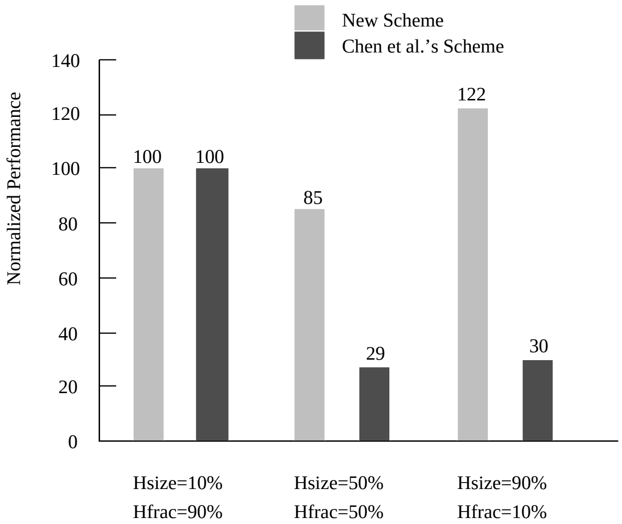

Figure 9(a): The sensitivity of the *inspector* to different types of loops for iteration count (N) = 1600 and number of acceses per iteration (r) = 1.

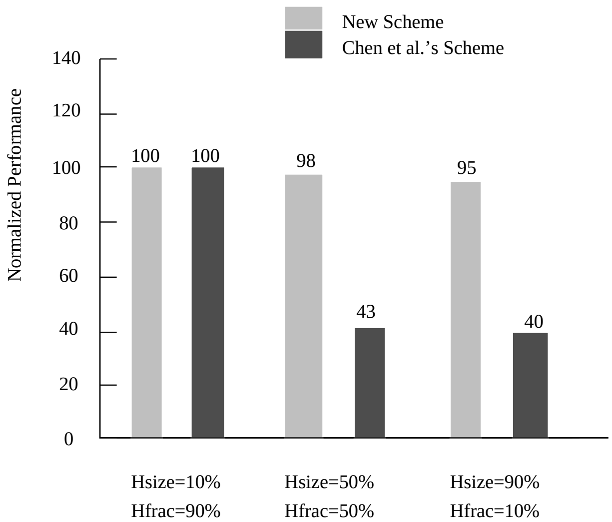

Figure 9(b): The sensitivity of the *inspector* to different types of loops for iteration count (N) = 25600 and number of acceses per iteration (r) = 1.

In Figure 10 (a & b), the sensitivity of the inspector (both schemes) to the number of references per iteration is shown. The columns are normalized to $r = 1$ case. Results show that our inspector gives much better performance compared to Chen *et al.*'s scheme even though it is sensitive to the number of references per iteration.

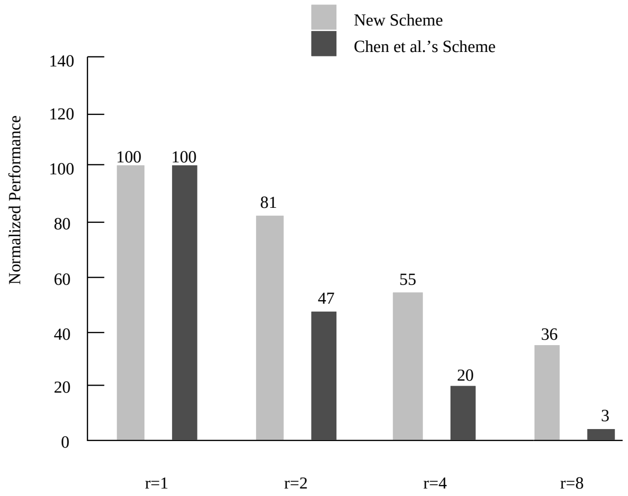

Figure 10(a): The sensitivity of the *inspector* to number of references per iteration for iteration count $(N) = 1600$.

