This paper is revised and resubmitted to *Concurrency: Practice and Experience* for publication.

Using Knowledge-Based Systems for Research on Parallelizing Compilers

Chao-Tung Yang, Shian-Shyong Tseng
Yun-Woei Fann, Ting-Ku Tsai
Ming-Huei Hsieh, Cheng-Tien Wu

Ground System Section
National Space Program Office
8F, No. 9 Prosperity 1st Road
Science-Based Industrial Park
Hsinchu, Taiwan 300, ROC
email: ctyang@nspo.gov.tw
Tel: +886-3-5784208 ext.1563
Fax: +886-3-5779058
Nov. 3, 2000

Using Knowledge-Based Systems for Research on Parallelizing Compilers*

Chao-Tung Yang†
Ground System Section
National Space Program Office
Hsinchu, Taiwan 300, ROC
ctyang@nspo.gov.tw

Shian-Shyong Tseng‡
Yun-Woei Fann, Ting-Ku Tsai
Ming-Huei Hsieh, and Cheng-Tien Wu
Dept. Computer & Information Science
National Chiao Tung University
Hsinchu, Taiwan 300, ROC
sstseng@cis.nctu.edu.tw

November 3, 2000

Abstract

The main function of parallelizing compilers is to analyze sequential programs, in particular, the loop structure, to detect hidden parallelism and automatically restructure sequential programs into parallel subtasks that are executed on multiprocessor. This paper describes the design and implementation of an efficient parallelizing compiler to parallelize loops and achieve high speedup rates on multiprocessor systems. It is well known that the execution efficiency of a loop can be enhanced if the loop is executed in parallel or partially parallel, such as in a DOALL or DOACROSS loop. This paper also reviews a practical parallel loop detector (PPD) that is implemented in our PFPC on finding the parallelism in loops. The PPD can extract the potential DOALL and DOACROSS loops in a program by verifying array subscripts. In addition, a new model by using knowledge-based approach is proposed to exploit more loop parallelisms in this paper. The knowledge-based approach integrates existing loop transformations and loop scheduling algorithms to make good use of their ability to extract loop parallelisms. Two rule-based systems, called the KPLT and IPLS, are then developed using repertory grid analysis and attribute ordering tables respectively, to construct the knowledge bases. These systems can choose an appropriate transform and loop schedule, and then apply the resulting methods to perform loop parallelization and obtain a high speedup rate. For example, the IPLS system can choose an appropriate loop schedule for running on multiprocessor systems. Finally, a runtime technique based on inspector/executor scheme is proposed in this paper for finding available parallelism on loops. Our inspector can determine the wavefronts of a loop with any complex indirected array-indexing pattern by building a DEF-USE table. The inspector is fully parallel without any synchronization. Experimental results show that the new method can resolve any complex data dependence patterns where no previous research can. One of the ultimate goals is to construct a high-performance and portable FORTRAN parallelizing compiler on shared-memory multiprocessors. We believe that our research may provide more insight into the development of a high-performance parallelizing compiler.

Keywords: Parallelizing Compiler; Knowledge-based System; Loop Parallelization; Multithreaded OS; Program Restructuring;

*This work was supported in part by National Science Council of Republic of China under Grants No. NSC86-2213-E009-081 and NSC87-2213-E009-023.

†Corresponding author. Associate Researcher, National Space Program Office, Hsinchu, Taiwan 300, ROC. Phone: +886-3-5784208 ext. 1563, Fax: +886-3-5779058. E-mail: ctyang@nspo.gov.tw.

‡Professor at Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan 300, ROC. Phone: +886-3-5715900, Fax: +886-3-5721490. E-mail: sstseng@cis.nctu.edu.tw.

