CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 7(5), 349–369 (AUGUST 1995)

Compile-time scheduling of multithread with data localities on multiple vector processors

CHIH-YUNG CHANG AND JANG-PING SHEU

Department of Computer Science and Information Engineering
National Central University, Chung-Li 32054, Taiwan

SUMMARY

A large class of loop programs applied in solving differential equations, Fourier transforms, image processing and neural processing can be translated or rewritten into a vector execution form with a *π-block* dependence graph. In the paper we propose a multithreading strategy to partition such vectorized loops into multithread execution form. Each partitioned thread consists of instances of statements with localities in vector registers. The multithreading scheme gives a novel combination of *loop unrolling*, *statement instances reordering*, *index shifting*, *vector register reuse exploiting* and *multithreading*. For some cases of loop program with *π-block* dependence graph, experimental results show that our scheme assists vector compilers of the Convex C38 series to reduce the number of memory accesses and synchronizations among CPUs.

1. INTRODUCTION

Recent vector computers are equipped with several vector processors and a hierarchical memory to offer both vectorization and multiprocessing capabilities. In vector processing, several attempts have been made at discovering vector operations from sequential loop programs to meet the capabilities of vector functional units in vector machines. Considerable effort [1–3] has gone into the development of exploiting parallelism from sequential loops and then transforming them into DOALL or wavefront execution form. This also makes it possible that the transformed loop can be further vectorized for a single CPU. Most of the developments in parallelism extraction treated iteration as the minimum parallelism unit. To exploit maximum parallelism for a given loop program, recently, several researchers have treated a computation of statement instance as the minimum parallelism unit[2,4,5]. For these available vectorized programs, proper partition of the vector operations for parallel processing can significantly utilize the hardware design of multiple vector processors and reduce the execution time. In parallel processing, several vector processors can work together to concurrently perform tasks which consist of scalar or vector operations defined in the program. Synchronizations are needed among these processors if their references of array data have a dependence relation. To minimize the execution time of the vectorized loop program, it is important to carefully partition the vector operations into a number of sets such that programs can be executed in high parallelism and with few synchronizations.

The factors that determine the system performance of vector computers are not only parallelism but also the memory management. A comprehensive study[6–8] aims at reducing the number of data movements between different memory hierarchies. If compilers are capable of exploiting the opportunities of vector data reuse, the execution time of a loop program can be significantly improved[6]. However, discussions of reuse exploitation in vector computers focus mainly on a single CPU. Recently, Irigoin and Triolet proposed

CCC 1040–3108/95/050349–21

Received 8 October 1994
Revised 21 April 1995

a supernode partitioning technique[9] to vectorize and parallelize the sequential program according to the data dependence relation. Multiple CPUs can thus concurrently execute vector operations in the manner of data reuse exploitation. However, two more aspects need further attention to enhance the system performance. Firstly, there is one dependence vector to exploit the reuse opportunity. The degree of reuse exploitation may be further improved if the given loop program has a complex dependence relation. Secondly, the supernode partitioning treats an iteration as the minimum parallelism unit. Statement instance partitioning may extract more parallelism from loops with a π-*block* dependence graph. We are motivated to design a systematic strategy to flexibly partition the vector operations into multithread such that the number of synchronizations and the degree of data reuse exploitation can be improved.

The approach adopted here automatically collects instances of vector statements that have common data references into one thread. Two advantages can be obtained in our multithreading scheme. Firstly, the common referenced vector data can be preserved in a vector register for reuse in the successive running iterations. The time cost for vector loads is thus reduced. Secondly, a large number of synchronizations can be saved. Comparisons are made to illustrate that, with the assistance of our multithreading scheme, the transformed loop program has better performance.

The rest of this paper is organized as follows. The primary concept of the multithreading approach is introduced in Section 2. The systematic multithreading technique is addressed in Section 3. The performance analysis of our multithreading method is discussed in Section 4. The conclusions are given in Section 5.

2. LOOP MODEL AND BASIC CONCEPT

In this Section we first introduce the definition of the *dependence graph*. Some assumptions are made and two compiler directives are introduced. The multithread versions generated by the current vector compiler of Convex C3840 and the proposed multithreading scheme are then discussed by an example.

Definition: Data dependence graph (also appeared in [10])
A *data dependence graph* $DG(N,E)$ or DG of an n-nested sequential loop or an $(n-1)$-nested vectorized loop L consists of a set of nodes N and a set of directed edges E. The node labeled S denotes a statement S in loop L, while an edge labeled by a *dependence vector* $\vec{d}=(d_1,d_2,\ldots,d_n)$ links from node S to node S' if array data are first referenced by S in iteration $\vec{i}$ and then referenced again by S' in iteration $\vec{j}$, where $(d_1,d_2,\ldots,d_n)=\vec{j}-\vec{i}$.

There are four types of dependence vectors, the *input*, *output*, *true dependencies* and the *antidependence*, possibly existing in loop L. Traditional vectorization techniques[10] first analyze the data dependence relation and then decompose the DG into several π-*blocks*; each of them is a strongly connected component. Loops without a π-*block* DG are easy to deal with in parallelism extraction, synchronization reduction and reuse exploitation. Here we focus our attention on loops with a π-*block* DG. There are very few reuse opportunities if references are not uniformly generated. In this paper, only constant true dependence and input dependence which significantly affect the extraction of reuse and parallelism are considered. The loop L considered as our input source is formulated as the following $(n-1)$-nested vectorized loop program with s statements:

```
DO   I2 = l2,u2
     :
     DO   In = ln,un
          vectorized statements          (L)
     ENDDO
     :
ENDDO
```

where l_i and u_i are, respectively, the lower and upper bounds of index variables I_i, for $2 \leq i \leq n$. The *DG* of loop L is a *π-block*. A large class of application algorithms such as solving differential equations, Fourier transforms, image processing and neural processing falls in this model.

Several vector compilers offer compiler directives for the programmer to manually perform the loop optimization. However, improper use of these directives can cause not only semantic errors but also inefficient execution. Unless users are skilled in parallel program design, it is difficult to write an efficient program with explicit definitions of multithread. In this paper the proposed multithreading scheme automatically restructures the loop program and inserts the proper compiler directive to specify the multithread. Two compiler directives related to multithread identification are introduced as follows. Instructions introduced here are recognizable to the compiler of the Convex C38 series. Similar instructions can also be found in compilers of other vector machines such as the Cray series[11].

Let Force_Parallel be the compiler directive statement that informs the compiler to parallelize the loop that follows, regardless of the apparent dependencies between iterations[12]. As an example, consider the following program:

```
DO   I1 = 1,n
     C$DIR Force_Parallel
     DO   I2 = 1, 4
          vectorized statements
     ENDDO
ENDDO
```

The Force_Parallel directive will assist the compiler to parallelize the execution of four instances I_2=1, I_2=2, I_2=3 and I_2=4 of vector statements on four CPUs. Since loop I_1 is a sequential loop, synchronizations are needed among these CPUs to guarantee the sequential execution of loop I_1. In this example there are n synchronizations needed among four CPUs. A loop program instructed by Force_Parallel can be performed by several CPUs in such a manner that all CPUs concurrently perform the same loop body with different instances. The execution form is referred to as *SBDI* (same body different instances).