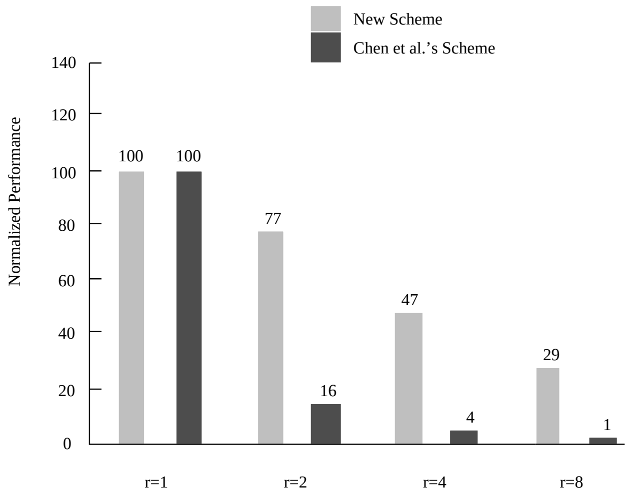

Figure 10(b): The sensitivity of the *inspector* to number of references per interation for iteration count $(N) = 25600$.

In Figure 11 (a & b), the sensitivity of the executor (both schemes) to the different types of loops is shown. The columns are normalized to the *Mostly-Parallel* case. Results show that our executor is less sensitive to differences in loop types compared to Chen *et al.*'s scheme.

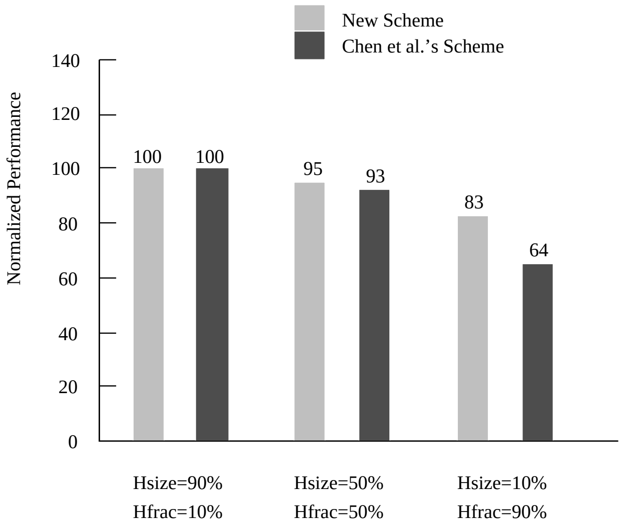

Figure 11(a): The sensitivity of the *executor* to different types of loops for iteration count $(N) = 1600$ and number of acceses per iteration $(r) = 1$.

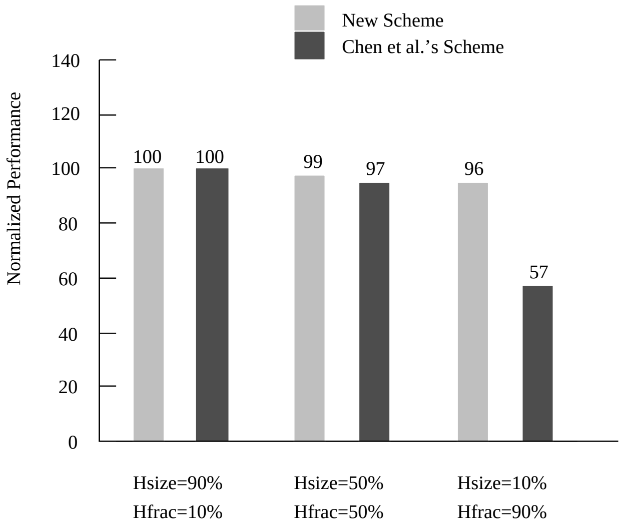

Figure 11(b): The sensitivity of the *executor* to different types of loops for iteration count $(N) = 25600$ and number of acceses per iteration $(r) = 1$.