1 Introduction

The last decade has seen the coming of age of parallel computing. Many different classes of multiprocessor systems have been designed and implemented in industry and academia, for example, IBM RP3, Cray T3D, NEC SX-3, CONVEX C4, CONVEX SPP, and IBM SP2. To achieve high speedup of such systems, it requires decomposition of tasks into several sub-tasks that can be executed on different processors in parallel. Unfortunately, it possesses several difficulties for the users to write explicitly parallel programs. First, they had to rewrite their existing sequential programs into parallel programs. Second, most of the resulting explicitly parallel programs were not portable. Third, writing efficient parallel programs often required optimizations that need intimate knowledge of the machine's architecture and the program's access patterns, e.g., data distribution, prefecting, or blocking.

To address these difficulties, parallelizing compilers were developed to transform sequential programs into parallel ones [16, 1, 9]. Parallelizing compilers can be broken into two components: a component that identifies parallelism in a program, and a component that exploits this parallelism. The component that identifies parallelism attempts to determine what parts of a program can be run in parallel. The component that exploits parallelism determines which of these parallel parts should be run in parallel, as well as how to generate efficient codes for them. Therefore, design of efficient parallelizing compiler is an important part of achieving maximum parallelism on multiprocessors. However, the generation processes of parallel object codes by parallelizing compilers are very difficult and complicated. Most investigations of parallelizing compiler still focus on source-to-source transformation, for example, *Parafrase-2 and Polaris* developed at UIUC [1, 10], *ParaScope* developed at Rice University [3], and *SUIF* developed at Stanford University [8].

In addition to the advance in computer architecture, some operating systems also support parallelism. Multithreading support seems to be the most obvious approach for helping programmers to take the advantage of parallelism by operating system. For example, Mach, OSF/1, Solaris, Microsoft Windows NT are operating systems that support multithreading. These operating systems usually have packages for handling multithreads [2], e.g., the C Threads package in Mach and P Threads package in OSF/1. Although a multithreading operating system for a multiprocessor system can be powerful, it still needs good parallelizing compilers to help programmers exploit parallelism and gain performance benefit. So, we wanted to design and implement a portable parallelizing compiler for multithreading operating system. Our compiler can generate parallel object codes for running on multiprocessor systems rather than being just a source-to-source restructurer [12, 14].

This paper describes the design and implementation of an efficient parallelizing compiler to parallelize loops and achieve high acceleration rates on multiprocessor systems. In this paper we introduce how to design and implement a portable FORTRAN parallelizing compiler (PFPC) on a shared-memory multiprocessor machine running multithreading operating system OSF/1. Our compiler is highly modularized so that porting to other platforms will be very easy. Furthermore, the compiler can partition parallel loops into multithreaded codes based on several DOALL loop-partitioning algorithms.

Then, this paper reports on the practical parallelism detector (PPD) that is implemented in PFPC at

NCTU to concentrate on finding available the parallelism on loops [15]. The PPD is used on extracting the potential DOALL and DOACROSS loops in a program. Moreover, if DOACROSS loops are available, an optimization of synchronization statements was made.

In a shared-memory multiprocessor system, scheduling decisions 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 imbalance when the loop style is not uniformly distributed or the loop bounds cannot be known at compile time. In contrast, dynamic scheduling is more suitable for load balancing; however, runtime overhead must be taken into consideration. Traditionally, the parallelizing compiler dispatches the loop by using only one scheduling algorithm, either static or dynamic.

To exploit more parallelism, a new model by using knowledge-based techniques is proposed in this paper [11]. The knowledge-based approach integrates existing loop transformations and loop scheduling algorithms to make good use of their ability to extract loop parallelisms. Two rule-based systems, called the KPLT (Knowledge-based Parallel Loop Transformation) and IPLS (Intelligent Parallel Loop Scheduling), are then developed using repertory grid analysis and attribute ordering tables respectively, to construct the knowledge bases. For instance, IPLS can choose an appropriate algorithm and then apply the resulting algorithm to assigning parallel loops on multiprocessor systems to achieve high speedup rates [4].