Another compiler directive can also be used to define multithread. Let the compiler directive

```
C$DIR Begin_Tasks
    {statement group 1}
C$DIR Next_Task
    {statement group 2}
        ⋮
C$DIR Next_Task
    {statement group t}
C$DIR End_Tasks
```

instruct the compilers to parallelize t threads (or tasks). If the Begin_Tasks ... End_Tasks directive appears in the loop body of a loop program, the execution form is referred to as *DBSI* (different body same instance) since the multithread is defined by running the same instance of the loop on different statement groups.

To illustrate our multithread identification approach in detail, we use the following artificial example throughout this paper. Performance improvements of several practical application programs are shown in Section 4. Consider the following vectorized loop:

```
DO   J = 6, 645
  S1:  A(2:129,J)=B(1:128,J − 3) * E(2:129,J)
  S2:  B(2:129,J)=A(2:129,J − 2) + C(2:129,J − 5) − E(2:129,J − 2)      (L1)
  S3:  C(2:129,J)=B(2:129,J − 4) * 3
ENDDO
```

$\bar{d}_1 = (0,2)$ for variables $A(2{:}129,J)$ and $A(2{:}129,J-2)$
$\bar{d}_2 = (0,2)$ for variables $E(2{:}129,J)$ and $E(2{:}129,J-2)$
$\bar{d}_3 = (1,3)$ for variables $B(2{:}129,J)$ and $B(1{:}128,J-3)$
$\bar{d}_4 = (0,4)$ for variables $B(2{:}129,J)$ and $B(2{:}129,J-4)$
$\bar{d}_5 = (0,5)$ for variables $C(2{:}129,J)$ and $C(2{:}129,J-5)$

as shown in Figure 1, where the input dependence $\bar{d}_2$ is denoted by a dotted line and the true dependencies are denoted by solid lines. Although there exists an input dependence $(1, -1)$ of array B between statements S_1 and S_3, data reuse opportunities caused by this dependence relation can be exploited by considering the $\bar{d}_3$ and $\bar{d}_4$ since $(1, -1)$ is a linear combination of $\bar{d}_3$ and $\bar{d}_4$ for variable B. Thus this input dependence is redundant in reuse exploitation and can be ignored.

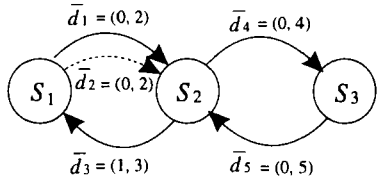


Figure 1. DG of loop L1

Assume that there are four CPUs supported by the system. The compiler of Convex C3840 will parallelize loop I and sequentially execute loop J. The assembly code compiled by Convex C3840 is equivalent to the following code:

```
DO   J = 6, 645
   C$DIR Force_Parallel
   DO  2  I = 2, 129, 32
      S1: A(I:I + 31,J)=B(I − 1:I + 30,J − 3) * E(I:I + 31,J)
      S2: B(I:I + 31,J)=A(I:I + 31,J − 2) + C(I:I + 31,J − 5) − E(I:I + 31,J − 2)   (L1')
      S3: C(I:I + 31,J)=B(I:I + 31,J − 4) * 3
   ENDDO
ENDDO
```

The compiler partitions the vector length of 128 into four subsets and then vectorizes each subset in a vector length of 32. When executing a particular instance of J, there exists no dependence in four subsets; the four threads (instances I=2,I=34,I=66 and I=98 of the loop body) can be concurrently executed on four CPUs. Because loop J is a sequential loop, synchronization should be made at each running iteration of loop J.

Two drawbacks are found in multithread version $L1'$:

1. The number of synchronizations is equal to 645 − 6 + 1=640, which can be reduced by the proposed technique introduced in this paper.
2. No data reuse is exploited in loop $L1'$. Consider the execution of loop $L1$. Array elements A(2:129,6) generated by execution of statement S_1 at J=6 are referenced again by the execution of statement S_2 at J=8. Decreasing the reuse distance will exploit the reuse opportunities. However, in loop $L1'$, the reuse distance is 2. The generated array data will be swept out at J=6 and then be loaded again from shared memory to vector register at J=8. The number of the vector loads in $L1'$ is the same as that in $L1$.

Instead, loop $L1$ can be transformed into another, better, version, the *DBSI* execution form, to exploit the reuse opportunities and reduce the number of synchronizations.

We first introduce the definitions of *SIDG* and *parallel block*. The *DBSI* method is then introduced by the example of loop $L1$.

Definition: Statement instance dependence graph (*SIDG*)

A *statement instance dependence graph*, denoted by *SIDG*(V,E) or *SIDG*, of loop L consists of a set of vertices V and a set of directed edges E. Each vertex in set V represents an instance of a vectorized statement in L and a directed edge labeled by $\bar{d}$ connects two vertices from v_i to v_j if there is an edge labeled by $\bar{d}$ linking from S_i to S_j in *DG*, where v_i and v_j are the respective instances of S_i and S_j.

Figure 2 displays the *SIDG* of $L1$. The *SIDG* can be viewed as the extension of the *DG* from iteration level to statement instance level. In *SIDG*, instances without data dependence can be executed simultaneously. In what follows, we give the definition of a *parallel block* to denote the union of these instances.

Definition: Parallel block

The *SIDG* can be partitioned into several disjoint parallel blocks. The first parallel block of *SIDG* consists of instance vertices that are not dependent on any other vertex. The $(i + 1)$th parallel block is the maximum collection of all instances that can be parallel performed after the execution of the ith parallel block.

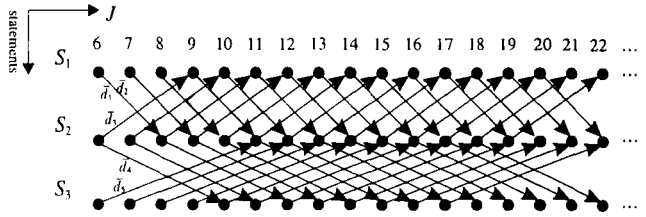


Figure 2. SIDG of loop L1

Figure 3(a) displays the parallel blocks of *SIDG* of *L*1. In the next Section we illustrate how to derive the parallel blocks. In Figure 3(a) the first and the last parallel blocks are irregular. For simple discussion of the *DBSI* method, we ignore these two irregular blocks. When transforming a program into *DBSI* execution form, instances within these two irregular blocks can be unrolled to meet our assumption.

Two types of parallel blocks can be found in Figure 3(a). Seven and eight parallel instances can be concurrently executed in type one and type two parallel blocks, respectively.