In Figure 12 (a & b), the sensitivity of the executor (both schemes) to the number of references per iteration is shown. The columns are normalized to $r = 1$ case. Results show that both the schemes are not very sensitive to the number of references per iteration. Still our scheme has relatively better performance compared to Chen *et al.*'s scheme.

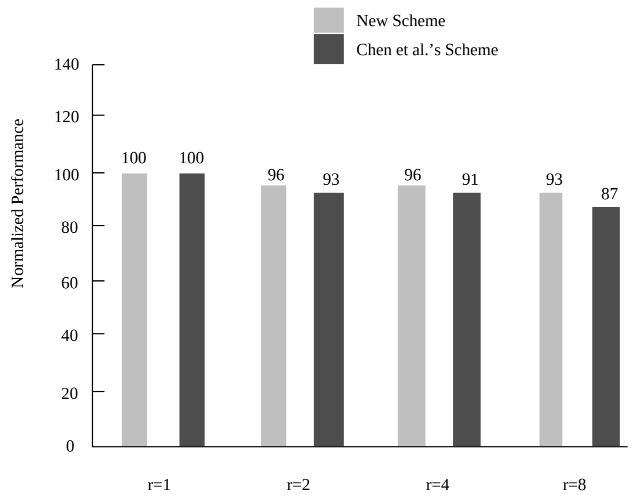

Figure 12(a): The sensitivity of the *executor* to number of references per iteration for iteration count $(N) = 1600$.

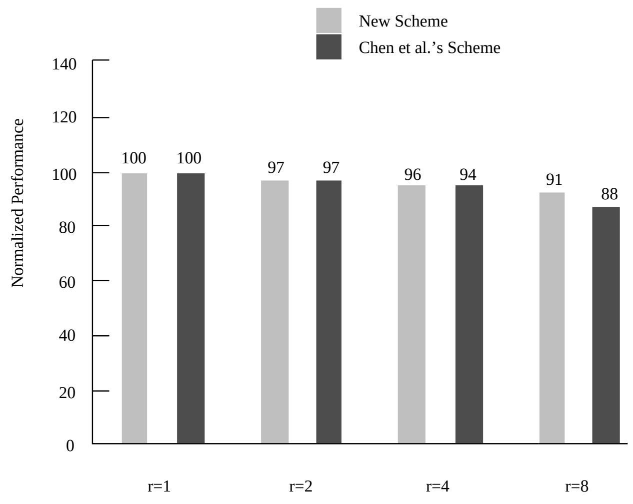

Figure 12(b): The sensitivity of the *executor* to number of references per iteration for iteration count $(N) = 25600$.

To summarize,

- Our inspector gives consistent performance for all the different types of loops whereas Chen *et al.*'s inspector lacks in consistency (i.e, performance degrades as the DOACROSS loop structure becomes more and more parallel in nature).
- During the executor stage, the new scheme ensures better iteration space to processor mapping thereby achieving better load balance and overall speedup.

4.5 Performance of the executor: *Barrier synchronization vs. Busy-wait*

Table 8 compares the speedups of the two different types of the executor in the new scheme over Chen *et al.*'s executor. The results show that the executor (*busy-wait*) gives better performance over the executor (*barrier synchronization*) for *Mostly-Serial* loops. From Table 9 it is evident that for *Mostly-Parallel* loops the performance of the executor (*busy-wait*) is poor compared to the executor (*barrier synchronization*). The results include the run-time overhead of setting up the *sequence table* in the case of executor (*busy-wait*).

W	r	**Executor (New scheme)** (*Barrier synchronization*)				
		N=1600	3200	6400	12800	25600
40 μs	8	0.52	0.39	0.47	0.60	1.08
	4	0.37	0.49	0.42	0.55	0.98
	2	0.39	0.35	0.47	0.51	0.94
	1	0.28	0.32	0.35	0.42	0.88
160 μs	8	0.95	0.71	0.85	1.09	1.99
	4	0.72	0.97	0.78	1.07	1.93
	2	0.66	0.73	0.98	1.03	1.92
	1	0.61	0.66	0.72	0.97	1.85
640 μs	8	1.32	0.96	1.14	1.47	2.74
	4	1.00	1.35	1.09	1.47	2.74
	2	0.92	1.00	1.36	1.44	2.72
	1	0.85	0.93	1.01	1.36	2.61

(a)

W	r	**Executor (New scheme)** ($Busy\text{-}wait$)				
		N=1600	3200	6400	12800	25600
40 μs	8	0.89	0.79	0.88	1.03	1.42
	4	0.94	1.10	0.99	1.25	1.80
	2	1.13	1.00	1.30	1.37	2.19
	1	0.94	1.03	1.07	1.39	2.37
160 μs	8	1.30	0.98	1.15	1.45	2.36
	4	1.12	1.44	1.18	1.55	2.68
	2	1.06	1.15	1.53	1.59	2.89
	1	0.98	1.07	1.16	1.55	2.91
640 μs	8	1.52	1.08	1.28	1.63	2.96
	4	1.17	1.57	1.25	1.68	3.10
	2	1.08	1.18	1.60	1.67	3.17
	1	0.99	1.08	1.19	1.60	3.07

(b)

Table 8: Comparison of the speedup of the two different types of executors in the new scheme over Chen et al.'s executor. The results correspond to a Mostly-Serial Loop ($Hsize$ = 0.1%, $Hfrac$ = 90%).

W	r	**Executor (New scheme)** ($Barrier\ synchronization$)				
		N=1600	3200	6400	12800	25600
40 μs	8	1.00	1.00	1.09	1.33	1.11
	4	1.00	1.00	1.00	1.15	1.08
	2	0.67	0.80	0.90	1.00	1.03
	1	1.50	1.25	1.00	0.95	1.03
160 μs	8	1.13	1.06	1.03	1.03	1.03
	4	1.00	1.00	1.00	1.02	1.02
	2	1.00	0.94	1.00	1.00	1.00
	1	0.89	0.94	1.00	1.00	1.01
640 μs	8	1.00	1.00	1.01	1.01	1.01
	4	1.00	0.98	1.01	1.01	1.00
	2	1.00	0.98	1.00	1.00	1.01
	1	1.00	0.98	0.98	1.00	1.01

(a)

W	r	**Executor (New scheme)** (*Busy-wait*)				
		N=1600	3200	6400	12800	25600
40 μs	8	0.25	0.26	0.25	0.29	0.26
	4	0.43	0.36	0.34	0.48	0.36
	2	0.40	0.40	0.45	0.48	0.48
	1	1.00	0.71	0.60	0.60	0.59
160 μs	8	0.53	0.53	0.52	0.50	0.50
	4	0.67	0.64	0.66	0.64	0.65
	2	0.80	0.80	0.78	0.77	0.76
	1	1.00	0.83	0.84	0.85	0.84
640 μs	8	0.84	0.83	0.82	0.82	0.81
	4	0.91	0.90	0.90	0.89	0.89
	2	0.97	0.94	0.95	0.94	0.94
	1	0.97	0.97	0.96	0.96	0.98

(b)

Table 9: Comparison of the speedup of the two different types of executors in the new scheme over Chen et al.'s executor. The results correspond to a Mostly-Parallel Loop(*Hsize* = 90%, *Hfrac* = 10%).

In the *Mostly-Serial* loops, there are a lot more chances for overlap among dependent iterations (since there will be a number of dependent iterations), whereas in the case of *Mostly-Parallel* loops there are fewer dependent iterations, so the chances for overlapped execution of dependent iterations is much less. Thus, it is quite obvious that the executor (*busy-wait*) does not give better performance when compared to the executor (*barrier synchronization*) in the *Mostly-Parallel* case.

5 Conclusions

The automatic parallelization of loops is usually based on compile-time analysis of data dependencies. In some loops, however, the data dependencies cannot be determined at compile time. An example is loops accessing arrays with subscripted subscripts. To parallelize these loops, run-time analysis is needed. In this paper, we presented and evaluated a new algorithm to parallelize these loops at run-time. Our scheme handles any type of data dependence pattern without requiring any special architectural support. It minimizes inter-processor communication, leads to better processor utilization and allows reuse of the inspector results across several loop invocations. Comparison of the new algorithm with an older scheme with the same generality shows that the new scheme has a consistent performance (i.e., performance does not degrade rapidly with the number of iterations or accesses per iteration) during the inspector stage and leads to higher speedups during the executor stage.

We have evaluated our algorithm with a set of parametrized loops running on a 8-processor SGI shared memory multiprocessor. We used loops with varying parameters, such as the number of iterations and references. The results show better speedups and they are scalable with higher number of processors.

Acknowledgements

The authors would like to thank Dr. Derek L. Eager, Professor, Department of Computer Science, University of Saskatchewan, Saskatoon, SK S7N OWO, Canada and Dr. P. Sadayappan, Professor, Department of Computer and Information Science, Ohio State University, Columbus, OH 43210 for their help in the experiments and the anonymous referees for their helpful comments.

References

[1] U. Banerjee, *Dependence analysis for supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[2] D.-K. Chen, J. Torrellas and P.-C. Yew, *An efficient algorithm for the run-time parallelization of DOACROSS Loops*, Supercomputing 1994.

[3] D.-K. Chen and P.-C. Yew, *A scheme for effective execution of irregular DOACROSS loops*, in Int'l Conf. on Parallel Processing, pages 285-292, August 1992.

[4] V. P. Krothapalli, *On synchronization and scheduling for multiprocessors*, Ph.D thesis, The Ohio State University 1990.

[5] V. P. Krothapalli and P. Sadayappan, *An approach to synchronization for parallel computing*, in Proc. 1988 Conf. on Supercomputing, St. Malo, 1988.

[6] V. P. Krothapalli and P. Sadayappan, *Dynamic scheduling of DOACROSS loops for multiprocessors*, IEEE Parallel Architectures 1991.

[7] D. Kuck et al, *The Cedar system and an initial performance study*, in 20th Int'l Symp. on Computer Architecture, May 1993.

[8] S.-T. Leung and J. Zahorjan, *Improving the performance of run-time parallelization*, in 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pages 83-91, May 1993.

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

[10] D. A. Padua, *Multiprocessors: Discussion of some theoretical and practical problems*, Ph.D thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1979.

[11] C. Polychronopoulos, *Advanced loop optimizations for parallel computers*, In E. Houstis, T. Papatheodorou, and C. Polychronopoulos, editors, *Lecture Notes in Computer Science No. 297: Proc. First Int'l Conf. on Supercomputing, Athens, Greece*, pages 255-277, New York, June 1987, Springer-Verlag.

[12] J. Saltz, R. Mirchandaney, and K. Crowley, *Run-time parallelization and scheduling of loops*, IEEE Trans. Computers, 40(5):603-612, May 1991.

[13] J. Thulasiraman, V. P. Krothapalli and M. Giesbrecht, *Run-time parallelization of irregular DOACROSS loops*, June 5, 1995, Technical Report No. 95/03, Department of Computer Science, University of Manitoba, Winnipeg, Canada.

[14] T. H. Tzen and L. Ni, *Dependence uniformization: A loop parallelization technique*, IEEE Trans. on Parallel and Distributed Systems, 4(5), May 1993.

[15] M. J. Wolfe, *Optimizing compilers for supercomputers*, Ph.D thesis, University of Illinois at Urbana-Champaign, 1982.

[16] M. J. Wolfe, *Loop skewing: The wavefront method revisited*, Int. J. of Parallel Programming, 4(15), 1986.

[17] C.-Q.Zhu and P.-C.Yew, *A scheme to enforce data dependence on large multiprocessor systems*, IEEE Trans. on Software Engineering, June 1987.