Finally, a runtime technique based on inspector/executor scheme is proposed in this paper for finding available parallelism on loops. Our inspector can determine the wavefronts of a loop with any complex indirected array indexing pattern by building a DEF-USE table [13]. The inspector is fully parallel without any synchronization. Experimental results show that the speedup delivered by our compiler is high. Furthermore, for system maintenance and extensibility, our approach is obviously superior to others. As an ultimate goal, a high-performance and portable FORTRAN parallelizing compiler on shared-memory multiprocessors will be constructed.

2 The Model of Portable FORTRAN Parallelizing Compiler

2.1 An Overview

Multithreading support may be the most obvious approach to help programmers take the advantage of parallelism by operating systems. Therefore, we propose a new model of parallelizing compiler for exploiting potential power of multiprocessors and gaining performance benefit on multithreaded operating systems OSF/1 [2]. The portable FORTRAN parallelizing compiler (PFPC) intended to produce parallel object codes rather than just acting as a source-to-source restructurer is shown in Figure 1 [12, 14].

First, a practical parallelism detector (PPD) is used to test the data dependences of array references and then restructure a sequential FORTRAN source program into a parallel form at compile-time [15], i.e., if a loop can be parallelized or partially parallelized, then PPD marks that loop with DOALL loop or DOACROSS loop by comments. If the access patterns of some arrays cannot be determined at compile-time or have non-constant dependence vector, then PPD marks that loops with DOCONSIDER loop by comments. The flow of loops parallelization is shown in Figure 2. The PPD (practical parallelism detector) will analyze the loop's array access patterns to find the data dependences of array references.

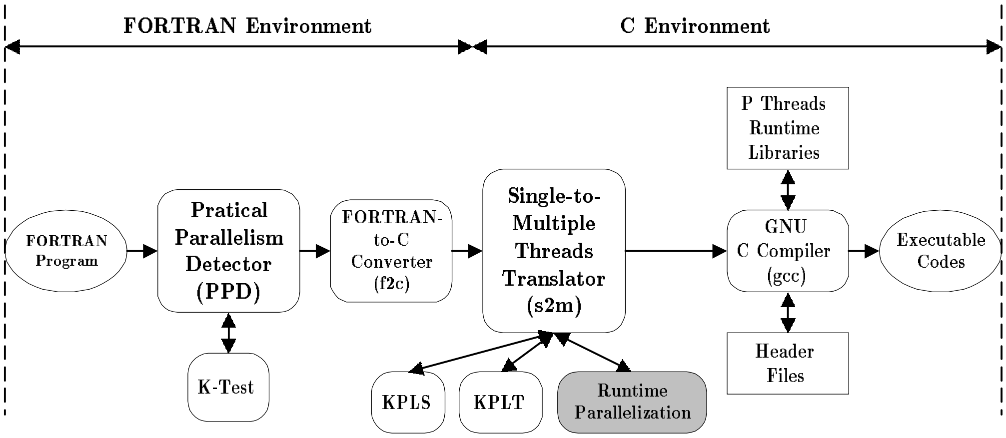


Figure 1: The **PFPC** model running on OSF/1.

As we know, if the information of data dependence is not available until the program is running, i.e., defy the static analysis, then PPD will mark it as a DOCONSIDER loop. If there is no dependence between statements in a loop, or these dependences are loop-independent dependences, different iterations can be executed in parallel on separate processors as DOALL loops. If dependence is occurring across different iterations, i.e., is a loop-carried dependence, it is called a DOACROSS loop. The iterations are executed either sequentially, or partially in parallel by means of enforced synchronization instructions within the bodies of the concurrent loops, and incur some run-time overhead will be incurred. Otherwise, if the loop dependency patterns are too complex to analyze by current algorithms, for example, with non-linear array index expressions or with non-constant dependence distance, then we also can mark it as a DOCONSIDER loop.

Second, because OSF/1 has no FORTRAN compiler and because multithreading only supports C programming, a FORTRAN-to-C (f2c) converter is used to convert the FORTRAN program output by PPD into its C equivalent. Third, the single-to-multiple threads translator (s2m) takes the program obtained from f2c as input, and then generates the output in which the parallel loops (DOALL or DOACROSS) are translated into sub-tasks by replacing them with multithreaded codes. For run-time parallelization, the s2m will generate the inspector and executor codes for DOCONDISER loops at compile-time.

