Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Design and Implementation of a Parallelism Detector for CONVEX SPP-1000 System

Yi-Hsin Hsiao Chao-Tung Yang† Shian-Shyong Tseng‡
Dept. of Comp. and Info. Sci. Nat'l. Chiao Tung Univ.
Hsinchu, Taiwan 300, ROC, Phone: +886-3-5715900, Fax: +886-3-5721490

Abstract

In this paper we implement a robust parallelism detector for CONVEX SPP-1000 shared-memory multiprocessor system. The parallelism detector is improved by combining the PPD and K-Test that both are our previous research results. The detector can extract the available parallelism on loop of a program and translate the original program into parallel form by using CONVEX compiler directive. Then, the result program can be run on CONVEX SPP-1000 system to achieve high speedup rates.

1 Introduction

In this paper, we implement a robust parallelism detector to translate a sequential program into a parallel form for CONVEX SPP-1000 multiprocessor system. This parallelism detector is proved by combining our previous research results, the PPD [9] and K-Test [6]. PPD collects the characteristics of the loop for K-Test and takes the traditional FORTRAN 77 source program as input to yield the corresponding parallel code. Most data dependence test algorithms provided a particular way to analyze the whole program, but in a program not all loops have the same property and suit for the same algorithm. Based on the experience of K-Test [6], different loops in a program may take different algorithms to obtain the more correct and efficient results. The framework of PPD is divided into two phases, *analysis phase* and *codegen phase*. In analysis phase, the K-Test is used for checking if the linear equation formed by array subscript has an appropriate integer solution. The effect is the determination of the execution modes of all loops. In codegen phase, the outcome of analysis phase is referred to produce the prompted parallel code and PPD will generate the code with compiler directives which are provided by CONVEX SPP FORTRAN compiler. The optimizations for synchronized statements of DOACROSS loops are also taken. The K-Test will choose an appropriate test by knowledge-based techniques, and then will apply the resulting test to detect data dependence on loops. PPD will generate the code with compiler directives which are provided by CONVEX SPP FORTRAN compiler.

*This work was supported in part by National Science Council, Republic of China, under Grant No. NSC85-2213-E009-082.

†He is a Associate Researcher in the ROCSAT Ground Segment at Nat'l. Space Program Office (NSPO), Hsinchu, Taiwan 300, ROC. E-mail: ctyang@aho.cis.nctu.edu.tw.

‡Corresponding author. He is a Professor in the Dept. of Comp. and Info. Sci. at Nat'l. Chiao Tung Univ., Hsinchu, Taiwan 300, ROC. E-mail: sstseng@cis.nctu.edu.tw.

2 Background

Data dependence testing [10, 4, 8] is the method used to determine whether dependencies exist between two subscript references to the same array in a nested loop. The index variables of the nested loops are assumed to be normalized to increase by 1 in this paper. Suppose that we want to decide whether or not there exists a dependence from statement S_1 to S_2. As we know, data dependence testing is equivalent to integer programming, and the most efficient integer programming algorithms known either depend on the value of the loop bounds or are order $O(n^{O(n)})$ [5] where n is the number of the loop variables. In general, there are approximate methods to solve this problem. So many algorithms are proposed to solve the problem by analyzing the linear equations formed by a pair of array references. Every existing test has its own advantage as for some aspect [6].

2.1 K-Test

A new approach by using knowledge-based techniques to solve the data dependence analysis problem is proposed in [6]. It can choose an appropriate test by using knowledge-based techniques, and then applies the resulting test to detect data dependence on loop. A rule-based system, called the K-Test, is developed by repertory grid analysis to construct the knowledge base.

A knowledge-based system is composed of two parts: the *development environment* and the *runtime environment* [7]. The former is used to build the knowledge base, while the latter is used to solve the problem. The *runtime environment* of the K-Test contains three components as shown in Figure 1, which are briefly described as follows.

Knowledge Base: This component contains knowledge required for solving the problem of determining an appropriate test to be applied. The knowledge can be organized in many different schemes, and can be encoded into many different forms. Therefore, there exist many choices of building the knowledge base.

Inference Component: This component is essentially a computer program that provides a method for reasoning about information in the knowledge base along with the input, and for forming conclusion.

Testing Algorithm Library: The library collects several representative tests either proposed by others or designed by ourselves, including the GCD test, Banerjee test, I test, and Power test.

The dependence testing process can be described as follows. First, the input, a set of equations, is fed into the

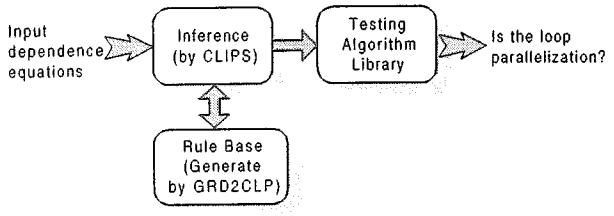

Figure 1: Components of K-Test.

inference component. Then, the inference component reasons about knowledge and draw a conclusion, a test. Finally, the resulting test is applied and the answer is generated. It should be noted that the knowledge base and the testing algorithm library shown in Figure 1 are flexible; that is, they are not fixed. Users can modify these two components so long as the efficiency and precision of the system are retained.

2.2 PPD

The practical parallelism detector (PPD) [9], that uses the lex and yacc auxiliaries is implemented in PFPC (a portable FORTRAN parallelizing compiler running on OSF/1) at NCTU for finding the parallelism available in loops. The PPD can extract the potential DOALL and DOACROSS loops in a program by invoking a combination of the ZIV test and the I test for verifying array subscripts. Furthermore, if DOACROSS loops are available, an optimization of synchronization statement is made. Experimental results show that PPD is more reliable and accurate than previous approaches.

PPD takes a conventional FORTRAN 77 source program as input and yields corresponding prompted parallel code. PPD is divided into two phases, *analysis* (Figure 2 (a)) and *codegen* (Figure 2 (b)). In the analysis phase, a single-subscript testing algorithm, the combination of the I test and ZIV test, is used to check whether the linear equations formed by array subscript have appropriate integer solutions. Instead of linearizing the array subscripts, we check them subscript-by-subscript since there is no certainty that any of them overrides the other in precision.

It also proposed two *ad hoc* techniques that look for trivial contradictions of direction vectors to improve upon the drawbacks of conventional subscript-by-subscript testing mechanisms [9]. The effect of the analysis phase is to determine all loop execution modes. A loop's execution mode may be made DOALL, DOACROSS, or DOSEQ. The first two can be executed in a fully or partially parallel manner, while the last one must be executed in the normal sequential style. In the codegen phase, the results of the analysis phase are used to produce prompted parallel codes. Optimizations of DOACROSS loop synchronized statements are also taken.

3 Parallelization on SPP-1000

3.1 SPP-1000 System

Our target machine is CONVEX SPP-1000 multiprocessor system, which was designed by Hewlett-Packard Company with 8 CPUs and crossbar 256 MB shared memory installed. This machine uses the CONVEX Exemplar SPP-UX 3.1 operating system, and FORTRAN compiler version 3.1. CONVEX FORTRAN includes standard FORTRAN as defined by the Americas National Standard FORTRAN 77 (ANSI X3.9-1978). It also includes selected FORTRAN 90 extensions; VAX-11 features; certain features of Cray, Sun, and Hewlett-Packard FORTRAN; and unique CONVEX extensions. The FORTRAN Language Reference contains a complete description of the CONVEX FORTRAN language [1].

The FORTRAN compiler provided some machine-independence optimization's levels. The programmers can specify the following optimization levels for performing machine-independent optimizations at the specified level as in Table 1. If this option is not specified, no machine-independent optimization is performed. The default optimizations is set at the machine instruction level. The FORTRAN language compiler cannot automatically parallelize loops when containing dependencies, but a rich set of directives, pragmas and data types are available to help programmers manually parallelize such loops by synchronizing (and, if necessary, ordering) access to the code containing the dependency. These directives can also be used to synchronize dependencies in parallel tasks. They allow programmers who can efficiently exploit parallelism in loop and region.

Level	Description
-O0	Basic block machine-independent scalar optimization
-O1	Program unit level scalar optimizations and global register allocation
-O2	Global instruction scheduling, software pipelining, and data localization optimizations.
-O3	Parallel optimizations

Table 1: Optimization levels.

3.2 Compiler Directives

The SPP-1000 FORTRAN compiler has provided some compiler directives to help programmer and compiler to make use of parallelism. Some compiler directives provide information to the compiler that it cannot determine on its own. Other directives instruct the compiler to override certain default conditions that control optimization, parallelization. In CONVEX FORTRAN, a compiler directive line has the following format:

```
C$DIR [ SPP | CSERIES ] directive [ , directive ... ]
```

A directive line begins in column one with the characters C$DIR. If one of the optional target machine attributes (SPP or CONVEX C series) is specified, the directive line is applied only when compiled for the target machine (either CONVEX SPP Series or CONVEX C Series). If two or more directives are specified, they are separated by commas. A directive must fit on one line; it cannot be continued. A directive can be surrounded by any number of comment lines. The CONVEX SPP-1000 has provided the following compiler directives for SPP Series as shown in Table 2. Those compiler directions was described as follows:

UNROLL

The UNROLL directive reduces loop overhead by replicating the body of the loop that follows. Unrolling is performed only on scalar loop. This directive is effective only in code

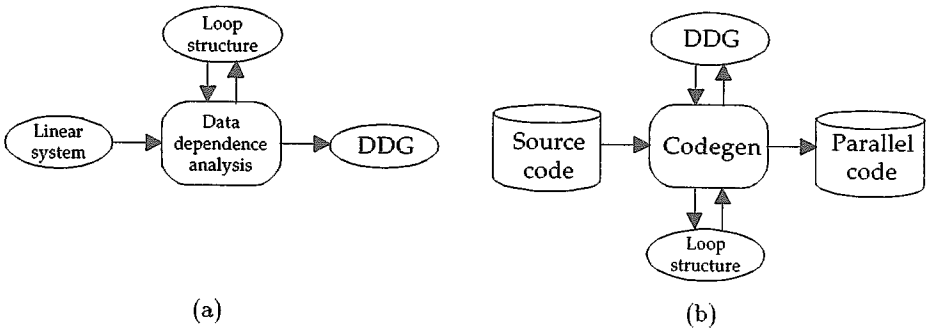

Figure 2: An overview of the analysis and codegen phases.

compiled at optimization level -O2 or higher. It has the form:
`C$DIR UNROLL [(UNROLL_FACTOR= n)]`
where, if the `UNROLL_FACTOR` argument is included, its value n specifies the number of times to replicate the body of the loop. Otherwise, the loop will be totally unrolled.

GATE

The `GATE` directive declares one or more variables of type `GATE`. This directive is useful only in code compiled at optimization level -O3. The format of this directive is shown as below:
`C$DIR GATE (gate-name [, gate-name ...])`
where `gate-name` is the name of the `GATE` variable to be declared. By using `GATE` variables and the attendant intrinsic routines, programmers can specify a block of execution code to one thread at a time.

LOOP_PARALLEL

The `LOOP_PARALLEL` directive specifies that the immediately following loop should be run in parallel. This directive is effective only in code compiled at optimization level -O3. The `LOOP_PARALLEL` compiler directive is available only on CONVEX SPP Series machines. A loop marked with a `LOOP_PARALLEL` directive must have a known number of iterations at loop invocation time. For this reason, `LOOP_PARALLEL` is not applied to WHILE loops, loop containing RETURN or STOP statements, or any other loop that exits by abnormal termination. A `LOOP_PARALLEL` will not be interchanged with any other loop in a loop nest.

The `LOOP_PARALLEL` directive is not applied to the implicit loop of FORTRAN 90 array section syntax; FORTRAN 90 array assignments are data-independent and are candidates for automatic parallelization. Any secondary induction variables in a `LOOP_PARALLEL` loop must be written as linear expressions of the primary induction variable and must be declared `LOOP_PRIVATE`. The format of the `LOOP_PARALLEL` directive is as follows:
`C$DIR LOOP_PARALLEL [(attribute-list)]` where the optional *attribute-list* can contain any one of the following combinations of attributes as shown in [2].

4 Our Approach

4.1 Integration of K-Test and PPD

To construct the robust parallel loop detector on CONVEX SPP-1000 the PPD and K-Test have been combined. The PPD includes two phase, one is the analysis phase and the other is codegen phase. The analysis phase is used to analyze the section of loop in source program, collect the attributes of the loop, and detect the parallelism of a loop. The part of data dependence analysis is the most importance in analysis phase. The K-Test is used to replace the part of data dependence analysis instead of the original one in analysis phase. The K-Test chooses an appropriate test by knowledge-based techniques, and then applies the resulting test to detect data dependence on loop. In order to use the K-Test, some additional information about the loop must be known. So the PPD's analysis phase is improved to increase the power of analysis, and to know the characteristic of the loop.

The K-Test is based on the characteristic of program and the knowledge base to decide which test will be used. An expert system shell called CLIPS, a forward reasoning rule-based tool, was used for this inference component. The knowledge base is constructed as a rule base, the translator, GRD2CLP, is utilized to translate the repertory grid and attribute ordering table to CLIPS's production rule. Since the loop characteristic was known, then the K-Test inference component can use the information to reason about knowledge and draw a conclusion, a test. The resulting test was applied to this loop of source program, and it will get the answer whether the loop can be run in parallel, with synchronization or series, and keep the result in data dependence graph.

The attributes of the loop nest structures for the sake of using K-Test include Unity_Coef, Bound_Known, Multi_Dim, Few_Ver, and Couple. These five attributes are described below:

- *Unity_Coef* whether the coefficients of variables are 1, 0, -1, or not. if yes, we assign this values as 5, otherwise 1.
- *Bound_Known* whether the loop bounds are known or not. If the loop bound was known we assign this value as 5, otherwise 1.
- *Multi_Dim* whether the array reference is multi-dimensional or not. We assign this value as the number of the dimension of the array reference.
- *Few_Ver* whether the number of variables in the equation is small or not. We assign this value as the number of variables in the loop.

- *Couple* whether the array references are couple or not. If yes, we assign this value as 5, otherwise 1.

The K-Test is based on the five attributes and his knowledge base to decide which test will be used. The repertory grid of the K-Test contains five attributes and four objects which are four existing data dependence tests as shown in Table 3.

BARRIERE	BEGIB_TASKE
NEXT_TASK	END_TASKS
BLOCK_LOOP	BLOCK_SHARED
CRITICAL_SECTION	END_CRITICAL_SECTION
FAR_SHARED_POINTER	GATE
LOOP_PARALLEL	LOOP_PRIVATE
NEAR_SHARED	NEAR_SHARED_POINTER
NO_BLOCK_LOOP	NODE_PRIVATE
NODE_PRIVATE_POINTER	NO_LOOP_DEPENDENCE
NO_PARALLEL	NO_PEEL
NO_PROMOTE_TEST	NO_SIDE_EFFECTS
NO_UNROLL_AND_JAM	ORDERED_SECTION
FAR_SHARED	END_ORDERED_SECTION
PEEL	PEEL_ALL
PREFER_PARALLEL	PROMOTE_TEST
PROMOTE_TEST_ALL	ROW_WISE
SAVE_LAST	SCALAR
TASK_PRIVATE	THREAD_PRIVATE
THREAD_PRIVATE_POINTER	UNROLL
UNROLL_AND_JAM	

Table 2: Compiler directives.

	GCD	Banerjee	I	Power
Unity_Coef	1/1	5/2	1/1	1/1
Bound_Known	1/2	5/D	1/1	5/2
Multi_Dim	1/1	1/1	1/2	5/2
Few_Var	5/1	5/1	1/2	1/2
Couple	1/1	1/1	1/1	5/D

Table 3: The RGA/AOT of the K-Test.

After performing parallelism detector, each loops will be marked with one of three different types. At the outer loop, if the loop has loop-carried dependences, it was marked with DOACROSS, otherwise, it was marked with DOALL. If the loop was not outer loop, the parallelizing overhead was too large, it was marked with DOSEQ.

4.2 The Algorithm

We now summarize the above discussion into two phases: *analysis* and *codegen*. An overview of these two phases is depicted in Figures 3, respectively. Since the analysis proceeds during parsing, DDG construction must accomplish after parsing the entire input source. During the codegen phase, the serial code, DDG, and loop nest structure are referenced in order to produce the desired prompts.

Analysis phase: The analysis phase is divided into four stages. These four stages are started after each loop block grammar rule reduction.

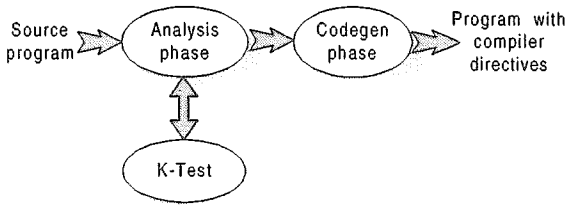

Figure 3: An overview of our approach.

- **Stage 1:** This stage primarily determines loop execution modes. Suppose that *instance* stands for DOALL, DOACROSS or DOSEQ, while L is a set of loop levels. `conditional_mark(`*instance*`,` L`)` will mark the *do-type* of each loop in L with *instance* according to the following rule:
  if *instance* takes precedence over a loop's original *do-type*, and the loop *mask* is false, then *instance* can be marked on the loop; otherwise the loop retains its original *do-type*.
- **Stage 2:** Break the nested DOACROSS loops in the loop nest structure. Since it is expensive much to synchronize all nested DOACROSS loops while the program is being executed, we preserve the outermost DOACROSS loop and mark DOSEQ on the other nested DOACROSS loops.
- **Stage 3:** Remove the virtual dependences in the DDG. Given a dependence relationship, say $S_1\delta_c^{(d)}S_{2_{(ap)}}$, if there exists another dependence relation, say $S'\delta_c^{(d')}S_{2(ap)}$ in which S' performs a write access and if $d' < d$, or $d' = d$ and S' is textually after S_1, then the given dependence relationship can be removed.
- **Stage 4:** Perform optimization on synchronization.
  - Step 1: Remove the LID records from the DDG since they are now necessary.
  - Step 2: For each entry in the DDG, remove redundant synchronization records that have the same dependence distance.
  - Step 3: Perform further optimization which can be applied if there are no conditional branches in the loop block. The process is mentioned in [10]:
    Given a dependence relationship $S_1\delta_c^{(d)}S_2$, if there exists another dependence relationship $S''\delta_c^{(d')}S'$ where S'' is textually after or equal to S_1, S' is textually before S_2, and d is a multiple of d', then the given dependence relationship can be removed.

Codegen phase: The prompted parallel codes are composed by the original source code, and compiler directives. The following rules show the way prompted parallel codes are generated.

- For each loop in the loop nest structure, if the *do-type* is DOALL, the compiler directive C$DIR LOOP_PARALLEL is inserted in the front of the original loop.
- For each loop in the loop nest structure, if the *do-type* is DOACROSS, the compiler directive C$DIR LOOP_PARALLEL(ordered) and C$DIR GATE is inserted in the front of the original loop.
- For each inmost loop in the loop nest structure, the compiler directive UNROLL is inserted in the front of the original loop.
- For each statement enclosed in the loop block, say S_i, the i^{th} row and the i^{th} column of the dependence matrix are referenced.
  - For each non-null entry in the i_{th} column traversed[1], and if there is a record, say $S_j \delta_c^{(d)} S_i$ in the entry (j, i), then the directive C$DIR ORDERED_SECTION(Gc)[2] is inserted at the appropriate position as follows:
    If there is no loop nested under loop level c, the directive C$DIR ORDERED_SECTION(Gc) is inserted immediately before S_i; or else it is inserted at the front of loop level c.
  - If there exists any synchronization record in i^{th} row, the compiler directive C$DIR END_ORDERED_SECTION is inserted at the appropriate position as follows:
    If there is no loop nested under loop level c, the directive C$DIR END_ORDERED_SECTION is inserted immediately after S_i; or else it is inserted at the rear of loop level c.

4.3 Program Transformation

In the code generation phase, the original code generation unit was to produce the parallel code for a prototype parallel FORTRAN compiler. Now, we replace it by generating the code with compiler directives that were provided by CONVEX SPP-1000 FORTRAN compiler, and the compiler directive will increase the effects of programs in SPP-1000. After performing parallelism detector, each loops will be marked with one of three different types, which are DOALL, DOACROSS, and DOSEQ. The different types of loop will be made the different transformation. The transformations are described detailedly as follows:

- DOALL: If the loop was marked with DOALL, it means that the following loop can be performed in parallel, so we add the compiler directive LOOP_PARALLEL in front of the loop. The LOOP_PARALLEL directive specifies that the immediately following loop should be run in parallel.
- DOACROSS: If the loop was marked with DOACROSS, it means that the loop can be performed in parallel with synchronization. If a loop can't be done in parallel, it may have a loop carry dependence. We can make use of this advantage to achieve high speedup rates. So we also add the compiler directive LOOP_PARALLEL in front of the loop. To keep the correctness of the result, some other option and directive must be added to guarantee the execution order right. Consider the FORTRAN code in Figure 4, which contains a *true dependence* on the array A that normally inhibits parallelization.

```
DO I = 1, N
  . PARALLELIZABLE CODE ···
  ...
  A(I) = A(I-1) + B(I)
  . MORE PARALLELIZABLE CODE ···
  ...
ENDDO
```

Figure 4: FORTRAN code with true dependence.

Assuming that the dependence shown is the only one in the loop, and that a significant amount of parallel code exists elsewhere in the loop, we can isolate the dependence and parallelize the loop as shown in Figure 5.

```
C$DIR GATE(LCD)
  LOCK = ALLOC_GATE(LCD)
  ...
  LOCK = UNLOCK_GATE(LCD)
C$DIR LOOP_PARALLEL(ORDERED)
  DO I = 1, N
    . PARALLELIZABLE CODE ···
    ...
C$DIR ORDERED_SECTION(LCD)
    A(I) = A(I-1) + B(I)
C$DIR END_ORDERED_SECTION
    . MORE PARALLELIZABLE CODE ···
    ...
  ENDDO
  LOCK = FREE_GATE(LCD)
```

Figure 5: loop_parallel synchronization code.

The loop is now parallelized in the manner described in Figure 6, and the ordered section containing the A(I) assignment will be executed in right iteration order, insuring that the value of A(I-1) used in the assignment is always valid. Assuming this loop runs on 4 threads, the synchronization of statement execution between threads is illustrated in Figure 7.

- DOSEQ: If the loop was marked with DOSEQ, it means that the following loop can't be performed in parallel, it doesn't need to make any transformation.

CONVEX SPP-1000 has also provided some other directives that can improve efficiency, In this paper, we also use the UNROLL directive, which can unroll the loop and improve efficiency by eliminating loop. But it can be only used in the inner loop. If the UNROLL_FACTOR argument is included (shown in Figure 8), its value n specifies the number of times to replicate the body of the loop.

[1] There may be multiple records linked in the entry.

[2] (Gc) represents the loop index variable that corresponds to loop level.

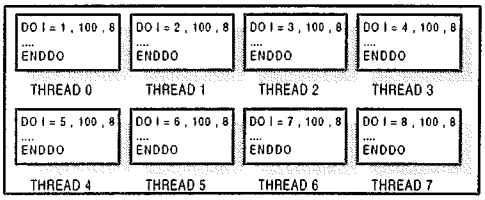

Figure 6: LOOP_PARALLEL (ORDERED)

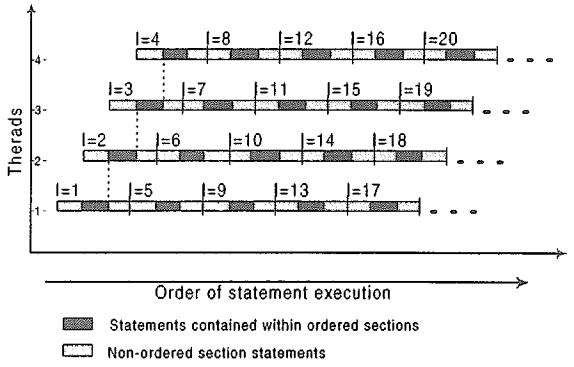

Figure 7: LOOP_PARALLEL (ORDERED) with synchronization.

When the loop can be done in parallel or has a loop-carried dependence by K-Test, the loop is translated if it is an outer loop; otherwise, since the overhead is too large, it will not be deal with. In the inner loop, the UNROLL directive will be used to eliminate the loop overhead.

5 Experiments

5.1 Efficiency of CONVEX SPP-1000 Compiler Directives

The experiments were performed on the CONVEX SPP-1000 system under the following conditions:

1. We compare the execution time between a program with compiler directives and without compiler directives.
2. We compare the execution time varying with different number of threads.
3. We compare the execution time between using UNROLL directive or not.
4. We compare the execution time on some practical data, e.g., matrix multiplication, adjoint convolution, and reverse adjoint convolution, and transitive closure with different matrix size.

First, the comparison of the execution time between with compiler directives and without compiler directives on CONVEX SPP-1000 is shown in Table 4. Obviously, the program

```
C$DIR UNROLL (UNROLL_FACTOR = 4)
DO I=1, N
  A(I)=B(I)+C(I)
ENDDO
```

(a) Source loop.

```
DO I=1, N ,4
  A(I)=B(I)+C(I)
  A(I+1)=B(I+1)+C(I+1)
  A(I+2)=B(I+2)+C(I+2)
  A(I+3)=B(I+3)+C(I+3)
ENDDO
```

(b) After translated.

Figure 8: Translation example with partial UNROLL.

with compiler directives will spend less time then the program without compiler directive, if the loop is a DOALL loop.

Examples	With parallel code	Without parallel code	Speedup
Loop 1	0.74	4.05	5.47
Loop 2	1.81	0.57	0.32
Loop 3	0.45	0.62	1.38
Loop 4	0.39	0.42	1.08

Table 4: Execution time with parallel and without parallel.

Where the loop examples are described as below:

- *Loop 1* is a single loop as shown in Figure 12(a), that can be executed in parallel without any synchronization code.
- *Loop 2* is a single loop, as shown in Figure 12(b), that can be executed in parallel but some synchronization codes are needed to guarantee the execution order.
- *Loop 3* is a nested loop, as shown in Figure 12(c), the outer loop can be executed in parallel without any synchronization code.
- *Loop 4* is a nested loop, Figure 12(d), the outer loop can be executed in parallel with some synchronization code.

The Loop 1 and Loop 3 are DOALL loops, it means that they can be executed in parallel without any synchronization overhead, so, the number of iterations is the factor of speedup. The different number of iterations causes different speedup are shown in Figure 9 for Loop 1.

The Loop 2 is a single loop, that can be executed in parallel with some synchronization code. As shown in Chapter 4.2, the translation with synchronization code must have significant amount of parallel code exists elsewhere in the loop, then the translate can gain the speedup. Loop 2 does not

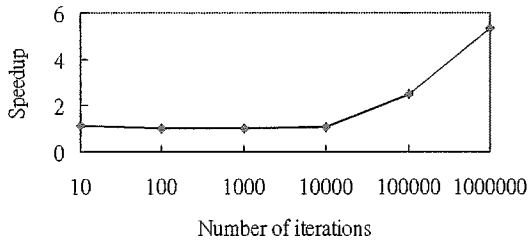

Figure 9: Speedup for different iteration.

have significant amount of parallel code exists, so this translation can not gain the speedup. In Loop 4, some parallel code was added in the Loop 2, the speedup was improved.

Because CONVEX SPP-1000 system is a multiusers system, the ability of gaining the right to used CPU number will effect the efficiency. So we compare the execution time of programs that run with different threads number in Table 5, and the speedup is shown in Figure 10.

	No. threads				
Examples	1	4	8	16	32
Loop 1	3.99	1.24	0.75	0.74	0.75
Loop 2	1.81	6.81	9.07	9.16	9.16
Loop 3	0.44	0.46	0.45	0.46	0.45
Loop 4	0.45	0.24	0.38	0.38	0.38

Table 5: Execution time with different number of threads.

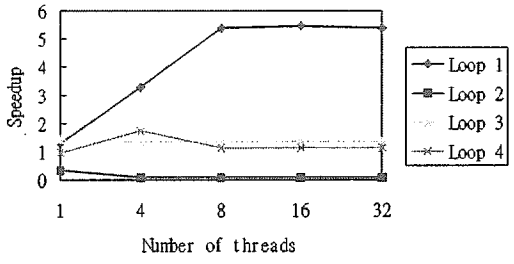

Figure 10: Speedup for different threads.

Where the loop examples are described as below:

- *Loop 1*, a single loop that can be run in parallel without any synchronization code, is shown in Figure 12(a).
- *Loop 2*, a single loop that has a loop carry-dependence, and must be performed with some synchronization code, is shown in Figure 12(b).
- *Loop 3*, a nested loop whose outer loop can be performed in parallel completely,, shown in Figure 12(c).
- *Loop 4* is a nested loop whose outer loop can be performed in parallel with synchronization code, is shown in Figure 12(d).

Since the Loop 1 and Loop 3 are DOALL loops, the number of threads plays a crucial part on speedup. The speedup will depend on the number of threads that can be executed concurrently. The number of threads depend on the ability of gaining the right to use the number of CPUs. In CONVEX SPP-1000, because there are eight CPUs in this machine, the suitable thread number is eight.

The Loop 2 and Loop 4 are DOACROSS loops, the synchronization overhead is the prime factor on speedup, the different number of threads cause the different speedup. In the Figure 7, the iteration five's ordered section can be executed only when the iteration four's ordered section is done, so, if the part of parallel code is too short, some CPUs are idle and the CPU's utilization is low, the speedup will be drop. If the ratio of parallelizing code segment to ordered section code segment is fixed, the number of threads is the factor of speedup. The maximal speedup depends on the ratio of parallelizing code to ordered code, the parallelability code must more than ordered code to gain the speedup. The speedup graph for different thread number of Loop 4 is shown in Figure 11.

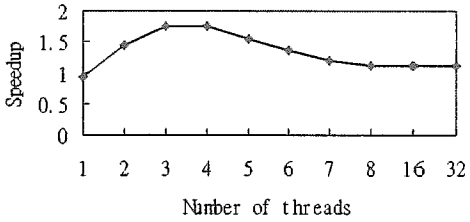

Figure 11: Speedup for different threads.

We use some other efficiency directive like `UNROLL` to measure the speedup as shown in Table 6.

Examples	Without unroll	Unroll	Speedup
Loop 1	0.57	0.43	1.33
Loop 2	0.80	0.43	1.86

Table 6: Execution time on SPP-1000.

Where the loop examples are described as below:

- *Loop 1* is a single loop, is shown in Figure 13(a).
- *Loop 2* is a nested loop, is shown in Figure 13(b).

The directive `UNROLL` only be used in the inner loop.

6 Conclusions

In this paper, we have provided a parallelism detector for CONVEX SPP-1000 system, it can point out the potential

```
      do 102 i=1, 1000000
        w(i)=tan(tan(2.0+i))+tan(tan(23.0))
102     continue
```

(a) Loop 1.

```
      do 102 i=1, 1000
        do 103 j=1, 1000
          w(i,i)=w(i,i-1)+tan(tan(23.0))
          w(i,i)=i+tan(tan(15.0))
103     continue
102   continue
```

(b) Loop 2.

```
      do 102 i=1, 1000000
        w(i)=w(i-1)+tan(tan(23.0))
        w(i)=i+tan(tan(15.0))
102   continue
```

(c) Loop 3.

```
      do 102 i=1, 1000
        w(i)=w(i-1)+tan(tan(23.0))
          do 104 j=1, 100
            w1(j)=tan(tan(j+3.0))
104       continue
102   continue
```

(d) Loop 4.

Figure 12: Example 1.

DOALL and DOACROSS loops of a program and translate the serial program into parallel form. In our previous researches, PPD can be easily extended to collect the characteristic of the loops and generated the parallel code for some machines. The K-Test based on the characteristic of program and the knowledge base to decide which test will be used, the testing algorithm library and the knowledge base can be easily modified to use the new testing algorithm is an effective approach to detect the loop. We combined the PPD and K-Test to construct a loop detector for CONVEX SPP-1000, it can automatically translate the tradition series programs to parallel programs with compiler directives. The combination of PPD and K-Test can detect more rich set of the loops which can be run in parallel. By the experiments on CONVEX SPP-1000 compiler directives, the compiler directives can improve the efficiency of execution time. So, this parallel detector achieves an automatical translation on CONVEX SPP-1000 and gain the high performance.

This new parallelism detector can be easily constructed for some other machines, if the machine has provided the compiler directives or some prompt parallel code, the codegen phase can be changed to produce the different prompt parallel codes for some other machines.

```
      do 102 i=1, 1000000
        w(i,i)=w(i,i-1)+tan(tan(23.0))
        w(i,i)=i+tan(tan(15.0))
102   continue
```

(a) Loop 1.

```
      do 102 i=1, 1000
        do 103 j=1, 1000
          w(i,i)=w(i,i-1)+tan(tan(23.0))
          w(i,i)=i+tan(tan(15.0))
103     continue
102   continue
```

(b) Loop 2.

Figure 13: Example 2.

References

[1] CONVEX Revision Information for *CONVEX Example Programming Guide*, First edition, 1994.

[2] Yi-Hsin Hsiao, *Design and Implementation of a Parallelism Detector for CONVEX SPP-1000 System*, M.S. Thesis, Dept. Comp. & Info. Sci., Nat'l. Chiao Tung Univ., Hsinchu, 1996.

[3] M. C. Hsiao, S. S. Tseng, C. T. Yang, and C. S. Chen, "Implementation of a portable parallelizing compiler with loop partition," in *Proc. ICPADS'94 Int. Conf. Parallel and Distributed Systems*, Hsinchu, Taiwan, R.O.C. pp. 333-338, Dec. 1994.

[4] X. Kong, D. Klappholz, and K. Psarris, "The i test: An improved dependence test for automatic parallelization and vectorization," *IEEE Trans. Parallel Distrib. Syst.*, 2(3):342-349, July 1991.

[5] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons, 1986.

[6] W. C. Shih, C. T. Yang, and S. S. Tseng, "Knowelwdge-based data dependence testing on loops," in *Proc. 1994 International Computer Symposium*, Hsinchu, Taiwan, R.O.C., pp. 961-966, Dec. 1994.

[7] E. Turban, *Expert Systems and Applied Artificial Intelligence*, Macmillan Publishing Co., New York, 1992.

[8] M. Wolfe, "High-Performance Compiler for Parallel Computing" pp. 137-162, Addison-Wesley Publishing, New York, 1995.

[9] C. T. Yang, C. T. Wu, and S. S. Tseng, "PPD: A practical parallel loop detector for parallelizing compilers on multiprocessor systems," to appear in *IEICE Trans. Information and Systems*, the previous version in *Proc. ICPADS'96*, 274-281, Japan, June 1996.

[10] H. P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing and ACM Press, New York, 1990.