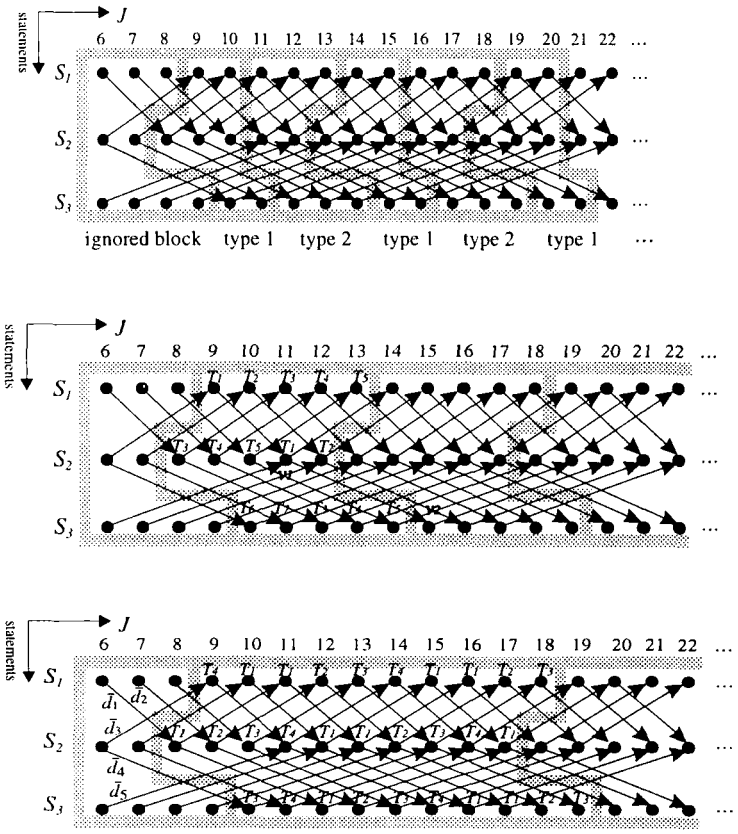


Figure 3. Parallel blocks and the execution blocks of loop L1; (a) Two types of parallel blocks in L1; (b) Combining two parallel blocks into an execution block; (c) Combining four parallel blocks into an execution block

In a parallel block, too many independent instances have no benefit to parallelism since a vector computer lacks enough CPUs to concurrently perform them. Let an execution block be the union of several parallel blocks. Combining two or more parallel blocks into an *execution block* will incur data dependence in an execution block, and the number of independent sets may be reduced. As shown in Figure 3(c), combining four parallel blocks into an execution block, the vertices in an execution block can be partitioned into four independent sets which are labeled by $T_i, 1 \leq i \leq 4$. In the following we define the *threads* to denote the maximum independent sets in each execution block.

Definition: Thread
Vertices in an execution block can be partitioned into a maximum number of sets such that there exists no edge connecting two vertices which belong to different sets. Under this partition each set is called a *thread*.

The *DBSI* execution form is translated based on an execution block. That is, multiple CPUs can concurrently perform an execution block without synchronization. This can be achieved by assigning each thread to each CPU. Since there exists no edge connecting two threads, no synchronization is needed. Within an execution block, reuse opportunities exist between two vertices if they have a dependence relation. That is, there is a reuse opportunity between two instance vertices if they are linked by an edge. Combining several parallel blocks into an execution block, the number of threads may be decreased and the number of reuse opportunities may be increased. Figures 3(b) and 3(c), respectively, display the combination of two and four parallel blocks into an execution block. In Figure 3(b) there are 15 statement instances in an execution block in which seven threads can be found to be concurrently executed on four CPUs. Let $S_i(j)$ denote the computation of instance $J=j$ running on statement S_i in loop $L1$. Since $S_1(9)$ and $S_2(11)$ are assigned to the same CPU, $A(2{:}129,9)$ generated at executing $S_1(9)$ will be reused at executing $S_2(11)$. Because instances connected by dependence edges will be collected into a thread and be performed by one CPU, the number of exploited reuse opportunities within an execution block can be measured by the number of edges falling in an execution block. As shown in Figure 3(b), ten vector reuse opportunities (there are double edges linking from instances of S_1 to instances of S_2) are exploited during the execution of an execution block. Since there are roughly $640/5 = 128$ execution blocks, in total, $128 \times 10 = 1280$ vector loads from memory to registers can be saved.

Instead, combining four parallel blocks into an execution block, the execution block contains 30 instance vertices which can be partitioned into four threads as shown in Figure 3(c). Since $S_1(9), S_2(11), S_3(11), S_1(14), S_3(15), S_2(16)$ are collected in thread T_4 and can be executed by the same CPU, reuse opportunities existing in the execution of these instances can be exploited. Within an execution block, there are 33 reuse opportunities (or edges) exploited by four CPUs. Since there are $640/10=64$ execution blocks in Figure 3(c), in total, there are $33 \times 64 = 2112$ reuse opportunities exploited when executing $L1$. Thus, combining more parallel blocks into an execution block can exploit more reuse opportunities. This is due to the fact that more edges will fall in an execution block of larger size.

Within an execution block, the scheduling of threads on multiple CPUs are determined by systems at run time. Most compilers of supercomputers specify the multithread at compile-time and schedule the threads on multiple CPUs at run-time. Let v_1 and v_2 denote two instance vertices that respectively exist in two neighboring execution blocks; there is at least one dependence edge linking from v_1 to v_2, as shown in Figure 3(b). Assume that

instances labeled v_1 and v_2 are respectively scheduled on CPUs P_1 and P_2 at run time. To guarantee that the execution of v_1 is before the execution of v_2, synchronization should be made between P_1 and P_2. Reuse can thus only occur within an execution block since the boundary vertices that have a dependence relation and are located in different execution blocks may be scheduled on different CPUs. Since the number of synchronizations is proportional to the number of execution blocks, combining more parallel blocks into an execution block will reduce the number of synchronizations. As shown in Figures 3(b) and 3(c), combining two and four parallel blocks into an execution block will cause 128 and 64 synchronizations, respectively.

However, combining too many parallel blocks into an execution block will lose the parallelism and result in low performance. Thus, factors such as the number of CPUs and the complexity of the dependence relation should be considered as criteria for determining the size of an execution block. In Section 3 we formally describe how to systematically determine the feasible size of an execution block and identify multithreads within an execution block. There we determine that combining four parallel blocks of loop $L1$ into an execution block is the best choice for vector computers equipped with four CPUs. As shown in Figure 3(c), four threads $T_i, 1 \leq i \leq 4$, within an execution block are identified as follows:

$$T_1 = \{S_2(8),S_1(10),S_1(11),S_2(12),S_3(12),\ S_2(13),S_1(15),S_1(16),S_3(16),S_2(17),S_3(17)\}$$

$$T_2 = \{S_2(9),S_1(12),S_3(13),S_2(14),S_1(17),S_3(18)\}$$

$$T_3 = \{S_2(10),S_3(10),S_1(13),S_3(14),S_2(15),S_1(18),S_3(19)\}$$

$$T_4 = \{S_1(9),S_2(11),S_3(11),S_1(14),S_3(15),S_2(16)\}$$

The final *DBSI* execution version can be written in the following loop $L1''$ by using the compiler directive Begin_Tasks ... End_Tasks.

```
Parallel do the following instances of the irregular parallel block
    S1(6),S1(7),S1(8),S2(6),S2(7),S3(6),S3(7),S3(8),S3(9)
End_Parallel
DO J=9,645,10
    C$DIR Begin_Tasks
        B(2:129,J - 1)=A(2:129,J - 3) + C(2:129,J - 6) - E(2:129,J - 3)
        A(2:129,J + 1)=B(1:128,J - 2) * E(2:129,J + 1)
        A(2:129,J + 2)=B(1:128,J - 1) * E(2:129,J + 2)
        B(2:129,J + 3)=A(2:129,J + 1) + C(2:129,J - 2) - E(2:129,J + 1)
        C(2:129,J + 3)=B(2:129,J - 1) * 3
        B(2:129,J + 4)=A(2:129,J + 2) + C(2:129,J - 1) - E(2:129,J + 2)
        A(2:129,J + 6)=B(1:128,J + 3) * E(2:129,J + 6)
        A(2:129,J + 7)=B(1:128,J + 4) * E(2:129,J + 7)
        C(2:129,J + 7)=B(2:129,J + 3) * 3
        B(2:129,J + 8)=A(2:129,J + 6) + C(2:129,J + 3) - E(2:129,J + 6)
        C(2:129,J + 8)=B(2:129,J + 4) * 3
    C$DIR Next_Task
        B(2:129,J)=A(2:129,J - 2) + C(2:129,J - 5) - E(2:129,J - 2)
```

```
        A(2:129,J + 3)=B(1:128,J) * E(2:129,J + 3)
        C(2:129,J + 4)=B(2:129,J) * 3
        B(2:129,J + 5)=A(2:129,J + 3) + C(2:129,J) - E(2:129,J + 3)          (L1'')
        A(2:129,J + 8)=B(1:128,J + 5) * E(2:129,J + 8)
        C(2:129,J + 9)=B(2:129,J + 5) * 3
    C$DIR Next_Task
        B(2:129,J + 1)=A(2:129,J - 1) + C(2:129,J - 4) - E(2:129,J - 1)
        C(2:129,J + 1)=B(2:129,J - 3) * 3
        A(2:129,J + 4)=B(1:128,J + 1) * E(2:129,J + 4)
        C(2:129,J + 5)=B(2:129,J + 1) * 3
        B(2:129,J + 6)=A(2:129,J + 4) + C(2:129,J + 1) - E(2:129,J + 4)
        A(2:129,J + 9)=B(1:128,J + 6) * E(2:129,J + 9)
        C(2:129,J + 10)=B(2:129,J + 6) * 3
    C$DIR Next_Task
        A(2:129,J)=B(1:128,J - 3) * E(2:129,J)
        B(2:129,J + 2)=A(2:129,J) + C(2:129,J - 3) - E(2:129,J)
        C(2:129,J + 2)=B(2:129,J - 2) * 3
        A(2:129,J + 5)=B(1:128,J + 2) * E(2:129,J + 5)
        C(2:129,J + 6)=B(2:129,J + 2) * 3
        B(2:129,J + 7)=A(2:129,J + 5) + C(2:129,J + 2) - E(2:129,J + 5)
    C$DIR End_Tasks
ENDDO
Parallel do the following irregular parallel block
    S1(639),...S1(645),S2(638),...,S2(645),S3(640),...,S3(645)
End_Parallel
```

Compared with $L1'$, loop $L1''$ is superior since the number of synchronizations in $L1''$ is $(645 - 9 + 1)/10=64$. There are 33 reuse opportunities which have been exploited in an execution block. In total, $33 \times (645-9+1)/10=2112$ vector loads are saved. Thus, loop $L1''$ is better than $L1'$ in the number of synchronizations and the degree of reuse exploitation. We run loops $L1'$ and $L1''$ on Convex C3840 with four CPUs by 10000 calls to scale the execution time, measured in seconds. Compared to loop $L1'$, loop $L1''$ has a 39.08% improvement in execution time. Table 1 summarizes the comparisons of loops $L1'$ and $L1''$. In general, statement instances within a thread (or a task) can be transformed into a loop form. In loop $L1''$, because the number of instances within a thread is small, we apply a loop unrolling technique to each task. In the next Section we formally discuss the *DBSI* technique to deal with problems of identifying parallel blocks, determining a feasible size of an execution block, and transforming the original vectorized program to *DBSI* execution form.

3. EXTRACTING MULTITHREAD WITH LOCALITIES IN *DBSI* EXECUTION FORM

The *DBSI* method can be categorized into three phases. In phase one, the translator should identify the basic *parallel blocks*. The second phase is to combine some *parallel blocks* into an *execution block*. In the third phase, compilers or a translator should identify the threads in an execution block. A transformation scheme then automatically restructures the

Table 1. Comparisons of loops $L1'$ and $L1''$

	Factors related to system performance			
Versions of loops	Number of threads	Number of synchronizations	Opportunities of reuse exploitation	Execution time (s)
$L1'$	4	640	0	19.34
$L1''$	4	64	2112	10.68

original vectorized loop program and inserts the proper compiler directives to generate the *DBSI* execution form.

The first two phases of the *DBSI* mechanism are introduced here. We first describe how the translator can identify the basic parallel blocks for a given loop program. Without loss of generality, we assume that the input source is vectorized in the first dimension of array operations. Let *value* $v(\bar{d})[5]$ of the dependence vector $\bar{d}=(d_1,\ldots,d_n)$ be

$$\sum_{i=2}^{n-1}\left(d_i\prod_{j=i+1}^{n}N_j\right)+d_n,N_j=u_j-l_j+1$$

where u_j and l_j, respectively, denote the value of upper and lower bounds of induction variable I_j. As an example, in Figure 1, the values of $\bar{d}_1$, $\bar{d}_2$, $\bar{d}_3$, $\bar{d}_4$ and $\bar{d}_5$ are, respectively, 2, 2, 3, 4 and 5. To identify the basic parallel blocks, a *restricted dependence graph* defined as follows should first be constructed.

Definition: Restricted dependence graph (*RDG*) Assume there are s statements in loop L. Let E_i denote the set of edges pointing to node S_i in *DG*, where $1 \le i \le s$. Let $\bar{d}_j$ be the labels of edges $e_j \in E_i$. A *restricted dependence graph* $RDG(N,E')$ or *RDG* of loop L is a subgraph of $DG(N,E)$. The node set in *RDG* is the same as the one in *DG*, and the edge set E' is a subset of E. Let $E'=\bigcup_{1\le i\le s}E_i'$. Edge $e_j \in E_i$ belongs to E_i' if its label has a minimum value in E_i, for $1 \le i \le s$. That is,

$$E'=\bigcup_{1\le i\le s}E_i' \text{ and } E_i'=\{e_j|e_j\in E_i, v(\bar{d}_j)=\min(v(\bar{d}_{j'})), \text{ for all } e_{j'}\in E_i\}$$

Figures 1 and 4 display the *DG* and *RDG*, respectively of loop *L*1. The value of the dependence vector pointing to S_i in *RDG* indicates the amount of parallel instances of S_i in a parallel block.

To identify the parallel block in *SIDG* for a given *RDG*, an $s \times 1$ restricted matrix R_m representing the *RDG* and an $s \times s$ transformation matrix T operated on R_m should be constructed. In *RDG*, let the dependence vector $\bar{d}$ be the label of edge e pointing to node S_i. The value of the ith row of R_m is equal to the value of $\bar{d}$. The jth row of matrix T is equal to I_i if there exists an edge pointing from statement S_i to statement S_j in *RDG*, where I_i is the ith row of the $s \times s$ identity matrix and $1 \le i,j \le s$. For instance, the restricted matrix R_m and the transformation matrix T of loop *L*1 are, respectively,

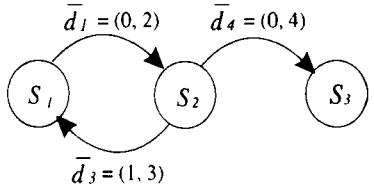


Figure 4. Restricted dependence graph of loop L1

$$R_m = \begin{bmatrix} v(\bar{d}_3) \\ v(\bar{d}_1) \\ v(\bar{d}_4) \end{bmatrix}_{3\times 1} = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}_{3\times 3}$$

The value in the ith row of R_m denotes the number of instances of S_i in the first irregular basic parallel block. The multiplication relation of matrices T and R_m denotes the block size transition from the current parallel block to the next one.