Finally, The resulting multithreaded program is then compiled and linked with the P Threads or C Threads run-time libraries by using the native C compiler, e.g., GNU C compiler. Then, the generated parallel object codes can be scheduled and executed in parallel on the multiprocessors to achieve high performance. Based upon this model, we implemented a FORTRAN parallelizing compiler to help programmers take advantage of multithreaded parallelism on AcerAltos 10000 multiprocessor system, running OSF/1.

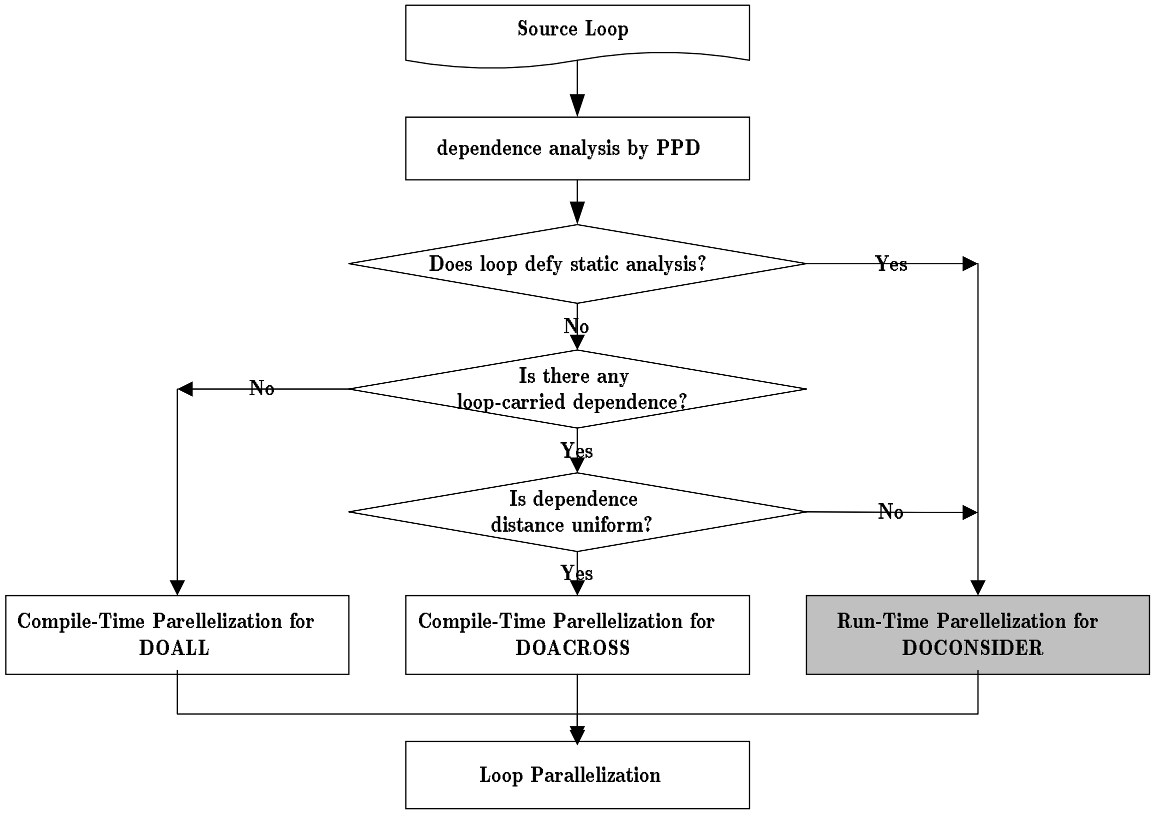


Figure 2: The flow of loop parallelization.

2.2 PPD: A Practical Parallel Loop Detector

PPD (in Figure 3) takes the traditional FORTRAN 77 source program as input and yields the corresponding prompted parallel code. The framework of PPD is divided into two phases, *analysis phase* and *codegen phase*. In analysis phase, a single-subscript testing algorithm, the I test, is used for checking if the linear equation formed by array subscript has an appropriate integer solution. Instead of linearizing the subscript of an array, we check it subscript-by-subscript since there is no certainty that either of them overrides the other in precision. The effect of analysis phase is the determination of the execution modes of all loops. The execution mode of a loop may be the one of the following three types: DOALL, DOACROSS, and DOSEQ, where the former two ones point out that a loop can be executed in a fully or partial parallel manner respectively, and the last one is the normal sequential style. In codegen phase, the outcome of analysis phase is referred to produce the prompted parallel codes. The optimizations for synchronized statements of DOACROSS loops are also taken.

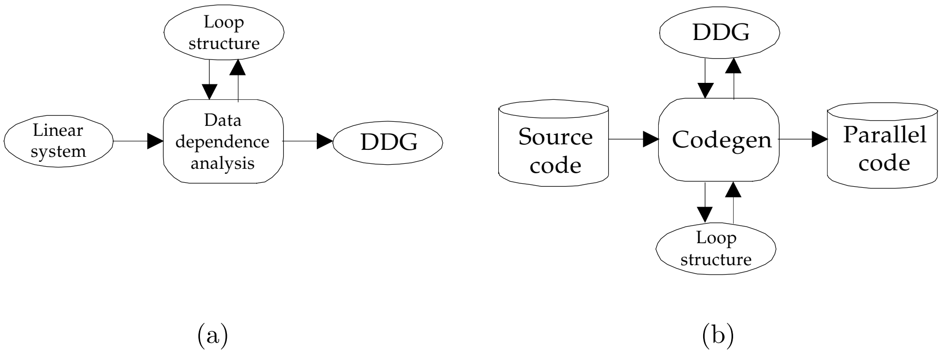


Figure 3: An overview of the analysis and codegen phases.

2.3 S2m: A Single-to-Multiple Threads Translator

The component, single-to-multiple threads translator (s2m), takes the program obtained from f2c as input, and then generates the output in which the parallel loops (DOALL or DOACROSS) are translated into subtasks by replacing them with multithreaded codes. The structure of single-to-multiple threads translator (s2m) [12] consists of five modules as shown in Figure 4. The kernel module is written so as to be portable; it calls functions in the thread-code generating module and calls functions in the DOALL loop-partition module. The thread-code generating module contains several functions that are used to generate different thread specific codes: P Threads or C Threads. The DOALL loop-partition and DOACROSS loop-partition modules contain routines that partition DOALL and DOACROSS loops, respectively. In this paper, we improve the power of s2m to partition and generate corresponding multithreaded codes for a DOACROSS loop. The config module is very small and contains several arrays of functions. When the s2m kernel calls a function in thread-code generating, DOALL loop-partition modules or DOACROSS loop-partition modules, there must be an entry in the config module pointing to that called function so that the s2m kernel can access the function through the config module. If users want to add their own thread-code generating routines, DOALL loop-partition routines, or DOACROSS loop-partition routines, they can append their own functions to these three modules and then append entries pointing to those functions in the config module. Therefore, a new version of s2m can be produced by simply compiling the config module and user functions directly, which can be ported to other platforms easily.

We will now explain how the s2m converts specific types of conventional sequential programs, i.e., DOALL loops, into their parallel equivalents with the P Thread runtime library codes embedded in them. The general form of a DOALL loop program for s2m is shown in Figure 5. In this figure, there is one for-loop enclosed in "`/* /$DOALL$/ L???:  */`" and "`/* /$END_DOALL$/ L???:  */`" comments, these two comments are used to indicate the for-loop enclosed by them is a DOALL loop. The `???` here stands for the loop label used in the original FORTRAN program.

The output of the main program has the form shown in Figure 6 produced by s2m. There are six rectangles in this figure; each corresponds to a session that performs a specific job. The first session, thread-related definition, outputs thread-related definitions. Some variables for using the thread package

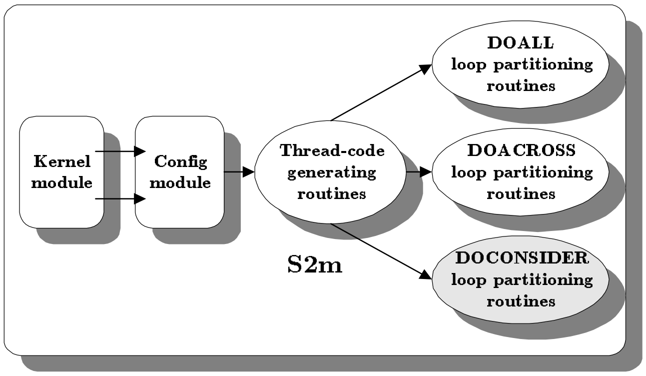


Figure 4: The structure of **s2m**.

```
main()
{
   Variables declaration area
   ...
/* /$DOALL$/???:  */
   for (i= ....  ){
      ...
/* L???:  */
   }
/* /$END_DOALL$/ ???:  */
   ...
}
```

Figure 5: The DOALL loop of input program to s2m.

are defined in this session. The `loop` variable is an array of `loop_args`, which is used to pass the begin iteration, end iteration, and the iteration step for each pthread created later on. The `ThCount` variable records the number of threads; this number is decreased by one when a thread is going to be terminated.

The second session is devoted to variables declaration. This session, is originally in the main function of the input program (see Figure 5), s2m removes the variable declaration from the main function to make them visible to the entire program. This eases the parameter passing problem when a thread forks since all the necessary variables are global! Note that when this approach is applied to functions other than the main function, the variables may need to be renamed to avoid conflicts.

The third session is mutex and condition variables initialization. This session initializes a mutex object and a condition variable with default attributes since we need these two variables when performing synchronization. The P Threads codes for this session are shown below.

```
pthread_mutex_init(&CountLock, pthread_mutexattr_default);
pthread_cond_init(&ThCond, pthread_condattr_default);
```

```
Thread related definition
Variables declaration area
main()
{
    Mutex & condition variable initialization
    ...
    Iterations calculation
    Fork threads
    Synchronization
    ...
}
```

Figure 6: Main program of the general output produced by s2m.

The fourth, fifth, and sixth sessions are for DOALL loops only. The iterations calculation session, also called the loop partitioning session, partitions the DOALL loop according to the user-assigned loop-partitioning algorithms. The default loop partitioning algorithm is CSS/4 which divides the iterations into four chunks of equal size, but this can be changed with a command line option when s2m is invoked. At the end of this session, the variable `ThCount` will have the number of threads that need to be created later on. Start iteration, end iteration and the iteration step for the i_{th} thread is stored in `loop[i].begin`, `loop[i].end` and `loop[i].step`, respectively. This pre-calculation of tasks for each thread eliminates the need for synchronization of loop indices in several loop scheduling algorithms. This makes our approach faster. The fifth session is just to fork threads. The default number of threads to be created is four; however, this number can be changed with a command line option when s2m is invoked. Usually, this session contains a for-loop that performs threads creation. The sixth session uses previously initialized mutex objects and condition variables for synchronization purposes. It waits for the thread count (`ThCount` variable) to reach zero (the threads created for a particular parallel loop have all been finished) and then continues execution. The thread count is initially the number of threads created, and is decreased by one before each thread is terminated. This synchronization is necessary for ensuring the correctness of program execution. The P Threads codes for this session may look like this:

```
pthread_mutex_lock(&CountLock);
while(ThCount != 0)
pthread_cond_wait(&ThCond, &CountLock);
pthread_mutex_unlock(&CountLock);
```

Figure 7 is the function definition of a DOALL loop for a thread. This function definition is mainly the corresponding for-loop with minor change. First, the loop is executed from `loop->begin` to `loop->end`; these two variables are calculated in the iteration calculation session. Second, the thread count (`ThCount` variable) is decreased by one before the thread is terminated. We use mutex objects to ensure mutex exclusion, and then decrease the thread count by one. The P Threads codes for this session are shown below.

```
pthread_mutex_lock(&CountLock);
ThCount--;
pthread_mutex_unlock(&CountLock);
pthread_cond_signal(&ThCond);
```

```
void DOALL??(loop)
struct loop_args *loop;
{
int i;
   for (i=loop->begin; i<=loop->end; i++){
      ...
   }
   Decrease the thread count by 1
}
```

Figure 7: The DOALL function definition for an output thread produced by s2m.

3 Main Results

3.1 Using Knowledge-based Techniques on Loop Parallelization

It is well known that knowledge system is a system that depends 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 now knowledge or update existing knowledge easily. The rule-based approach is one of the commonly used forms 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). RGA is easy to use, but it suffers from the problem of missing embedded meanings. For example, when a doctor expresses the features of catching a cold are headache, cough and sneeze, he may have 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. 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 this paper, the development environment is not discussed here. 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, scheduling, or transformation 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. In our implementation, 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 system, the latter method is used.

A translator, GRD2CLP, translates the repertory grid and attribute ordering table to CLIPS's production rules. This approach has great flexibility as we can add new scheduling algorithms to the repertory grid and attribute ordering table, and then use GRD2CLP to convert the tables into CLIPS rules. The process of generating a knowledge base is shown in Figure 9.

- **Inference Engine:** The inference engine is the interpreter of the knowledge stored in the knowledge base. It examines the contents of the knowledge base and the data including the system characteristics and the loop attributes provided by machine architecture and programmers to derive a conclusion, an appropriate parallel loop-scheduling algorithm. The inference engine attempts to find connections between the input attributes stated in section three and the selected loop-scheduling algorithm according to RGA and AOT. An example of applying RGA/AOT is shown in Table 1. 'X' means that the attribute has no relation with the scheduling algorithm. 'D' means that the attribute dominates the scheduling algorithm, i.e., if the attribute is not equal to the entry value, it is impossible for the scheduling algorithm to be implied. For those entries that are not labeled 'X' or 'D', integer numbers are used represented the relative degree of importance for attribute does not dominate the object but is of some degree of importance relative to other attributes. Larger integer number implies the attribute being more important to the object. According to the table, four rules can be generated. As we observe, [A1, S1]=1,5,6, [A2, S1]=YES, [A3, S1]=X; hence the resulting rule will be generated.
  RULE:
  If (A1 is in 1,5,6) and (A2=YES) Then Choose S1

Table 1: The repertory grid and the attribute ordering table.

	S1	S2	S3	S4
A1	1,5,6/D	X/X	3/D	2,4/D
A2	YES/D	X/X	YES/D	X/X
A3	X/X	NO/2	NO/D	X/X

- **Algorithm Library:** The library collects several representative transformations and schedules, either proposed by others or designed by ourselves. The question of how these transformations and schedules are chosen in the *development environment*, so here we assume that it has been built. For example, we have included eight scheduling algorithms in the library for loop scheduling, that are static scheduling, SS, CSS, GSS, Factoring, TSS, AFS (MAFS, DAFS), and LDS. This is another advantage of using knowledge-based system; we can easily modify the rules and add any new scheduling strategy.

3.2 IPLS: An Intelligent Parallel Loop Scheduling

In PFPC, we propose a system as shown in Figure 10, called intelligent parallel loop scheduling (IPLS), by using knowledge-based techniques to select an appropriate loop-scheduling algorithm. The approach will make good use of the advantages of the algorithms for loop parallelism. By the resulting algorithms for

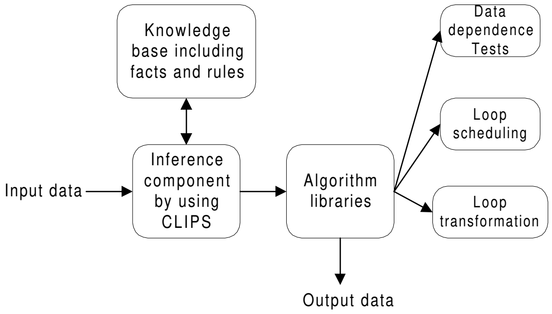


Figure 8: Components of our new model

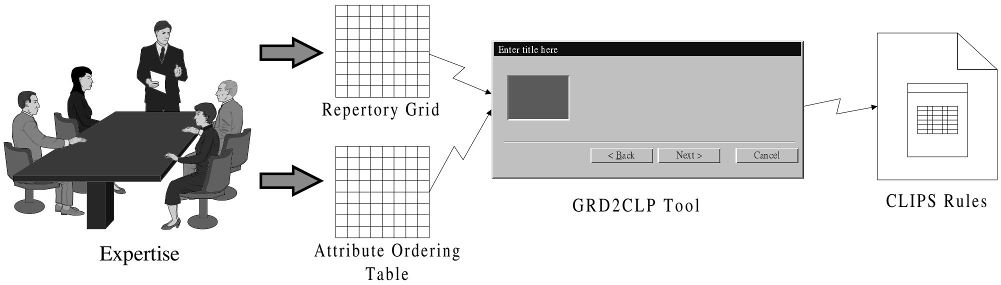


Figure 9: The process of generating a knowledge base

assigning parallel loop on multiprocessor systems, it is believed that the applications can save execution time and achieve high speedup. The runtime environment of IPLS contains two more components, which are briefly described as follows:

- Profile Information - After the program applying the selected loop scheduling algorithm is executed, some information about number of iterations, maximal time of iteration, minimal time of iteration, total time of program, number of synchronization, number of remote memory accesses, and the workload distribution of each processor will be recorded and saved in a profile file. The profile file will be referred to modify the attributes by refining system.

- Refining System - When a program is embedded with some parallel loop scheduling algorithm, if we can refine some attributes, such as the values of factors in the loop-scheduling algorithm by using the profile information derived from the record of executing process of the program. Refining procedure in order to get ideal values will modify the factors. It is obvious that this make the parallelism of program higher and performance better.

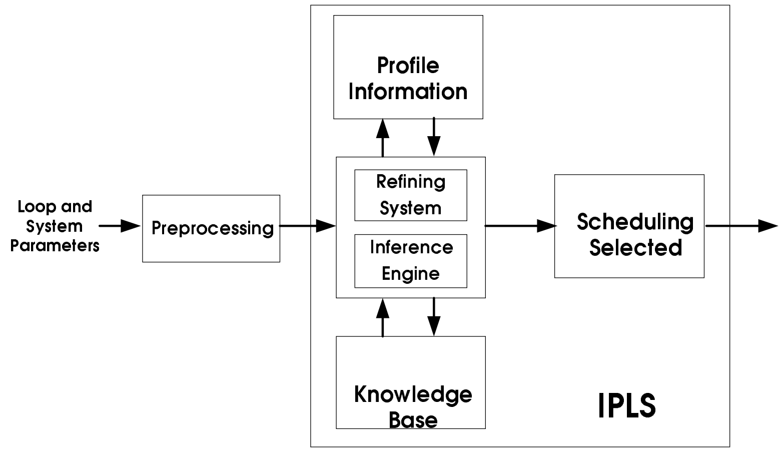


Figure 10: The system architecture of IPLS.

Tables 2 and 3 show the relationships between seventeen attributes and parallel loop scheduling algorithms in UMA and NUMA models respectively. Besides, each table is districted by different kind of loops, i.e., DOALL and DOACROSS loops. The features mentioned above are the attributes upon which we constructed our attribute grid. 'Machine model' is classified into UMA and NUMA. 'Memory access ratio' means the speed ratio of cache, memory and network. 'CPU number' denotes the system size, which can be classified into three levels, small, medium, or large. 'Loop style' includes four kinds of loop, such as U(uniform), I(increasing), D(decreasing) and R(random). 'Program size' shows the appropriate scale that algorithms fit. 'