ELSEVIER

Parallel Computing 23 (1997) 291–309

PARALLEL COMPUTING

Using knowledge-based techniques on loop parallelization for parallelizing compilers [1]

Chao-Tung Yang [2], Shian-Shyong Tseng *, Cheng-Der Chuang, Wen-Chung Shih

Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan 300, ROC

Received 10 July 1995; revised 22 March 1996

Abstract

In this paper we propose a knowledge-based approach for solving data dependence testing and loop scheduling problems. A rule-based system, called the K-Test, is developed by repertory grid and attribute ording table to construct the knowledge base. The K-Test chooses an appropriate testing algorithm according to some features of the input program by using knowledge-based techniques, and then applies the resulting test to detect data dependences for loop parallelization. Another rule-based system, called the KPLS, is also proposed to be able to choose an appropriate scheduling by inferring some features of loops and assign parallel loops on multiprocessors for achieving high speedup. The experimental results show that the graceful speedup obtained by our compiler is obvious.

Keywords: Parallelizing compiler; Data dependence testing; Loop parallelization; Parallel loop scheduling; Knowledge-based; Repertory grid analysis; Speedup

1. Introduction

In the past decade, multiprocessors have formed a major class of highly parallel and widely applicable machines. To achieve high speedup on such systems, it seems likely

* Corresponding author. Email: sstseng@cis.nctu.edu.tw.

[1] This work was supported in part by National Science Council of Republic of China under Grants No. NSC83-0408-E009-034 and NSC84-2213-E009-090. A preliminary version of this paper appeared in EURO-PAR'95, Lecture Notes in Computer Science, Vol. 966, pp. 417–428, Stockholm, Sweden, 1995.

[2] Email: ctyang@nspo.gov.tw.

0167-8191/97/$17.00 Copyright © 1997 Elsevier Science B.V. All rights reserved.
PII S0167-8191(96)00090-7

for task to be decomposed into several sub-tasks which can be executed on different processors in parallel. Parallelizing compilers [2-4,17,21,25], analyze sequential programs to detect hidden parallelism and use this information for automatic restructuring of sequential programs into parallel sub-tasks on multiprocessors by using loop scheduling algorithms [10,16,20]. So, it has become an important issue to develop parallelizing compiling techniques which exploit potential power of multiprocessors. In particular, loops are such a rich source of parallelism that their parallelization would lead to considerable improvement of efficiency on multiprocessors [4,7,25]. Therefore, we investigate the possibility of solving the problem on two fundamental phases, data dependence testing and parallel loop scheduling on loops, in parallelizing compilers.

In brief, the *data dependence testing* problem is that of determining whether two references to the same array within a nest of loops may reference to the same element of that array [8,14,15,13,18]. Traditionally, this problem has been formulated as *integer programming*, and the best integer programming algorithms are $O(n^{O(n)})$ where n is the number of loop indices. Obviously, these algorithms are too expensive to use. For this reason, a faster, but not necessarily exact, algorithm might be more desirable in some situations.

In this paper, we propose a new approach by using knowledge-based techniques for data dependence testing [6]. A rule-based system, called the *K-Test* [19], is developed by repertory grid and attribute ordering table to construct the knowledge base. The K-Test can choose an appropriate test according to some features of the input program by using knowledge-based techniques [11], and then apply the resulting test to detect data dependences on loops for parallelization. Furthermore, as for system maintenance and extensibility, our approach is obviously superior to others, for example, if a new testing algorithm or testing technique is proposed, then we can integrate it into the K test easily by adding knowledge base and rules.

Another fundamental phase, parallel loop scheduling, is a method that schedules the DOALL loops on multiple processors [10,16,20]. In a shared-memory multiprocessors, scheduling decision can be made either statically at compile time or dynamically at runtime. *Static scheduling* is usually applied to uniformly distributed iterations on processors. However, it has the drawback of load imbalancing when the loop style is not uniformly distributed or the loop bounds could not be known at compile time. In contrast, *dynamic scheduling* is more suitable for load balancing; however, the runtime overhead must be taken into consideration. Traditionally, the parallelizing compiler dispatches the loop by using only one scheduling algorithm, maybe static or dynamic. However, a program has the different kind of loops including uniform workload, increasing workload, decreasing workload, and random workload, every scheduling algorithm can achieve good performance on different loop styles [20].

To reduce the overhead and enhance the load balancing, the knowledge-based approach becomes another solution to parallel loop scheduling. In this paper, another rule-based system, named *Knowledge-Based Parallel Loop Scheduling* (KPLS), is also developed by repertory grid analysis, which can choose an appropriate scheduling according to some features of loops and system status, and then apply the resulting algorithm to assign DOALL loops on multiprocessors for achieving high speedup. The experimental results show that the graceful speedup obtained by using KPLS in our

compiler is obvious. In the near future, we wish to study whether knowledge-based approach may be applied to guide the wide variety of loop transformation [1] for parallelism in parallelizing compilers.

The rest of this paper is organized as follows. In Section 2, the data dependence testing and parallel loop scheduling problems will be reviewed. Next, our new approach for data dependence testing and loop scheduling are proposed in Section 3 and Section 4, respectively. Section 5 gives some experimental results. Finally, Section 6 draws a conclusion and indicates future work.

2. Background

2.1. A review of data dependence testing

A data dependence is said to exist between two statements S_1 and S_2 if there is an execution path from S_1 to S_2, both statements access the same memory location and at least one of the two statements writes the memory location [8]. There are three types of data dependences:

- *True (flow) dependence* (δ) occurs when S_1 writes a memory location that S_2 later reads.
- *Anti-dependence* ($\bar{\delta}$}) occurs when S_1 reads a memory location that S_2 later writes.
- *Output dependence* (δ_o) occurs when S_1 writes a memory location that S_2 later writes.

Data dependence testing is the method used to determine whether dependences exist between two subscript references to the same array in a nested loop. The index variables of the nested loop are normalized to increase by 1. Suppose that we want to decide whether or not there exists a dependence from statement S_1 to S_2. A model of loop is shown in Fig. 1. Let $\alpha=(\alpha_1,\alpha_2,\ldots,\alpha_n)$ and $\beta=(\beta_1,\beta_2,\ldots,\beta_n)$ be the integer vectors of n integer indices within the range of the upper and lower bounds of the n loops in the example. There is a *dependence* form S_1 to S_2 if and only if there exist α and β, such that α is lexicographically less than or equal to β and the dependence equations are satisfied as $f_i(\alpha)=g_i(\beta)$, for $1 \le i \le m$.

```
      DO I1 = L1,U1
        DO I2 = L2,U2
          ...
          DO In = Ln,Un
S1 :        A(f1(I1,I2,...,In),...,fm(I1,I2,...,In)) = ...
S2 :        ... =A(g1(I1,I2,...,In),...,gm(I1,I2,...,In))
          ENDDO In
          ...
        ENDDO I2
      ENDDO I1
```

Fig. 1. The model of nested loop for data of the K-Test.

In this case, we say that the system of equations is *integer solvable* with the loop-bounds constraints. Otherwise, the two array reference patterns are said to be *independent*. Basically, the data dependence testing problem is equivalent to integer programming if all the f_i's and g_i's are linear functions. The data dependence tests can be classified into three classes: single dimensional tests (e.g., *GCD Test*, *Banerjee Test* [25], and *I Test* [13]), multiple dimensional tests, (e.g., *Extended-GCD Test* [2], *λ Test* [14], *Power Test* [22], and *Ω Test* [18]), and classification tests [8,15]. We find these two papers [8,15] are similar to our approach (*K-Test*) in some aspects. Both of them collect a small set of test algorithms, and try to use them to solve the problem both efficient and exact in practical cases. However, our approach is different from theirs in essence.

- *Practical Test* [8]: The test is based on classifying pairs of subscript variable references. The major difference between the Practical test and our approach is that the Practical test is essentially designed for practical input cases, and its strategy is fixed. However, our approach is not constraint to some kind of input cases.
- *MHL Test*: The major difference between the MHL test and our approach is that the MHL test is a cascaded method; that is, the Extended-GCD test is tried first; if it fails, the next test is applied, and so on. However, our approach uses only the appropriate after the conclusion is drawn.

2.2. A review of parallel loop scheduling

A loop is called as a DOALL loop if there is no data dependence among all iterations by using data dependence testing, i.e., iterations can be executed in any order or even simultaneously. *Parallel loop scheduling* is used to assign a DOALL loop into each processor as even as possible. In a shared-memory multiprocessor system, there are two kind of parallel loop scheduling strategies which can be made either statically at compile time or dynamically at runtime. *Static scheduling* may be applied when the loop iterations take roughly the same amount of execution time, and the compiler must know how many iterations are run in advance. However, it may perform unacceptable when the loop style is not uniformly distributed or the loop bounds can not be known at compile time. *Dynamic scheduling* adjusts the schedule during execution, so we use it whenever it is uncertain how many iterations to be run, or each iteration takes different amount of execution time, due to a branch statement inside the loop. Dynamic scheduling is more suitable for load balancing between processors, but the runtime overhead and memory contention must be considered.

Various scheduling methods have been developed, In the following, the review of these scheduling algorithms will be given. We use N and P to denote the number of iterations and the number of processors, respectively.

- *Static Scheduling*: Traditional static scheduling makes the scheduling decision at compile time, and uniformly distributes loop iterations into each processor. In static scheduling, the N iterations are divided into $\lceil N/P \rceil$ rounds. Each round is assigned to one processor. Thus, the number of iterations and available processors must be known at compile time.
- *Self-Scheduling* (SS): It is the most easy and straight forward dynamic loop scheduling algorithm. Whenever a processor is idle, one iteration is allocated to it. This

algorithm can achieve good load balancing, but also introduce too much overhead for accessing shared index variables.

- *Chunk Self-Scheduling* (CSS) [20]: Instead of allocating one iteration to the idle processor as self-scheduling, CSS(k) allocates k iterations each time, where k, called the chunk size, is fixed and can be specified by programmers either or compilers. We also adopted the CSS/K, which is a modified version of CSS, where K denotes the number of chunks [9].
- *Guided Self-Scheduling* (GSS) [16,17]: GSS can dynamically change the number of iterations assigned to each processor. More specifically, the next chunk size is determined by dividing the number of the remaining iterations of a DOALL loop by the number of processors. The property of decreasing chunk size implies its efforts to achieve load balancing and reduce scheduling overhead. By allocating large chunks at the beginning, the frequency of mutually exclusive for accessing those shared index variables can be reduced. The small chunks at the end of a loop serve to balance the workload across all processors.
- *Factoring* [10]: In some cases, GSS might assign too much workload to the first few processors, so that the remaining iterations are not sufficiently time-consuming to balance the workload. This situation arises when the initial iterations of a loop are much more time-consuming that later iterations. The Factoring algorithm addresses this problem. The allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because Factoring allocates a subset of the remaining iterations in each phase, it balances load better than GSS when the computation times of loop iterations vary substantially. In addition the synchronization overhead of Factoring is not significant larger than GSS.
- *Trapezoid Self-Scheduling* (TSS) [20]: It tries to reduce the need for synchronization, while still maintaining a reasonable balance in load. This algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $N/2P$, and consecutive chunks differ in size $N/8P^2$ iterations. The difference in the size of successive chunks is always a constant in TSS, whereas it is a decreasing function both used in GSS and Factoring.

All of the algorithms are under some assumption, so that they will be suitable in some cases. After preliminary observation, we think concepts of knowledge-based system should be useful to parallelizing compiling because the compiling process is no more deterministic and many domain knowledge may be needed to solve loop scheduling efficiently.

3. Using knowledge-based techniques for data dependence testing

3.1. Knowledge-based approach

Knowledge-based systems are systems that depend on a vast base of knowledge to perform difficult tasks. The knowledge is saved in a knowledge base separately from the

inference component. This makes it convenient to append new knowledge or update existing knowledge without recompiling the inferring programs. The *rule-based* approach is one of the commonly used form in many knowledge-based systems. The primary difficulty in building a knowledge base is how to acquire the desired knowledge. To ease acquisition of knowledge, one primary technique among them is *Repertory Grid Analysis* (RGA) [12]. RGA is easy to use, but it suffers from the problem of *missing embedded meanings* [11]. For example, when a doctor expresses the features of catching a cold are headache, cough and sneeze, he means if a person catches a cold, he may has those features. However, in RGA, a person is not considered to catch a cold except that he gets all of the features. To overcome the problem, the concept of *Attribute Ordering Table* (AOT) is employed to elicit embedded meanings by recording the importance of each attribute to each object [11].

A knowledge-based system is composed of two parts: the *development environment* and the *runtime environment*. The former is used to build the knowledge base, while the latter is used to solve the problem. In our paper, the *development environment* is not discussed. The *runtime environment* contains three components, 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.
- *Applied algorithm libraries*: The libraries collect several representative data dependence tests and parallel loop scheduling algorithms for solving dependence problems and scheduling DOALL loops, respectively.

3.2. The anatomy of the K-test

The processes of knowledge-based data dependence testing can be described as follows. First, the input, a set of dependence equations, is fed into the inference component. Then, the inference component reasons about knowledge and draw a conclusion, a test. Finally, the resulting test is applied to detect dependence relations for loop parallelization, and generate the answer whether the loop is parallelizable or not. An implementation, called the *K-Test*, is proposed to demonstrate the effectiveness of the new approach. The K-Test is a rule-based system. The primary reason we choose a rule-based system is that this type of system is easy to understand; in addition, rule-based inference tools are widely available, which simplify the work of implementation.

The organization of the K-Test is shown in Fig. 2 that the three components are replaced by actual software. We describe them briefly.

- *Knowledge base*: The knowledge base is constructed as a rule base, i.e., the knowledge is expressed in the form of production rules. These rules can be coded by hand or generated by a translator. In our K-Test, the latter is adopted. A translator,

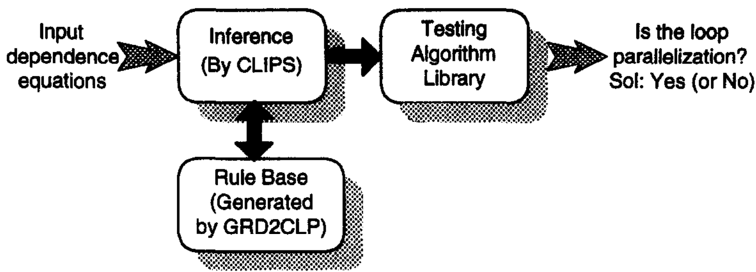

Fig. 2. Components of the K-Test

GRD2CLP, is utilized to translate the repertory grid and attribute ordering table to CLIPS's production rules.

- *Inference component*: An expert system shell, called CLIPS [7], is used as the inference component. CLIPS, a forward reasoning rule-based tool, is very efficient, and does not increase the execution time of the K-Test too much.
- *Testing algorithm library*: We include four tests in the library. There are GCD test, Banerjee test, I test and Power test for solving the data dependence problem.

It should be noted that the knowledge base and the testing algorithm library shown in Fig. 2 are flexible; that is, they are not fixed. You can modify these two components so long as the efficiency and precision of the system are retained. The repertory grid of the K-Test contains four attributes and four objects which are four existing data dependence tests. The four attributes of the K-Test are described below:

- Unity_Coef: whether the coefficients of variables are 1, 0, or -1 or not.
- Bound_Known: whether the loop bounds are known or not.
- Multi_Dim: whether the array reference is multi-dimensional or not.
- Few_Var: whether the number of variables in the equation is small or not.

In order to elicit the embedded meanings of RGA of the K-Test, we construct the AOT. The RGA/AOT of the K-Test is shown in Table 1. The process is described in dialog form. For example,

Q: If Bound_Known is not equal to 5, is it possible for the Banerjee test to be applied?A: No.

The answer means that Bound_Known dominates the Banerjee test, and hence AOT[Bound_Known, Banerjee] = 'D'. In AOT, large integer number implies the attribute being more important to the object (e.g. $2 > 1$).

Table 1
The RGA/AOT of the K-Test

	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

3.3. The algorithm of the K-test

We now summarize the discussion of the K-Test into an algorithm. The algorithm consists of two phases.

Algorithm: K-Test

Input:

$(a_0^1, a_1^1, \ldots, a_n^1, M_1^1, N_1^1, \ldots, M_n^1, N_n^1,$

$\ldots,$

$a_0^m, a_1^m, \ldots, a_n^m, M_1^m, N_1^m, \ldots, M_n^m, N_n^m,$

Unity_Coef,Bound_Known,Multi_Dim,Few_Var)

Output:
True: the input is integer solvable.
or False: the input is not integer solvable.
or Maybe: the input may be integer solvable.
Phase 1: calling CLIPS to draw a conclusion, that is, the most suitable dependence test.
Phase 2: calling the corresponding testing algorithm to check for data dependence.

4. Using knowledge-based techniques for parallel loop scheduling

If the parallelizing compiler can analyze a loop's attributes such as loop style, loop bound, data locality, etc.; then the suitable scheduling algorithms for the particular case should be applied. This leads to select scheduling algorithms by using knowledge-based approach. The processes of knowledge-based loop scheduling method can be described as follows. First, the compiler can get some attributes about a loop by parsing input program. Then, the inference component reasons about knowledge and draw a conclusion, a parallel loop scheduling. Finally, the resulting scheduling is applied for loop partition.

4.1. The features of parallel loop scheduling

There are many factors influence the selection of a loop scheduling algorithm, including number of iterations, number of processors, loop style, a start time of each processor, synchronization overhead in a machine, and the easiness of implementation for a algorithm, which will be discussed more detail as follows.

- The number of iterations and processors must be known at compile time if we want to use the static scheduling algorithm. This is a necessary condition for static scheduling since the scheduling decision is made at compile time. For dynamic scheduling methods, these two factors can be omitted.
- However, loops can be roughly divided into four styles as shown in Fig. 3 including uniform workload, increasing workload, decreasing workload, and random workload.

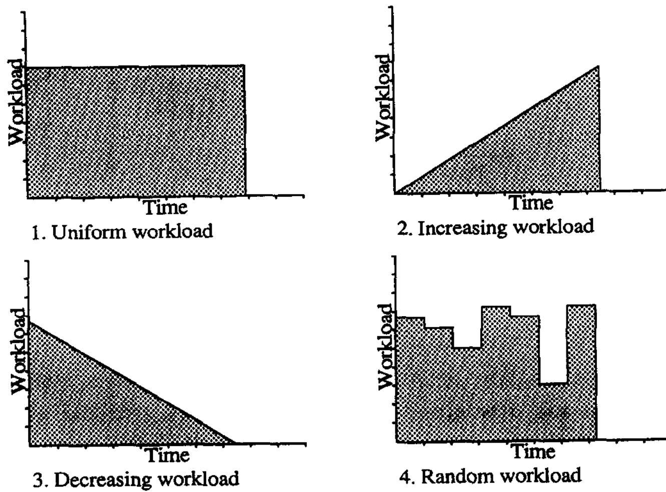

Fig. 3. Four different loop analysis.

These four styles of loops are most common in programs, and can cover the most cases. The static scheduling method is only suitable for the first style; dynamic scheduling are suitable for all loop styles except that GSS is not good for the second style. It is possible for GSS to allocate too many iterations on a processor at the beginning when applying the second loop style; that is, GSS may cause load imbalancing.

- The starting time of each processor is also important. If the starting times are unequal, static scheduling and CSS may not perform well. In contrast, dynamic method is good when applying to this unequal start time condition.
- The synchronization primitives provide by a system are close related to the synchronization overhead introduced by scheduling algorithms. If a system provides little synchronization primitives, the synchronization overhead will be high. Thus, we classify the synchronization overhead into four levels. Level one is none overhead, level two is little overhead, level three is fair overhead, and level four is high overhead. SS is not well when the synchronization overhead is not none, since it may introduce too much overhead. GSS is also not well when the overhead is fair or high.
- If there are two scheduling algorithms suitable for a particular loop, we should always choose the one that is easier to implement. So we also give an attribute about the easiness of implementation in our expert system approach.

With above attributes, we may choose a scheduling algorithm for a parallel loop with particular attributes. This is the basic idea of using knowledge-based approach to make the selection of scheduling algorithms.

4.2. The anatomy of KPLS

In this section, we describe our new method, named *Knowledge-Based Parallel Loop Scheduling* (KPLS). We propose this method using *knowledge-based*, because it is easy

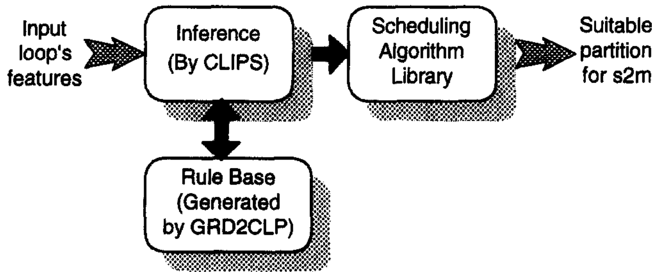

Fig. 4. Components of the KPLS.

to understand, implementation, maintenance and extension. This approach has great flexibility as we can add new algorithms to the repertory grid and attribute ordering table, and then use the conversion tool to convert tables to CLIPS rules. We do not need any modification in CLIPS source.

The organization of the KPLS is shown in Fig. 4 that consists three components. We describe the components of KPLS briefly in the following:

- *Knowledge base*: The knowledge base is constructed as a rule base, i.e., the knowledge is expressed in the form of production rules. We also use GRD2CLP to translate the repertory grid and attribute ordering table to CLIPS's production rules.
- *Inference component*: CLIPS is used as the inference component, which is very efficient for inferring, and does not increase the execution time of our KPLS too much.
- *Scheduling algorithm library*: There are six scheduling algorithms in the library including static scheduling, SS, CSS, GSS, Factoring, and TSS. It is also the advantage of expert system; whenever, we can easily modify the rules and adding the new scheduling strategy flexibly.

The repertory grid and attribute ordering table of KPLS are shown in Table 2. "X" means that the attribute has no relation with the object. There are six algorithms and five attributes in both tables. We describe these attributes as follows:

- Loop_Style: means the different styles of loop (1: uniform workload, 2: increasing workload, 3: decreasing workload, or 4: random workload).
- Start_Time: means whether the starting time of each processors is equal or not, influencing the execution time of loop.
- Loop_Bound: means whether the loop bounds are known or not in compile time.

Table 2
The RGA/AOT of the KPLS

	Static	SS	CSS	GSS	TSS	Factoring
Loop_Style	{1}/D	X/X	{1}/D	{2,4}/D	X/X	X/X
Start_Time	YES/D	X/X	YES/D	X/X	X/X	X/X
Loop_Bound	YES/D	X/X	NO/D	X/X	X/X	X/X
Overhead	X/X	{0}/D	{1,2,3}/D	{1}/2	{2,3}/1	{2,3}/1
Easy	X/X	X/X	X/X	NO/2	X/X	NO/2

- Overhead: means the different overhead of synchronization primitives on system (0: none, 1: little, 2: fair, or 3: high).
- Easy: means whether the implementation of algorithm is easy or not.

Six rules can be generated from the repertory grid, one rule per column. We list the first rules below:

```
(defrule rule_01
  (LoopStyle known ?v1? cf1)
  (StartTime known ?v2? cf2)
  (LoopBound known ?v3? cf3)
  (test (and
    (eq ?v1 1)
    (eq ?v2 yes)
    (eq ?v3 yes)
  ))
=g
  (bind ?fcf (min ?cf1 ?cf2 ?cf3))
  (assert(goal Static =(* 0.80 ?fcf)))
  (assert(phase print_goal)))
```

4.3. The Algorithm of the KPLS

In this section, we describe our algorithm of KPLS. The algorithm consists of three phases.

Algorithm: KPLS

Input:The following information can be obtained from the input file.
1. What kind of loop style? (1–4 styles)
2. Are the start time of processors roughly equal? (Yes/No)
3. Is the loop bound known during compiler time? (Yes/No)
4. What is the synchronization overhead level? (None/Low/Fair/High)
5. Use easy-to-implement methods only? (Yes/No)

A *certainty factor* (CF) [11] value for each question to express the question's importance is given.
Output:What kind of loop scheduling strategy will be applied. If there are more than one suggestion, the one with maximal CF value will be chosen.
Phase 1: Get the loop attributes from parallelism detector.
Phase 2: Call CLIPS to draw a conclusion by using rules; that is, the most suitable loop scheduling method.
Phase 3: s2m [9] uses the appropriate loop scheduling to partition the DOALL loop on multiprocessors.

4.4. An example

Now, we give an example to illustrate the algorithm. The program segment of adjoint convolution is shown in the following:

```
DO I=1,100
S1 :   A[I]=B[I]*2
S2 :   C[I]=A[I]+D[I]
ENDDO
```

Fig. 5. A program segment with loop-independent dependence.

```
DOALL 19 I=1, N*N
  DO 29 K=I, N*N
    A(I)=A(I)+X*B(K)*C(I-K)
  29 CONTINUE
19 CONTINUE
```

The following attributes can be obtained from parallelism detector during Phase 1.

1. The workload of loop is decreasing, that means the loop style is the second.
2. Because the workload of this program segment is uniform, the starting time is roughly equal.
3. The loop bound is unknown during compile time.
4. On our share-memory architecture, we assume the synchronization is fair.
5. The easy-to-implement method is chosen.

Phase 2, the KPLS uses those attributes, and determines that the TSS is the most suitable algorithm. In Phase 3, the TSS is invoked; that is, TSS is suitable for s2m, then s2m partition this loop by using TSS.

4.5. The relationships between K-Test and KPLS

For example, consider the program segment as shown in Fig. 5. Although S_2 is true dependent on S_1, the dependence is restricted in each iteration, that is loop-independent dependence. Our K-Test is used to determine whether dependences exist between two subscript references to the same array in the nested loop. Therefore, all of the iterations in the loop can be executed in parallel, called as DOALL loop. Another example is shown in Fig. 6, S_2 is true dependent on S_1, the dependence is occurred across iterations, that is loop-carried dependence, called as DOACROSS loop, whose iterations are either executed sequentially, or in parallel through the enforced synchronization instructions within the loop body.

If a loop can be executed in parallel, we want to break this loop down to a set of tasks on different processors. As we know, task granularity, which is an important issue in loop partitioning heavily influences load balancing. Therefore, a good loop-partition-

```
DO I=2,100
S1 :   A[I]=B[I]*2
S2 :   C[I]=A[I-1]+D[I]
ENDDO
```

Fig. 6. A program segment with loop-carried dependence.

ing algorithm will achieve better load balancing with only a small overhead. Once KPLS gets information from the loop and system, it can make good and correct scheduling decisions for that loop, while other scheduling methods try only one scheduling approach to solve all kinds of loops. Since no single loop scheduling algorithm performs well across all applications on multiprocessor systems, our new approach provides a good way to make compilers more flexible efficient, and intelligent in loop scheduling for achieving high parallelism.

5. Experiments

5.1. Our platform: AcerAltos 10000 system

Our experiments are run on AcerAltos 10000 system, which is a PC-based shared-memory, symmetric multiprocessor computer designed for departmental client/server environments. The system includes up to four i486-DX (33 MHz) CPUs, an 8 K internal cache and a 128 K external cache per CPU, 32 MB shared-memory, and a 64 Bit high-speed frame bus. Due to the symmetric architecture, computation tasks can easily be distributed to any available processor. This means that balanced loading of all processors can be achieved.

5.2. Operating system and threads

The operating system run on our target machine was OSF/1 [5]. Many of the kernel services provided by OSF/1 were derived from the Mach (version 3.0) operating system. Mach was designed from the ground up to support symmetric multiprocessing and distributed computing. It also contains a native threads facility, allowing separate threads of a single application to be executed simultaneously.

OSF/1 provides P Threads package which is a set of low level, language independent primitives for manipulating threads of control. The package, a runtime library, provides P Thread function calls which allow parallel programming in C under the operating system. These functions provide multiple threads of control for parallelism, shared variable, mutual exclusion for critical sections, and condition variables for synchronization of threads. Therefore, OSF/1 is used to develop our parallel compiler. Besides, our parallelizing compiler is very easy to be ported to other operating systems [9]. In the following, the thread related data types and functions used in our compiler are described.

5.3. Integrating K-Test and KPLS in PFPC

We integrate K-Test and KPLS in PFPC [9,24] to generate the efficient object codes for multiprocessors. The K-Test is used to treat the data dependence relations and then restructure a sequential FORTRAN source program into a parallel form, i.e., if a loop can be parallelized, then parallelism detector (K-Test) converts it into DOALL loop. In the previous version of our compiler, Parafrase-2 (p2fpp) is used to treat the data

```
1:  2 3
2:  1 1 10
3:  -1 1 10
4:  0 1 10
5:  1
6:  0 1 10
7:  -1 1 10
8:  1 1 10
9:  -1
```

Fig. 7. The input format of testing algorithms.

dependence analysis [23,19]. For improving the capacity of loop partition module in s2m [9], the KPLS is used, instead of the previous version, to build an intelligent loop scheduling method. Because the restriction of the OS scheduling, the system call for binding any thread onto the appropriate processor is not available and only dynamic scheduling is employed. We can only partition loops and encapsulate with data into threads by s2m and let the OS dynamically choose the threads to run on multiprocessors. The experiments only concerned the performance of KPLS in PFPC.

5.4. Experiments with K-Test

We have coded the GCD test, the Banerjee test, the I test, the Power test and the K-Test in C programming language. Experiments in previous work usually implement their methods in a prototype parallelizing compiler, such as Parafrase-2 [17,23], Tiny [18], etc. Nevertheless, we cannot afford such a large project. Consequently, we decide to construct a main program which reads the input equations from a file, then calls CLIPS, and finally invokes the tests. Hence, we examine all input codes, select the representative array subscripts, and encode them into an input file by parallelism detector. For example, the simultaneous equations

$$x - y = 1, \quad -y + z = -1, \qquad \text{where } 1 \le x, y, z \le 10$$

is encoded into the form of Fig. 7. The first line contains two numbers: the first represents the number of equations, and the second refers to the number of variables. Hence, (2 3) means there are 2 equations and 3 variables. Lines 2–5 represent the first equation and lines 6–line 9 mean the second equation. The three numbers in line 2 refer to the coefficient, the lower bound and the upper bound, respectively.

We have performed experiments on two numerical packages, EISPACK and LINPACK. EISPACK is a collection of subroutines computing the eigenvalues of matrices. LINPACK is a collection of Fortran subroutines which analyze and solve various systems of simultaneous linear algebraic equations. Because of their systemization and representativity, the packages have been widely adopted as benchmark programs [8,14].

EISPACK has 75 subroutines, which contain about 70000 pairs of array references. LINPACK has 51 subroutines, which contains about 5000 pairs of array references [8]. In our experiments, we consider only possible true dependent references. Table 3 lists the number of lines, subroutines and array pairs tested.

Table 3
Program characteristics for practical data

	lines	subrs	array pair tested
EISPACK	11519	75	211
LINPACK	7427	51	106

Table 4 shows the usage and success frequencies of the dependence tests for the two packages. The notations are similar to that in [8]. ''A'' denotes the number of times the test is applied; ''S'' denotes the number of times the test succeeds in determining dependences; ''I'' denotes the number of times the test proves the pair is data independent.

We know that practical array subscripts are usually simple, with unity coefficiencies and few index variables. Therefore, the GCD test seems sufficient for practical data and the Power test and the K-Test are not significantly superior to the GCD test. Note that in LINPACK and EISPACK, the loop bounds are all parameters and are unknown values. This is why the Banerjee test can not be applied to the two packages.

The execution time of the K-Test is significantly longer than the GCD test, the Banerjee test and the I test. The overhead results from the cost of inferential process and the usage of the expensive Power test. However, the time required by the K-Test is nearly the same as that of the Power test in the worst case. In fact, the amount of time is relatively small compared with the whole compiling process.

5.5. Experiments with KPLS

We show the performance gained by using PFPC's KPLS on the following four examples of different loop styles.

The first example is matrix multiplication, where the two outer loops can be parallelized. Since the example is highly load balanced, every iteration of the outermost DO loop takes constant time to execute; this kind of loop is called *uniform workload.* The matrix size is 600×600.

The second example is adjoint convolution, which exploits significant load imbalance; only the outer loop can be parallelized and the ith iteration of it takes $O(N^2 - i)$

Table 4
Application/Success/Independence frequencies for practical data

	EISPACK			LINPACK		
	A	S	I	A	S	I
GCD	206	204	0	83	83	0
Banerjee	0	0	0	0	0	0
I	206	204	0	83	83	0
Power	206	206	2	83	83	0
K	206	206	2	83	83	0

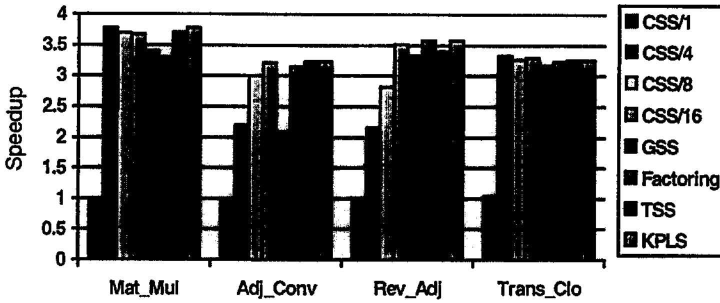

Fig. 8. The speedup for individual programs.

time to execute. As i increases from one to N^2, the workload decreases from $O(N^2)$ to $O(1)$. This kind of loop is called *decreasing workload.* We choose the problem size to be 150×150.

The third example is reverse adjoint convolution, which also exploits significant load imbalance; only the outer loop can be parallelized and the ith iteration of it takes $O(i)$ time to execute. As i increases from one to N^2, the workload also increases from $O(1)$ to $O(N^2)$. This kind of loop is called *increasing workload.* The problem size is 150×150.

The fourth example is transitive closure. The characteristic of this program is that the workload is dependent on the input data. Each iteration takes either $O(1)$ or $O(N)$ time. This kind of loop is called *random workload.* We selected different matrix sizes for testing alone, then combined four programs using 1000×1000 and 500×500, respectively.

There are two parts to each experiment: the first part concerns the execution time and speedup rate of each program, and the other is a combination of all four programs. The speedup rate of the four programs in Fig. 8 (see also Table 5) show that GSS performed poorly with a decreasing workload as in adjoint convolution. CSS/4 is suitable for uniform workloads like matrix multiplication, and Factoring is suitable for reverse adjoint convolution. We also adapted the CSS/l [9], which is a modified version of CSS, where l means the number of chunks. Among the scheduling algorithms, none is suitable for all tasks. KPLS can choose an appropriate schedule and obtain good results for all programs except transitive closure. In transitive closure, our approach does not choose the fastest one, CSS/4, but chooses TSS, because the imbalanced workload in

Table 5
The execution time of individual programs (sec)

	Serial	CSS/1	CSS/4	CSS/8	CSS/16	GSS	Factoring	TSS	KPLS
Mat_Mul	854.46	848.18	225.96	230.92	232.73	250.94	259.32	230.74	as CSS/4
Adj_Conv	769.50	776.38	350.80	256.70	239.90	369.54	245.01	237.68	as TSS
Rev_Adj	604.51	632.37	281.01	215.19	171.63	182.01	168.93	177.57	as Factoring
Trans_Col	2310.96	2221.81	695.73	708.83	702.60	730.30	713.62	710.68	as TSS

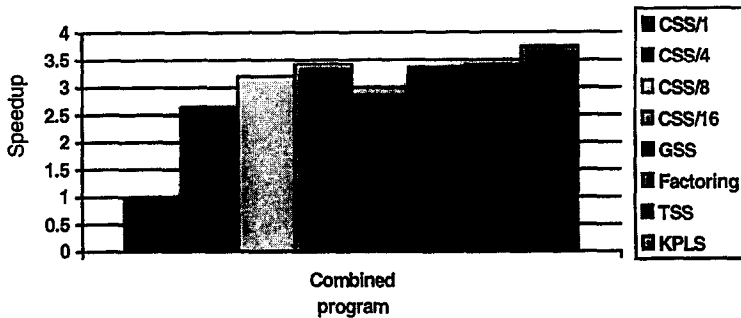

Fig. 9. The speedup for the combination of four programs.

this program is not so obvious, because the control flow is related to the input data, and because the matrix size is 500 × 500, which is divided exactly by the number of CPUs. In most cases, KPLS makes a better choice than other scheduling even for a single loop.

Fig. 9 (see also Table 6) shows the speedup for the big program integrating all four programs. Conventionally, every scheduling algorithm uses only one method through for an entire program. However, KPLS can always choose an appropriate scheduling algorithm according to the loop behaviors in a program. In second part of the experiment, KPLS chooses a different style of loop scheduling for each loop in the combined program. For example, according to the loop behavior, KPLS selected TSS for the adjoint convolution part of the combined program and CSS/4 for matrix multiplication, instead of only selecting one scheduling method. We were concerned about the loop runtime cost. During execution, selecting a good loop scheduling algorithm by considering the runtime cost is important; once the compiler specifies the right loop schedule, the program can save execution time. Furthermore, KPLS spends only a little time on knowledge-based inference. Since no single scheduling algorithm performs well across all applications, our method is able to make compilers more flexible and efficient at loop scheduling.

6. Conclusions and further directions

In this paper we have proposed a new approach by using knowledge-based techniques, which integrates existing data dependence testing algorithms and loop scheduling algorithms to make good use of their advantages for loop parallelization. A rule-based system, called the K-Test, was developed by RGA and AOT to construct the

Table 6
The execution time for the combination of four programs (sec)

	Serial	CSS/1	CSS/4	CSS/8	CSS/16	GSS	Factoring	TSS	KPLS
All	2185.59	2167.99	872.34	683.85	636.38	763.67	651.52	644.43	582.47

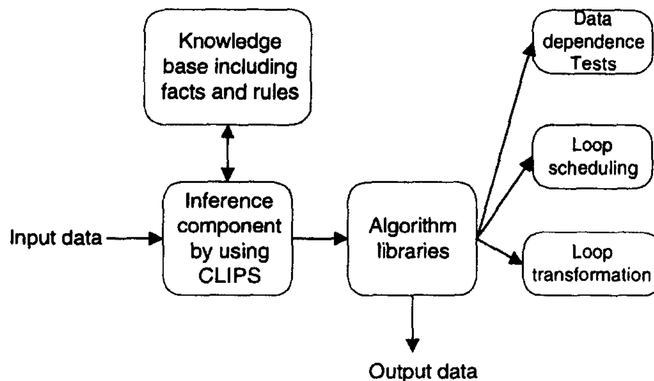

Fig. 10. Components of our approach.

knowledge base. The K-Test could choose an appropriate testing algorithm by knowledge-based techniques, and then apply the resulting test to detect data dependences on loops. Another rule-based system, called the KPLS, was also developed by RGA and AOT, which was embedded in our s2m, that could choose an appropriate scheduling and then apply the resulting algorithm for assigning parallel loops on multiprocessors to achieve high speedup. The experiments have shown that the KPLS can apply more suitable loop scheduling strategy. Once we choose the right method for loop scheduling, the program can save more execution time. The experimental results also have shown that the graceful speedup obtained by our compiler is obvious. Furthermore, as for system maintenance and extensibility, our approach is obviously superior to others. In addition, we are going to study whether knowledge-based approaches may be applied to guide the wide variety of loop transformation for parallelization in parallelizing compilers. The new model which using knowledge-based techniques for loop parallelization contains three components as shown in Fig. 10.

As a final goal, a high-performance and portable FORTRAN parallelizing compiler will be constructed at NCTU. We believe that our research will yield more insights into developing high-performance parallelizing compilers for multiprocessors running under multithreaded operating systems.

Acknowledgements

We would like to thank the anonymous reviewers for suggesting of improvements, and offering of encouragements.

References

[1] D.F. Bacon, S.L. Graham and O.J. Sharp, Compiler transformations for high-performance computing, *ACM Computing Surveys* 26 (4) !994) 345–420.

[2] U. Banerjee, *Dependence Analysis for Supercomputing* (Kluwer Academic Publishers, Norwell, MA, 1988).

[3] U. Banerjee, R. Eigenmann, A. Nicolau and D.A. Padua, Automatic program parallelization, *Proc. IEEE* 81 (2) (1993) 211–243.
[4] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D.A. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu and S. Weatherford, Polaris: The next generation in parallelizing compilers, Technical Report no. CSRD-1375, Cntr. for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, 1994.
[5] J. Boykin, D. Kirschen, A. Langerman and S. LoVerso, *Programming under Mach* (Addison Wesley, Reading, MA, 1993).
[6] B.M. Chapman and H.M. Herbeck, Knowledge-based parallelization for distributed memory systems, Proc. of the First International ACPC Conference on Parallel Computing, Salzburg, Austria (Springer, Berlin, 1991) 77–89.
[7] J.C. Giarratano and G. Riley, *Expert Systems: Principles and Programming* (PWS-Kent, Boston, 1993).
[8] G. Goff, K. Kennedy and C.W. Tseng, Practical dependence testing, in: *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, Toronto, Canada (1991) 15–29.
[9] M.C. Hsiao, S.S. Tseng, C.T. Yang and C.S. Chen, Implementation of a portable parallelizing compiler with loop partition, in: *Proc. 1994 ICPADS*, Hsinchu, Taiwan, ROC (1994) 333–338.
[10] S.F. Hummel, E. Schonberg and L.E. Flynn, Factoring: A method for scheduling parallel loops, *Comm. ACM* 35 (8) (1992) 90–101.
[11] G.J. Hwang and S.S. Tseng, EMCUD: A knowledge acquisition method which captures embedded meanings under uncertainty, *Internat. J. Man–Machine Studies* 33 (1990) 431–451.
[12] G.A. Kelly, *The Psychology of Personal Constructs, Vol. 1* (W.W. Norton, New York, 1955).
[13] X. Kong, D. Klappholz and K. Psarris, The i test: An improved dependence test for automatic parallelization and vectorization, *IEEE Trans. Parallel Distributed Systems* 2 (3) (1991) 342–349.
[14] Z. Li, P.C. Yew and C.Q. Zhu, An efficient data dependence analysis for parallelizing compilers, *IEEE Trans. Parallel Distributed Systems* 1 (1) (1990) 26–34.
[15] D.E. Maydan, J.L. Hennessy and M.S. Lam, Efficient and exact data dependence analysis, in: *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, Toronto, Canada (1991) 1–14.
[16] C.D. Polychronopoulos and D.J. Kuck, Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers, *IEEE Trans. Comput.* 36 (12) (1987) 1425–1439.
[17] C.D. Polychronopoulos, *Parallel Programming and Compilers* (Kluwer Academic Publishers, 1988).
[18] W. Pugh, A practical algorithm for exact array dependence analysis, *Comm. ACM* 35 (8) (1992) 102–114.
[19] W.C. Shih, C.T. Yang and S.S. Tseng, Knowledge-based data dependence testing on loops, in: *Proc. 1994 Internat. Computer Symp.*, Hsinchu, Taiwan, ROC (1994) 961–966.
[20] T.H. Tzen and L.M. Ni, Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers, *IEEE Trans. Parallel Distributed Systems* 4 (1) (1993) 87–98.
[21] M. Wolfe, *Optimizing Supercompilers for Supercomputers* (Pitman, London and MIT Press, Cambridge, MA, 1989).
[22] M. Wolfe and C.W. Tseng, The power test for data dependence, *IEEE Trans. Parallel Distributed Systems* 3 (5) (1992) 591–601.
[23] C.T. Yang, S.S. Tseng and C.S. Chen, The anatomy of parafrase-2, in: *Proc. Nat. Sci. Council Republic of China (Part A)* 18 (5) (1994) 450–462.
[24] C.T. Yang, S.S. Tseng and M.C. Hsiao, A model of parallelizing compiler on multithreaded operating systems, in: *Proc. HPC-ASIA '95*, Taipei, Taiwan, ROC, 1995.
[25] H.P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers* (Addison-Wesley, Reading, MA, 1990).