Extracting Multi-thread with Data Localities for Vector Computers *

Jang-Ping Sheu and Chih-Yung Chang

Department of Computer Science and Information Engineering
National Central University, Chung-Li 32054, TAIWAN
E-mail: sheujp@mbox.ee.ncu.edu.tw

Abstract

In this paper, we propose a source-to-source compilation strategy to partition vectorized loop programs into multi-thread execution form. Each partitioned thread consists of instances of statements with localities in vector registers. The multi-threading scheme gives a novel combination of loop unrolling [9], *statement instances reordering, index shifting* [5], *vector register reuse exploiting* [2], *and multi-threading. Expermantal results show that our multi-threading scheme assists vector compiler of Convex C38 series to reduce the number of memory accesses and the number of synchronizations among CPUs and usually obtains a better performance.*

Keywords: *Data dependence, loop optimization, multi-thread, parallelism, vector compilers, vector register reuse.*

1 Introduction

Recent vector computers are equipped with several CPUs and a hierarchical memory to offer both vectorization and multiprocessing capabilities. In vector processing, a large class of loop programs applied in solving differential equations, fourier transform, image processing, and neural processing can be translated or rewritten into a vector execution form [1]. For these available vectorized programs, proper partition of the vector operations for parallel processing can significantly utilize the hardware design of multiple CPUs and reduce the execution time. In parallel processing, several CPUs can work together to concurrently perform tasks which consist of scalar or vector operations defined in program. Synchronizations are needed among these CPUs if their references of array data have dependence relation.

The factors determine the system performance of vector computers are not only parallelism but also the memory management. If compilers are capable of exploiting the opportunities of vector data reuse, the execution time of loop program can be significantly improved [2]. However, the discussions of reuse exploiting in vector computers focus mainly on single CPU. Recently, Irigoin and Triolet proposed supernode partitioning technique [4] to vectorize and parallelize the sequential loop program according to the data dependence relation. Multiple CPUs thus can concurrently execute vector operations in a manner of data reuse exploitation. However, two more efforts are needed to be further paid. First, there is one dependence vector considered to exploit the reuse opportunity. The degree of reuse exploitation may be further improved if the given loop program has complex dependence relation. Second, the supernode partitioning treats an iteration as the minimum parallelism unit. Statement instance partition may extract more parallelism from loop program which has the *π-block* dependence graph. We are motivated to design a systematic strategy to flexibly partition the vector operations into multi-thread such that the number of synchronizations and the degree of data reuse exploitation can be improved.

The approach adopted here automatically reconstructs the loop programs into multi-thread execution form with fewer synchronizations and more reuse exploitation of vector register data. Comparisons are made to illustrate that vector compiler assisted by the proposed multi-thread scheme usually produces a more efficient code for users to early complete their program execution.

2 Loop Model and Basic Concept

Definition: Data Dependence Graph

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 by S denotes a statement S in loop L, while an edge labeled by a *dependence vector* $\bar{d} = (d_1, d_2, \ldots, d_n)$ links from node S to node S' if array data are first referenced by S in iteration $\bar{i}$, and then referenced again by S' in iteration $\bar{j}$, where $(d_1, d_2, \ldots, d_n) = \bar{j} - \bar{i}$. □

There are four types of dependence vectors, the *input, out-*

*This work was supported by the National Science Council of the Republic of China under Grant NSC 83-0408-E-008-018.

put, true dependencies, and the *anti-dependence* [9], possibly existing in loop L. Traditional vectorization techniques [9] 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 π-*block* DG are easy to be dealt with in parallelism extraction, synchronizations reduction, and reuse exploitation. Here, we focus our attention on loops with 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 effect significantly 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.

```
DO    I2 = l2, u2
  ...
  DO    In = ln, un
      vectorized statement S1
        ...                          (L)
      vectorized statement Se
  ENDDO
  ...
ENDDO
```

where l_i and u_i are respectively normalized lower bound and upper bound of index variable 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 transform, image processing, and neural processing falls in this model.

Several vector compilers offer compiler directive instructions for programer to manually perform the loop optimization. However, unless that users are skilled in parallel program design, it is difficult to write an efficient program with explicit definitions of multi-thread. In this paper, the proposed multi-threading scheme automatically restructures the loop program and inserts the proper compiler directives to specify the multi-thread. Two compiler directives related to multi-thread identification are introduced as follows. Instructions introduced here are recognizable to compiler of Convex C38 series [3].

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

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

The FORCE_PARALLEL directive will assist 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 4 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 4 CPUs.

Another compiler directive also can be used to define multi-thread. 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
```

can instruct the compilers to parallelize t threads (or tasks). If the BEGIN_TASKS ... END_TASKS directive appears in loop body of a loop program, the execution form is referred to *DBSI* (Different Body Same Instance) since the multi-thread is defined by running same instance on different statement groups.

Consider the following vectorized loop program $L1$.

```
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
```

In $L1$, the data dependence vectors are

$\bar{d}_1 = (0,2)$, $\bar{d}_2 = (0,2)$, $\bar{d}_3 = (1,3)$ between S_1 and S_2,
$\bar{d}_4 = (0,4)$, $\bar{d}_5 = (0,5)$ between S_2 and S_3

as shown in Figure 2.1 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, their data reuse opportunities can be exploited by considering the $\bar{d}_3$ and $\bar{d}_4$. Thus, this input dependence is redundant in reuse exploitation and can be ignored.

Assume that there are 4 CPUs supported by system. Compiler of Convex C3840 will parallelize loop I and sequentially execute the loop J as shown in the following equivalent code.

```
DO  J = 6, 645
  C$DIR Force_Parallel
  DO  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 4 subsets each contains contiguous 32 instances and then vectorizes each subset in a vector length of 32. Due to that loop J is a sequential loop, synchronization should be made at each running iteration of loop J. An attempt to perform loop interchange on loops I and J to reduce the

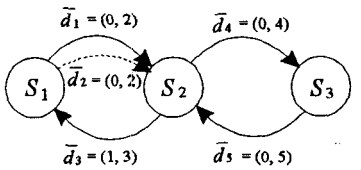


Figure 2.1: The DG of loop $L1$.

number of synchronizations will cause semantic error. This is due to the fact that $B(33,6)$ generated by the first thread $I = 2$ at the execution of $J = 6$ on S_2 will be referenced again by the second thread $I = 34$ at the execution of $J = 9$ on S_1. Two drawbacks are found in multi-thread 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 $L1'$.

Instead, the 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 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 they have dependence relation, where $\bar{d}$ is the label of edge linking from S_i to S_j in DG and v_i and v_j are the respective instances of S_i and S_j. □

Figure 2.2 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 *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 2.3(a) displays the parallel blocks of *SIDG* of $L1$. In the next section, we will illustrate how to derive the

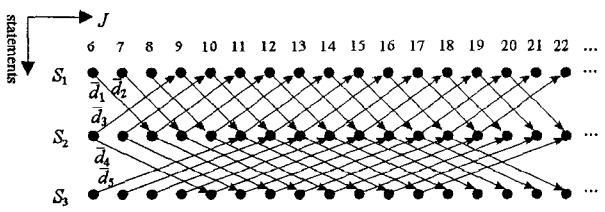
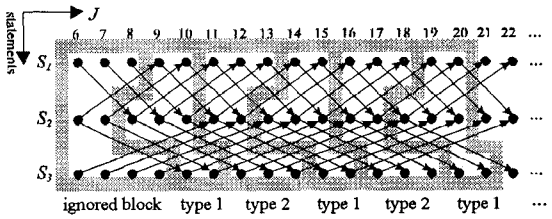
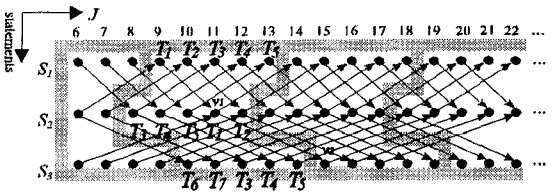


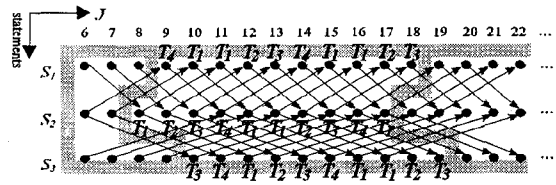
Figure 2.2: The *SIDG* of loop $L1$.



(a) Two types of parallel blocks in loop $L1$.



(b) Combining two parallel blocks into an execution block.



(c) Combining four parallel blocks into an execution block.

Figure 2.3: The paralle blocks and the execution blocks of loop $L1$.

parallel blocks. In Figure 2.3(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 loop 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 2.3(a). There are 7 and 8 parallel instances can be concurrently executed in type one and type two of parallel blocks, respectively. In a parallel block, too many independent instances have no benefit to parallelism since 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 2.3(c), combining four parallel blocks into an execution block, the vertices in an execution block can be partitioned into 4 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 maximum number of sets such that there exists no edge

connecting two vertices which are belonged 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 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 2.3(b) and 2.3(c) respectively display the combination of two and four parallel blocks into an execution block. In Figure 2.3(b), there are 15 statement instances in an execution block in which 7 threads can be found to be concurrently executed on 4 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 in the same CPU at executing $S_2(11)$. Because that 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 fallen in an execution block. As shown in Figure 2.3(b), 10 vector reuse opportunities (there are double edges linking from instances of S_1 and instances of S_2) exploited during the execution of an execution block. Since there are roughly 640/5 =128 execution blocks, in total, 128*10= 1280 vector loads from memory to vector registers can be saved.

Instead, combining 4 parallel blocks into an execution block, the execution block contains 30 instance vertices which can be partitioned into 4 threads as shown in Figure 2.3(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 existed in the execution of these instances can be exploited. Within an execution block, there are 33 reuse opportunities exploited by 4 CPU. In total, there are 33 * 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 with larger size.

Within an execution block, the scheduling of threads on multiple CPUs are determined by systems at run time. Most compilers of supercomputers can only specify the multi-thread and are not capable of scheduling threads on multiple CPUs. Let v_1 and v_2 denote two instance vertices that are respectively existed in two neighboring execution blocks and there is at least one dependence edge linking from v_1 to v_2, as shown in Figure 2.3(b). Assume that instances labeled by v_1 and v_2 are respectively scheduled on CPUs P_1 and P_2 at run time. For guaranteeing that the execution of v_1 is before the execution of v_2, synchronization should be made between P_1 and P_2. Reuse thus can only be occurred within an execution block since the boundary vertices that have 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 2.3(b) and 2.3(c), combining 2 and 4 parallel blocks into an execution block will cause respective 128 and 64 synchronizations.

However, combining too many parallel blocks into an execution block will loose the parallelism and result in some CPUs idle. Thus, factors such as the number of CPUs and the complexity of dependence relation should be considered as the criteria to the determination of the number of parallel blocks combined into an execution block. As described in the next section, we will determine that combining four parallel blocks of loop $L1$ into an execution block is the best choice for vector computers equipped with 4 CPUs. From Figure 2.3(c), four threads $T_i, 1 \leq i \leq 4$, within an execution block can be 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 finial *DBSI* execution version can be written in 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$DIR Begin_Tasks
    S11 : B(2:129, J-1) = A(2:129, J-3) + C(2:129, J-6)
                        - E(2:129, J-3)
    S12 : A(2:129, J+1) = B(1:128, J-2) * E(2:129, J+1)
    S13 : A(2:129, J+2) = B(1:128, J-1) * E(2:129, J+2)
    S14 : B(2:129, J+3) = A(2:129, J+1) + C(2:129, J-2)
                        - E(2:129, J+1)
    S15 : C(2:129, J+3) = B(2:129, J-1) * 3
    S16 : B(2:129, J+4) = A(2:129, J+2) + C(2:129, J-1)
                        - E(2:129, J+2)
    S17 : A(2:129, J+6) = B(1:128, J+3) * E(2:129, J+6)
    S18 : A(2:129, J+7) = B(1:128, J+4) * E(2:129, J+7)
    S19 : C(2:129, J+7) = B(2:129, J+3) * 3
    S1a : B(2:129, J+8) = A(2:129, J+6) + C(2:129, J+3)
                        - E(2:129, J+6)
    S1b : C(2:129, J+8) = B(2:129, J+4) * 3
  C$DIR Next_Task
    S21 : B(2:129, J) = A(2:129, J-2) + C(2:129, J-5)
                      - E(2:129, J-2)
    S22 : A(2:129, J+3) = B(1:128, J) * E(2:129, J+3)
    S23 : C(2:129, J+4) = B(2:129, J) * 3
    S24 : B(2:129, J+5) = A(2:129, J+3) + C(2:129, J)
                        - E(2:129, J+3)                    (L1'')
    S25 : A(2:129, J+8) = B(1:128, J+5) * E(2:129, J+8)
    S26 : C(2:129, J+9) = B(2:129, J+5) * 3
  C$DIR Next_Task
    S31 : B(2:129, J+1) = A(2:129, J-1) + C(2:129, J-4)
                        - E(2:129, J-1)
    S32 : C(2:129, J+1) = B(2:129, J-3) * 3
    S33 : A(2:129, J+4) = B(1:128, J+1) * E(2:129, J+4)
    S34 : C(2:129, J+5) = B(2:129, J+1) * 3
    S35 : B(2:129, J+6) = A(2:129, J+4) + C(2:129, J+1)
                        - E(2:129, J+4)
    S36 : A(2:129, J+9) = B(1:128, J+6) * E(2:129, J+9)
    S37 : C(2:129, J+10) = B(2:129, J+6) * 3
  C$DIR Next_Task
    S41 : A(2:129, J) = B(1:128, J-3) * E(2:129, J)
    S42 : B(2:129, J+2) = A(2:129, J) + C(2:129, J-3)
                        - E(2:129, J)
    S43 : C(2:129, J+2) = B(2:129, J-2) * 3
    S44 : A(2:129, J+5) = B(1:128, J+2) * E(2:129, J+5)
    S45 : C(2:129, J+6) = B(2:129, J+2) * 3
    S46 : 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 have been exploited in an execution block. For example, in the fourth task of loop $L1''$, vector data $A(2:129, J)$ and $E(2:129, J)$ referenced by S_{41} can be immediately reused by S_{42}. In total, there are $33*(645-9+1)/10 = 2112$ vector loads 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 4 CPUs by 10000 calls to scale the execution time measured in second. Compared to loop $L1'$, loop $L1''$ has 39.08% improvement in execution time.

Note that, 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 loop unrolling technique to each task.

3 Extracting Multi-Thread 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 translators should identify the threads in an execution block. The transformation scheme then automatically restructures the original vectorized loop program and inserts the proper compiler directives to generate the *DBSI* execution form.

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* $v(\bar{d})$ [7] of the dependence vector $\bar{d} = (d_1, \ldots, d_n)$ be

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

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 2.1, value of $\bar{d}_1$, $\bar{d}_2$, $\bar{d}_3$, $\bar{d}_4$, and $\bar{d}_5$ are respective 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 one in DG and the edge set E' is a subset of E.

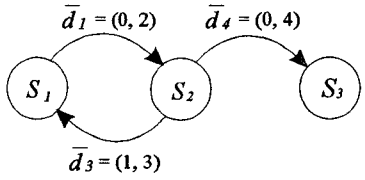


Figure 3.1: The restricted dependence graph of loop $L1$.

Edge $e_j \in E_i$ belongs to E' if its label has minimum value in E_i, for $1 \le i \le s$. That is,

$$E' = \{e_j | e_j \in E_i, v(\bar{d}_j) = \min(v(\bar{d}_{j'})), \text{ for all } e_{j'} \in E_i\}.$$

□

Figures 2.1 and 3.1 respectively display the DG and RDG of loop $L1$. The value of dependence vector pointing to S_i in RDG indicates the amount of parallel instances of S_i in a parallel block. The dependence vectors in RDG represent the information of the size of parallel blocks in $SIDG$.

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 dependence vector $\bar{d}$ be the label of edge e pointing to node S_i. The value of 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 $L1$ are respective

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

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 *repeating number* if r satisfies $T^c * R_m = T^{r+c} * R_m$ for $c \ge 1$. 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)

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

where the ith column matrix $T^{c+i-1} * R_m$, $1 \le i \le r$, 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 $L1$, $T * R_m = [2, 3, 2]^t$, $T^2 * R_m = [3, 2, 3]^t$, and $T^3 * R_m =$

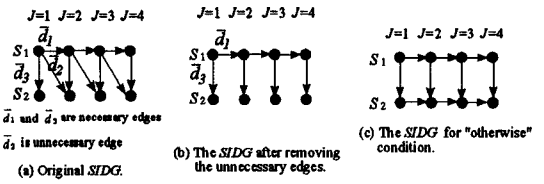


Figure 3.2: Necessary and unnecessary edges in *SIDG*.

$[2,3,2]^t$. Thus, we have $c = 1$ and $r = 2$. The size of 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 2.3(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 parallel block as shown in Figure 2.3(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 ith row of jth column of BSM. This yields the following property holds.

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 2.3(b), combining $r = 2$ parallel blocks into an execution block, each execution block contains 15 instances and the *SIDG* is equally partitioned. Usually, the number of CPUs is less than the number of threads in a basic parallel block. Too many threads can not benefit to parallelism and in contrary limit the reuse exploitation. Thus, further combining several basic parallel blocks into an execution block is needed.

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) unreachable. For example, the unnecessary dependence $\vec{d}_2$ in Figure 3.2(a) can be removed and the resultant *SIDG* is shown in Figure 3.2(b). Both $\vec{d}_1$ and $\vec{d}_3$ are necessary. For instance, removing $\vec{d}_1$ will cause $(S_1(1), S_1(2))$ 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 will pay attention to the determination of size of an execution block. Before that, we introduce some notations which are 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
$t(k)$:	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
m:	the number of necessary dependence vectors existed in an execution block
$\#CPU$:	the number of CPUs system supports

According to the Property 3.1, the number of total instances in an execution block is equal to

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

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

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

for all necessary dependence vectors $\vec{d}_i$ in an execution block. The reason is stated as follows. After removing unnecessary edges from an execution block, there still exist $\sum_{i=1}^{m} \#\vec{d}_i$ edges in an execution block. Since two vertices linked by a directed edge belong to the same thread, there are $\sum_{i=1}^{m} \#\vec{d}_i$ vertices should be combined with other vertices. The number of threads is at most $\xi(k) - \sum_{i=1}^{m} \#\vec{d}_i$. However, if there exists complex dependence relation among vertices such that the number of thread is only one, the value $\xi(k) - \sum_{i=1}^{m} \#\vec{d}_i$ may be less than one. As shown in Figure 3.2(c), all dependence edges are necessary, the value of $\xi(k) - \sum_{i=1}^{m} \#\vec{d}_i$ is $8 - 10 = -2$. Thus, the "otherwise" condition is needed to deal with this condition. As an example, in Figure 2.3(c), all edges except $\vec{d}_2$ are necessary, the value of $\xi(k) - \sum_{i=1}^{m} \#\vec{d}_i$ is $30 - (7+8+8+3) = 4$. Thus, we know that the execution block combined by 4 parallel blocks has 4 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 formula (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.

THEOREM 3.2: Let there be k parallel blocks combined into an execution block. The number of synchronizations and the number of reuse exploitation in loop L are respective

$$\frac{s * \prod_{i=2}^{n}(u_i - l_i + 1)}{\xi(k)} \text{ and } \frac{s * \sum_{j=1}^{x} \#\vec{d}_j * \prod_{i=2}^{n}(u_i - l_i + 1)}{\xi(k)},$$

where x is the number of dependence vectors in DG.

Proof:

See [6] in detail. □

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 2.3(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 formula (2), the number of threads in an execution block is

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

In Figure 2.3(b), instances of an execution block are divided into 7 threads T_i, for $1 \leq i \leq 7$. The number of synchronizations is $\frac{3*640}{15} = 128$. The number of reuse exploitation is

$$\frac{3 * (\#\bar{d}_1 + \#\bar{d}_2 + \#\bar{d}_3 + \#\bar{d}_4 + \#\bar{d}_5) * 640}{15}$$

$$= \frac{3 * (2 + 2 + 3 + 3 + 0) * 640}{15} = 1280.$$

Similarly, as shown in Figure 2.3(c), the execution block combined by 4 parallel blocks has $t(4) = 4$ threads. The 4 CPUs system supported can be fully utilized. The number of synchronizations is $\frac{3*640}{30} = 64$. The number of reuse exploitation is $\frac{3*33*640}{30} = 2112$. Thus, the execution block combined by 4 parallel blocks is better than one combined by 2 parallel blocks due to that the former has fewer synchronizations and 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.

CRITERION: The maximal value k that satisfies

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

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

Value k satisfying condition (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 Convex C3840 vector computer. Figure 2.3(c) displays the 4 threads in an execution block which is obtained by combining 4 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 condition (3) holds. 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 dimension. Thus, in worst case, the multi-threading scheme will degenerate to the original rules of current vector compilers.

After the size of execution block is determined, the next goal of *DBSI* approach is to identify the threads and transform them into the *DBSI* execution form. Partitioning method in [6] can be applied such that instances of an execution block can be partitioned into several threads. The time complexity of identifying t threads is $O(|E|)$, where $|E|$ denotes the number of edges in DG of loop L. Translation then can be 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.

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 multithreaded program are measured in Convex C3840 which is equipped with 4 CPUs for parallel processing. For the first version, we take vectorized program written in Fortran 90 language as the input of vector compiler of Convex. To inform the vector compiler of Convex C3840 generating a multi-thread object code, we set the compilation option with '-O3 -f90'. The object code generated by vector compiler is referred to the *original version*. Instead, another version is generated by the following two steps. First, the vectorized program written in Fortran 90 language is taken as the input of the *DBSI* scheme. The *DBSI* scheme analyzes the dependence relation and then partitions the vector operations into multi-thread by inserting the compiler directives. The multi-thread code translated by using the *DBSI* scheme is then taken as the input of vector compiler of Convex C3840 in the second step. The generated object code by these two steps is referred to the *DBSI version*.

Loops selected as the source programs for comparison can be roughly cataloged into two 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 [1] [9]. In total, there are 65 subroutines can be vectorized by vector compiler of Convex. The 65 subroutines are compiled and the execution time of two versions, the original version and the *DBSI* version, is compared. In total, there are 21 subroutines 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 level 1 BLAS (Basic Linear Algebra Subprograms) and level 2 BLAS respectively perform the vector/vector and matrix/vector operations. All subroutines of BLAS1 and BLAS2 are designed in libraries for calls in most supercomputers. The subroutines of BLAS1 and BLAS2 used as the input source are also stored in NETLIB of NCHC.

The experimental results of execution time and speedup for these two classes of programs are summarized in Table I. Note that, the benchmark programs with the same execution time in both versions are not listed in Table I. Under the assistance of *DBSI* scheme, the vector compiler of Convex C3840 generates more efficient multi-thread codes. Compared with the original version, the *DBSI* version in average has a speedup of 2.54. The main factors of improvement are the number of synchronizations and the reuse exploitation. The reuse exploitation of vector register data has 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 speedup of programs S029, S084, and S100 as shown in Table I.

Table I. Comparisons of original and DBSI versions for benchmarks.

Benchmarks	Problem Size	Original version	*DBSI* version	speedup
S022	1024 X 1024	482 ms	166 ms	3.98
S023	1024 X 1024	469 ms	147 ms	3.19
S029	1024 X 1024	153 ms	19.6 ms	7.80
S030	1024 X 1024	146 ms	48 ms	3.04
S032	1024 X 1024	142 ms	49 ms	2.89
S044	1048576	99 ms	51 ms	1.94
S045	1048576	126 ms	79 ms	1.59
S047	1024 X 1024	174 ms	68 ms	2.55
S048	1024 X 1024	162 ms	67 ms	2.41
S049	1024 X 1024	158 ms	72 ms	2.19
S067	1024 X 1024	143 ms	72 ms	1.98
S068	1024 X 1024	132 ms	62 ms	2.13
S070	1024 X 1024	167 ms	75 ms	2.26
S082	1024 X 1024	231 ms	125 ms	1.84
S083	1024 X 1024	368 ms	159 ms	2.31
S084	1024 X 1024	184 ms	23.4 ms	7.86
S090	1048576	43 ms	41.3 ms	1.04
S091	1048576	49 ms	45.2 ms	1.08
S092	1048576	135 ms	76 ms	1.77
S100	1024 X 1024	1327 ms	217 ms	6.11
S101	1024 X 1024	255 ms	138 ms	1.84
SAXPY	1048576	850 ms	509 ms	1.66
SCOPY	1048576	40 ms	32 ms	1.25
SSCAL	1048576	440 ms	319 ms	1.38
DAXPY	1048576	3340 ms	2657 ms	1.26
DCOPY	1048576	70 ms	67 ms	1.05
DSCAL	1048576	510 ms	269 ms	1.90
SGEMV	1024 X 1024	1770 ms	620 ms	2.85
SGBMV	1024 X 1024	180 ms	82 ms	2.20
DGEMV	1024 X 1024	2470 ms	1190 ms	2.08
DGBMV	1024 X 1024	396 ms	273 ms	1.45

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

5 Conclusions

In this paper, a systematic multi-threading technique has been proposed. The presented mechanism collects vector operations which have reuse opportunities into one thread and individually executed by one CPU. For vector computers with powerful vector processing and parallel processing capabilities, multiple CPUs can concurrently perform multi-thread with less synchronizations and higher degree of vector reuse exploitation.

Comparisons have been made in Convex C3840 supercomputer by using several application loop programs. Expermantal results show that our multi-threading scheme assists vector compiler of Convex C38 series to generate a more efficient multi-thread code and usually obtains a better performance.

References

[1] R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4, pp. 491-542, Oct. 1987.

[2] R. Allen and K. Kennedy, "Vector Register Allocation," *IEEE Transactions on Computers*, Vol. 41, No. 10, pp. 1290-1317, Oct. 1992.

[3] CONVEX FORTRAN Optimization Guide, CONVEX Computer Corporation.

[4] F. Irigoin and R. Triolet, "Supernode Partitioning," *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 319-329. Jan. 1988.

[5] L. S. Liu, C. W. Ho, and J. P. Sheu, "On the parallelism of Nested For-Loops Using Index Shift Method," *Proceeding of 1990 International Conference on Parallel Processing*, Vol. II, pp. 119-123, 1990.

[6] J. P. Sheu and C. Y. Chang, "Extracting Multithread, Reducing Synchronizations, and Improving Localities for Vector Computers," Technique Report, Department of Computer Science and Information Engineering, National Central University, 1994.

[7] S. D. Wang and C. M. Wang, "Compiler Techniques for Extracting Loop-Level Parallelism," *Journal of Information Science and Engineering 7*, pp. 543-563, 1991.

[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*, pp. 30-44, June 1991.

[9] H. Zima and B. Chapman, Supercompilers for Parallel and Vector Computers, Addison-Wesley Publishing Company, 1990.