Let T^i denote the multiplication of i matrices T. A minimum integer r is said to be a *repeating number* if r satisfies $T^c \times R_m = T^{r+c} \times R_m$ for $c \geq 1$. The existence of repeating number r is obvious. The size of basic parallel blocks of *SIDG* can be represented by several $s \times 1$ *block size matrices* (*BSM*),

$$BSM = \left[\ T^c * R_m\ \right]_{s\times 1} \left[\ T^{c+1} * R_m\ \right]_{s\times 1} \cdots \left[\ T^{c+r-1} * R_m\ \right]_{s\times 1}$$

where the ith matrix $[T^{c+i-1} * R_m]$, $1 \leq i \leq r$, denotes the size of the ith basic parallel block if we ignored the first c irregular parallel blocks. Statement instances in *SIDG* can be categorized into r types of basic block whose size can be denoted by *BSM*. For example, in loop $L1$,

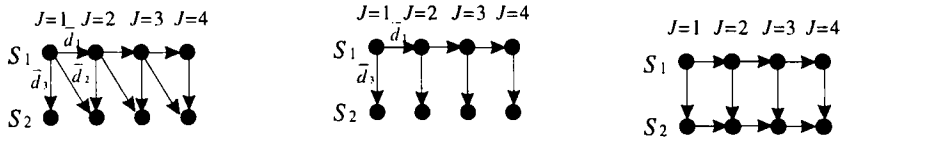
$$T * R_m = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix} \qquad T^2 * R_m = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix} \qquad T^3 * R_m = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix}$$

Thus, we have c=1 and r = 2. The size of the basic parallel blocks can be denoted by

$$BSM = \begin{bmatrix} 2 \\ 3 \\ 2 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}$$

which denotes the size of two types of basic parallel blocks as shown in Figure 3(a). The size of the first type of basic parallel block can be represented by the first column $[2,3,2]^t$ of *BSM*. The first row has value 2, denoting that there are two instances of S_1 in type one of the parallel block as shown in Figure 3(a).

If an execution block consists of r contiguous parallel blocks, the number of instances within an execution block is equal to $\sum_{i=1}^{s} \sum_{j=1}^{r} p_i^j$, where p_i^j denotes the value of the ith row of the jth column of *BSM*. This yields the following property.



$\vec{d}_1$ and $\vec{d}_3$ are necessary edges

$\vec{d}_2$ is unnecessary edge

(a) Original *SIDG*.

(b) The *SIDG* after removing the unnecessary edges.

(c) The *SIDG* for "otherwise" condition.

Figure 5. Necessary and unnecessary edges in SIDG

3.0.1. Property 3.1

The basic parallel blocks have the property that, combining r parallel blocks into an execution block, the instances of *SIDG* will be equally partitioned. That is, all execution blocks have the same size of $\sum_{i=1}^{s}\sum_{j=1}^{r} p_i^j$ instance vertices.

By way of example, property 3.1 can be examined in loop $L1$. In Figure 3(b), combining r=2 parallel blocks into an execution block, each execution block contains 15 instances.

Usually, the number of CPUs is less than the number of threads in a basic parallel block. Too many threads cannot benefit from parallelism and, on the contrary, limit reuse exploitation. Thus, further combining several basic parallel blocks into an execution block is needed. Once the size of an execution block is determined, the number of threads and synchronizations and the degree of reuse exploitation can be determined automatically.

Within an execution block, a pair of vertices (v_i,v_j) is said to be *reachable* if there exists at least one path in *SIDG* from v_i to v_j which may consist of several directed edges. A dependence vector $\vec{d}_i \in SIDG$ is said to be *necessary* if there exists at least one reachable pair (v_i,v_j) such that removing $\vec{d}_i$ will cause (v_i,v_j) to be unreachable. For example, the unnecessary dependence $\vec{d}_2$ in Figure 5(a) can be removed and the resultant *SIDG* is shown in Figure 5(b). Both $\vec{d}_1$ and $\vec{d}_3$ are necessary. For instance, removing $\vec{d}_1$ will cause $(S_1(1),S_1(2))$ to be unreachable. It is easy to verify that an existing dependence edge is necessary or unnecessary. A dependence edge $\vec{d}_i$ is unnecessary if it is a linear combination of other dependence vectors.

In what follows, we pay attention to the determination of size of an execution block. Before that, we introduce some notation which is used in the derivation of a feasible size of an execution block:

s:	the number of statements in loop body
k:	the number of parallel blocks combined into an execution block, k is a multiple of r
$t(k)$:	the number of threads existed in an execution block
$\#\vec{d}_i$:	the number of instance pairs that are connected by dependence vector $\vec{d}_i$ within an execution block
p_i^j:	the value of ith row of jth column of *BSM*
$\xi(k)$:	the number of total instances within an execution block
y:	the number of dependence vectors in an execution block
m:	the number of necessary dependence vectors in an execution block
$\#CPU$:	the number of CPU system supports

Without loss of generality, let $\bar{d}_i, 1 \leq i \leq m$, be the necessary dependence vectors and $\bar{d}_i, m+1 \leq i \leq y$, be the unnecessary dependence vectors. According to property 3.1, the number of total instances in an execution block is equal to

$$\xi(k) = \frac{k}{r} \sum_{i=1}^{s} \sum_{j=1}^{r} p_i^j \tag{1}$$

The number of threads within an execution block can be estimated by

$$t(k) = \begin{cases} \xi(k) - \sum_{i=1}^{m} \#\bar{d}_i & \text{if } \xi(k) - \sum_{i=1}^{m} \#\bar{d}_i > 1 \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

The reason is stated as follows. After removing unnecessary edges from an execution block, there still exist $\sum_{i=1}^{m} \#\bar{d}_i$ edges in an execution block. Since two vertices linked by a directed edge belong to the same thread, there are $\sum_{i=1}^{m} \#\bar{d}_i$ vertices collected into the same thread. The number of threads is, at most

$$\textit{the number of total vertices in an execution block} - \sum_{i=1}^{m} \#\bar{d}_i = \xi(k) - \sum_{i=1}^{m} \#\bar{d}_i$$

However, if there exists a complex dependence relation among vertices such that the number of threads is only one, the value $\xi(k) - \sum_{i=1}^{m} \#\bar{d}_i$ may be less than one. As shown in Figure 5(c), all dependence edges are necessary, and the value of $\xi(k) - \sum_{i=1}^{m} \#\bar{d}_i$ is $8 - 10 = -2$. Thus, the 'otherwise' condition is needed to deal with this condition. As an example, in Figure 3(c), all edges except $\bar{d}_2$ are necessary, and the value of $\xi(k) - \sum_{i=1}^{m} \#\bar{d}_i$ is $30-(7+8+8+3)=4$. Thus, we know that the execution block combined by four parallel blocks has four threads. Since the number of threads is the main parameter for determining the size of an execution block, for quickly determining the feasible size of an execution block we use equation (2) to measure the number of threads for a given fixed size execution block instead of extracting threads from the given execution block.

The following theorem derives the degree of reuse exploitation and the number of synchronizations for a given execution block.

3.0.2. Theorem 3.2:

Let there be k parallel blocks combined into an execution block. The number of synchronizations and the number of reuse exploitations in loop L are, respectively,

$$\frac{s * \prod_{i=2}^{n}(u_i - l_i + 1)}{\xi(k)} \text{ and } \frac{s * \prod_{i=2}^{n}(u_i - l_i + 1)}{\xi(k)} * \sum_{i=1}^{y} \#\bar{d}_i$$

Proof: Since synchronization is needed to guarantee the completion of execution of an execution block, the number of synchronizations is equal to the number of execution blocks and can be evaluated by the division of total statement instances in the loop over the size of an execution block. That is, the number of synchronizations is

$$\frac{s\prod_{i=2}^{n}(u_i - l_i + 1)}{\xi(k)}.$$

By the description of Section 2, the number of reuse exploitations is the multiplication of the number of execution blocks and the number of necessary and unnecessary edges in an execution block. Since the number of necessary and unnecessary edges in an execution block is $\sum_{i=1}^{y} \#\bar{d}_i$, theorem 3.2 holds.

For example, in loop $L1$, the number of statements is s=3. There are r=2 types of basic parallel blocks in *SIDG*. If we combine k=2 parallel blocks into an execution block as shown in Figure 3(b), the number of total instances in an execution block is equal to $\xi(2)=\sum_{i=1}^{3}\sum_{j=1}^{2} p_i^j=15$. By applying equation (2), the number of threads in an execution block is

$$\xi(2) - \sum_{i=1}^{m} \#\bar{d}_i = 15 - (2 + 3 + 3 + 0) = 7$$

In Figure 3(b), instances of an execution block are divided into seven threads T_i, for $1 \le i \le 7$. The number of synchronizations is $\frac{3\times640}{15}$=128. The degree of reuse exploitation is

$$\frac{3 \times 640(\#\bar{d}_1 + \#\bar{d}_2 + \#\bar{d}_3 + \#\bar{d}_4 + \#\bar{d}_5)}{15} = \frac{3 \times 640(2 + 2 + 3 + 3 + 0)}{15} = 1280$$

Similarly, as shown in Figure 3(c), the execution block combined by four parallel blocks has $t(4)$=4 threads. The number of synchronizations is $\frac{3\times640}{30}$=64. The number of reuse exploitations is $\frac{3\times640\times33}{30}$=2112. Thus, the size of the execution block combined by four parallel blocks is better than one combined by two parallel blocks due to the former having fewer synchronizations and a higher degree of reuse exploitation.

If the number of threads is larger than the number of available CPUs, we can further combine more parallel blocks into a larger execution block such that the number of threads and the number of synchronizations can be decreased and the degree of reuse exploitation can be increased. Value k that satisfies the following criterion can be a feasible solution to the determination of size of an execution block.

3.0.3. Criterion

The maximal value k that satisfies

$$t(k) = \xi(k) - \sum_{i=1}^{m} \#\bar{d}_i \ge \#CPU \tag{3}$$

is a feasible solution to the determination of size of an execution block.

Value k satisfying equation (3) yields that the reuse opportunities can be exploited and the number of synchronizations can be reduced as possible under the constraint that the degree of parallelism is maximized according to the number of available CPUs.

In loop $L1$, the optimal value of k=4 can be obtained since the *#CPU* is 4 in the Convex C3840 vector computer. Figure 3(c) displays the four threads in an execution block which is

obtained by combining four parallel blocks. A greedy algorithm can be applied to determine the value of k. If the current execution block has the number of threads larger than *#CPU*, we may increase the value of k as far as equation (3) holds. To prevent the value of k growing without limit, the following heuristic rule can be applied in the greedy algorithm.

3.0.4. Rule

If both the conditions

1. $t(k)=t(k+r)$ and
2. all the y dependence edges $\bar{d}_j$, for $1 \leq j \leq y$, are contained in the execution block with size $\xi(k)$

hold, combine all parallel blocks into an execution block. That is, set k to value

$$\frac{s \times r \times \prod_{i=2}^{n}(u_i - l_i + 1)}{\sum_{i=1}^{s}\sum_{j=1}^{r} p_i^j} \tag{4}$$

and then exit the greedy steps. Otherwise, increase the value of k as far as equation (3) holds. The heuristic rule will be activated if the number of threads is unchanged when increasing the size of an execution block. Note that, if the parallel block's size is too small such that the number of threads within a parallel block is less than the number of CPUs, compilers may additionally extract threads from the vectorized loop. Thus, in the worst case, the multithreading scheme will degenerate to the original rules of current vector compilers of the Convex C38 series. After the size of the execution block is determined, the next goal of the *DBSI* approach is to identify the threads and transform them into the *DBSI* execution form.

The time complexity of identifying t threads is $O(|E|)$[13], where $|E|$ denotes the number of edges in *DG* of loop L. Translation then should be further made to transform the original loop program L into *DBSI* execution form in which multiple threads are defined and the reuse of each thread is exploited as follows:

DO I=l,$l - 1 + \prod_{i=2}^{n}(u_i - l_i + 1), \frac{\xi(k)}{s}$
  C$DIR Begin_Tasks
    Statement instances of thread T_1
  C$DIR Next_Task
    Statement instances of thread T_2
    ⋮
  C$DIR Next_Task
    Statement instances of thread T_t
  C$DIR End_Tasks
ENDDO

The restructuring of instances in one thread into one loop or loops can be easily achieved since the instances of the same statement in one task have the modulation relation[13].

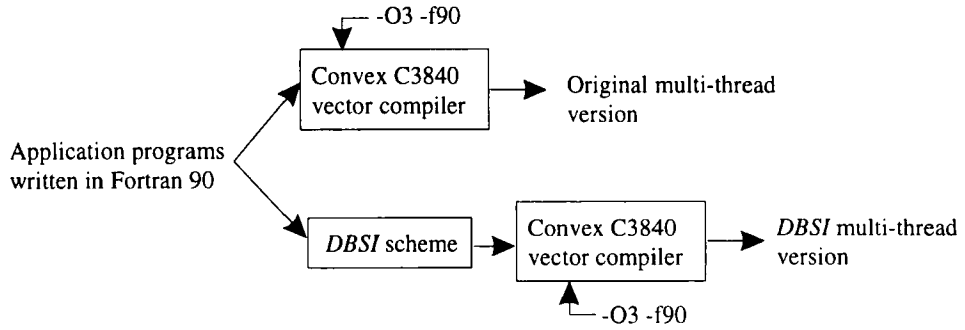


Figure 6. Two multithread versions of loop programs

Thus, instances within one thread can be easily restructured into loop or loops. A simple but not flexible scheme to schedule them into one task is loop unrolling, for the case that the number of instances in a thread is small. The final loop of the transformed *DBSI* form for loop $L1$ is thus obtained as shown in loop $L1''$.

In this Section we have described the multithreading scheme. The extracted threads can be concurrently executed on multiple CPUs in a vector processing fashion of a few synchronizations and a high degree of reuse exploitation. Firstly, we present a method to identify the basic parallel blocks. We also derive the condition to determine the feasible size of an execution block. The transformation scheme is then proposed to restructure the original vectorized program into a *DBSI* execution form. The performance analysis of our *DBSI* scheme is discussed in the next Section by considering several benchmarks.

4. PERFORMANCE ANALYSIS

In this Section we measure the performance improvement of several application programs by applying the proposed *DBSI* scheme. For each program, two versions of a multithreaded program are measured in Convex C3840 which are equipped with four CPUs for parallel processing. For the first version, we take a vectorized program written in Fortran 90 as the input of a vector compiler of Convex. To inform the vector compiler of Convex C3840 generating a multithread object code, we set the compilation option with '-O3 -f90'. The object code generated by a vector compiler is referred to the *original version*. Instead, another version is generated by the following two steps. Firstly, the vectorized program written in Fortran 90 is taken as the input of the *DBSI* scheme. The *DBSI* scheme analyzes the dependence relation and then partitions the vector operations into multithread by inserting the compiler directives. The multithread code translated by using the *DBSI* scheme is then taken as the input of the vector compiler of Convex C3840 in the second step. The generated object code by these two steps is referred to the *DBSI version*. The generation of these two versions is shown in Figure 6.

Loops selected as the source programs for comparison can be roughly cataloged into three classes. The first class is the vector benchmark that is extracted from NETLIB of NCHC (National Center for High Performance Computing). The benchmark consists of 107 subroutines of loops that are originally designed for testing the vectorization capability of PFC[10]. In total, 65 subroutines can be vectorized by the vector compiler of Convex. The 65 subroutines are compiled and the execution times of two versions, the original version

Table 2. Applications and their abbreviations

Applications	Abbreviation
Fourier least-squares approximation[15]	FLSA
Jacobi method for solution of linear equations[15]	Jacobi
Barycentric form of Lagrange interpolation[15]	BFLI
Accumulating a sum (A. Sum)[16]	ASum
Solving linear equation by Gaussian elimination (loop 1)[16]	SLEGE1
Solving linear equation by Gaussian elimination (loop 2)[16]	SLEGE2
Computing the uniform norm of matrix A and A inverse[16]	Uniform norm
Gauss–Seidel iterations[16]	GSI
Comparing compound Simposon's and Newton-Cotes integration[16]	CSNCI
Computing the value of a filtered discrete Fourier transform[16]	FDFT
Cholesky's method for solving algebraic equations[14]	CHOLESKY
Finding smallest eigenvalues and eigenvectors in ascending order[14]	EIGENVALUE
Finding value of LAMBDA by equation $BMX=(1/LAMBDA)X$[14]	LAMBDA
Solving elliptic partial differential equations[14]	ELLIPTIC
Combining two images into one image	IMAG
SOR relaxation	SOR
Loops widely discussed in previous papers[4][2][5]	PAPR

and the *DBSI* version, are compared. There are 44 subroutines which perform as well as without applying the *DBSI* scheme. This is due to the fact that the two versions of these subroutines have the same number of synchronizations and there is no reuse opportunity in these subroutines. In total, 21 subroutines are improved by applying our *DBSI* scheme. The second class selected as the benchmark programs consists of several libraries including subroutines of BLAS1 and BLAS2. The subroutines of BLAS1 and BLAS2 used as the input source are also stored in NETLIB of NCHC. The third class consists of several application programs[14–16]. Most of these programs are selected from the numerical computation programs. Only those that are vectorizable applications are considered. Table 2 lists these applications and its abbreviation.

The experimental results of execution time and speed-up for these three classes of programs are summarized in Table 3. Note that the benchmark programs with the same execution time in both versions are not listed in Table 3. Under the assistance of the *DBSI* scheme, the vector compiler of Convex C3840 generates more efficient multithread codes. The main factors of improvement are the number of synchronizations and the reuse exploitation. The *DBSI* scheme first analyzes the dependence relation and then partitions the vector operations into four threads such that the number of synchronizations and memory accesses can be as few as possible. The reuse exploitation of the vector register data has a significant effect on those programs that are vectorized in the second dimension of array operations. This is due to the fact that the vector data accessing with a larger memory stride needs more memory accessing time. This effect can be found in the speed-up of programs S029, S084, S100, CSNCI and FDFT as shown in Table 3.

To preserve semantic correctness, the Convex vector compiler will produce a sequential loop in the outermost loop level. This will enable a large number of synchronizations. By applying the *DBSI* scheme, multithread will be identified under the consideration of reducing the number of synchronization and reuse exploitations. For instance, consider

Table 3. Comparisons of *original* and *DBSI* versions for subroutines, libraries and applications (only programs for which speed-up was achieved are listed)

Benchmarks		Problem size	Original version (ms)	*DBSI* version (ms)	Speed-up
Vector Subroutines	S022	1024 × 1024	482	166	3.98
	S023	1024 × 1024	469	147	3.19
	S029	1024 × 1024	153	19.6	7.80
	S030	1024 × 1024	146	48	3.04
	S032	1024 × 1024	142	49	2.89
	S044	1048576	99	51	1.94
	S045	1048576	126	79	1.59
	S047	1024 × 1024	174	68	2.55
	S048	1024 × 1024	162	67	2.41
	S049	1024 × 1024	158	72	2.19
	S067	1024 × 1024	143	72	1.98
	S068	1024 × 1024	132	62	2.13
	S070	1024 × 1024	167	75	2.26
	S082	1024 × 1024	231	125	1.84
	S083	1024 × 1024	368	159	2.31
	S084	1024 × 1024	184	23.4	7.86
	S090	1048576	43	41.3	1.04
	S091	1048576	49	45.2	1.08
	S092	1048576	135	76	1.77
	S100	1024 × 1024	1327	217	6.11
	S101	1024 × 1024	255	138	1.84
BLAS1	SAXPY	1048576	850	509	1.66
	SCOPY	1048576	40	32	1.25
	SSCAL	1048576	440	319	1.38
	DAXPY	1048576	3340	2657	1.26
	DCOPY	1048576	70	67	1.05
	DSCAL	1048576	510	269	1.90
BLAS2	SGEMV	1024 × 1024	1770	620	2.85
	SGBMV	1024 × 1024	180	82	2.20
	DGEMV	1024 × 1024	2470	1190	2.08
	DGBMV	1024 × 1024	396	273	1.45
Applications	FLSA	1024	32.1	25	1.28
	Jacobi	1024 × 1024	16	13	1.23
	BFLI	1024 × 1024	31	27	1.15
	ASum	1024	3.8	2.6	1.46
	SLEGE1	1024 × 1024	7.8	6.5	1.2
	SLEGE2	1024 × 1024	55	39	1.40
	Uniform norm	1024 × 1024	74	58	1.27
	GSI	1024 × 1024	6.3	4.8	1.31
	CSNCI	1024 × 1024	53	5.7	9.29
	FDFT	1024 × 1024	57	5	11.4
	CHOLESKY	1024 × 1024	308	215	1.43
	EIGENVALUE	1024 × 1024	412	293	1.40
	LAMBDA	1024 × 1024	373	197	1.89
	ELLIPTIC	1024 × 1024	211	134	1.57
	IMAG	1024 × 100	10.66	8.28	1.28
	SOR	1024 × 1024	362	198	1.83
	PAPR	512 × 8192	6	3.2	1.88

the following vectorized loop program (IMAG) which combines two images, respectively stored in arrays $A(I,J)$ and $B(I,J)$:

```
DO   J=2,n
   A(2:m + 1,J)=(B(1:m,J − 1) + A(2:m + 1,J))/2
   B(2:m + 1,J)=(A(1:m,J − 1) + B(2:m + 1,J))/2                    (L2)
ENDDO
```

By setting option '-O3 -f90' to the vector compiler of the Convex C3840 supercomputer, we obtain the generated assembly code equivalent to the following program:

```
DO   J = 2,n
   C$DIR Force_Parallel
   DO   I=2,m + 1,m/4
      A(I:I + m/4 − 1,J)= (B(I − 1:I + m/4 − 2,J − 1)+ A(I:I + m/4 − 1,J))/2      (L2')
      B(I:I + m/4 − 1,J)=(A(I − 1:I + m/4 − 2,J − 1) + B(I:I + m/4 − 1,J))/2
   ENDDO
ENDDO
```

The Convex compiler divides the vector operations with length m into four threads concurrently executed by four CPUs. An attempt to interchange loops I and J to reduce the synchronizations will result in semantic error. By applying our *DBSI* scheme, the following program can be obtained:

```
B(2,3:n)=(A(1,2:n − 1) + B(2,3:n))/2
B(2:m + 1,2)=(A(1:m,1) + B(2:m + 1,2))/2
DO  J=2,n − 1,2
   C$DIR  Begin_Tasks
         A(2:m/2,J)=(B(1:m/2 − 1,J − 1) + A(2:m/2,J))/2
         B(3:m/2 + 1,J + 1)=(A(2:m/2,J) + B(3:m/2 + 1,J + 1))/2
   C$DIR  Next_Task
         A(m/2 + 1:m,J)=(B(m/2:m − 1,J − 1) + A(m/2 + 1:m,J))/2
         B(m/2 + 2:m + 1,J + 1)=(A(m/2 + 1:m,J) + B(m/2 + 2:m + 1,J + 1))/2
   C$DIR  Next_Task
         A(2:m/2,J + 1)=(B(1:m/2 − 1,J) + A(2:m/2,J + 1))/2                  (L2'')
         B(3:m/2 + 1,J + 2)=(A(2:m/2,J + 1) + B(3:m/2 + 1,J + 2))/2
   C$DIR  Next_Task
         A(m/2 + 1:m,J + 1)=(B(m/2:m − 1,J) + A(m/2 + 1:m,J + 1))/2
         B(m/2 + 2:m + 1,J + 2)=(A(m/2 + 1:m,J + 1) + B(m/2 + 2:m + 1,J + 2))/2
   C$DIR  End_Tasks
ENDDO
A(m + 1,2:n − 1)=(B(m,1:n − 2) + A(m + 1,2:n − 1))/2
A(2:m + 1,n)=(B(1:m,n − 1) + A(2:m + 1,n))/2
```

The comparison of $L2'$ and $L2''$ in the number of threads, synchronizations and the reuse exploitation is displayed in Table 4. Under the assistance of the *DBSI* scheme, loop $L2''$ has a higher degree of reuse exploitation and fewer synchronizations than $L2'$. Figure 7

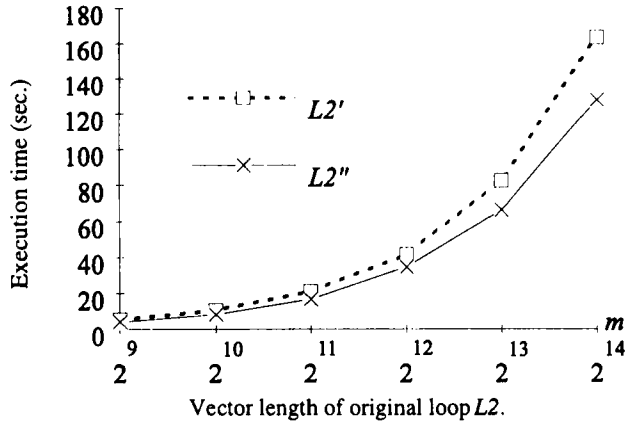


Figure 7. Comparison of execution times of loops L2' and L2"

displays the comparison of the execution time of loops $L2'$ and $L2''$ by calling these two versions of loops 1000 times (to measure the execution time in seconds) with n=100 and m ranging from 2^9 to 2^{14}. On average, the performance improvement of the *DBSI* execution form is 22.3%.

Table 4. Comparisons of loops $L2'$ and $L2''$

	Factors related to execution time of loops program		
Versions of loops	Number of threads	Number of synchronizations	Opportunities of reuse exploitation
$L2'$	4	$n-1$	0
$L2''$	4	$\frac{n-1}{2}$	$4 \times \frac{n-1}{2}$

The *DBSI* scheme reduces not only the number of synchronizations but also the memory accesses for a vectorized program written in Fortran 90. Experimental results show that a vector compiler assisted by the *DBSI* multithreading technique usually produces a more efficient code for users to complete their program execution.

5. CONCLUSIONS

In this paper a systematic multithreading technique has been proposed. The presented mechanism collects vector operations which have a dependence relation into one thread and are individually executed by one CPU. Multiple vector processors can concurrently perform multithread with fewer synchronizations and a higher degree of vector reuse exploitation. Several real applications are measured by applying our *DBSI* approach on Convex C3840 vector computers. The experimental results show that the proposed *DBSI* technique assists the vector compiler, transforming some cases of the loop program into a favorite execution form and obtains a shorter execution time.

ACKNOWLEDGEMENT

This work was supported by the National Science Council of the Republic of China under grant NSC 84-2213-E-008-010.

REFERENCES

1. L. S. Liu, C. W. Ho and J. P. Sheu, 'On the parallelism of nested for-loops using index shift method', *Proceedings of 1990 International Conference on Parallel Processing*, Vol. II, 1990, pp. 119–123.
2. W. Shang, M. T. O'Keefe and J. A. B. Fortes, 'On loop transformations for generalized cycle shrinking', *IEEE Trans. Parallel Distrib. Syst.*, **5**, (2), 193–204 (1994).
3. J. P. Sheu and C. Y. Chang, 'Synthesizing nested loop algorithms using nonlinear transformation method', *IEEE Trans. Parallel Distrib. Syst.*, **2**, (3), 304–317 (1991).
4. C. D. Polychronopoulos, 'Compiler optimization for enhancing parallelism and their impact on architecture design', *IEEE Trans. Comput.*, **C-37**, (8), 991–1004 (1988).
5. S. D. Wang and C. M. Wang, 'Compiler techniques for extracting loop-level parallelism', *J. Inf. Sci. Eng.*, **7**, 543–563 (1991).
6. R. Allen and K. Kennedy, 'Vector register allocation', *IEEE Trans. Comput.*, **C-41**, (10), 1290–1317 (1992).
7. D. Callahan, S. Carr and K. Kennedy, 'Improving register allocation for subscripted variables', *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 53–65.
8. M. E. Wolf and M. S. Lam, 'A data locality optimizing algorithm', *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 30–44.
9. F. Irigoin and R. Triolet, 'Supernode partitioning', *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1988, pp. 319–329.
10. H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing Company, 1990.
11. Cray. *CF77 volume 4: Parallel Processing Guide*, Cray Research Inc., 1981.
12. Convex, *Convex Fortran Optimization Guide*, Convex Computer Corporation, Richardson, FX, 1990.
13. C. Y. Chang, 'Design and implementation of compilation techniques for vector computers', Ph.D. dissertation, Department of Computer Science and Information Engineering, National Central University, Taiwan, ROC, June 1995.
14. M. L. James, G. M. Smith and J. C. Wolford, *Applied Numerical Methods for Digital Computation*, Harper and Row, 1985.
15. S. M. Pizer and V. L. Wallace, *To Compute Numerically Concepts and Strategies*, Boston Little, Brown and Company, 1983.
16. S. Yakowitz and F. Szidarovszky, *An Introduction to Numerical Computations*, 2nd edn, Macmillan Publishing Company, New York, 1989.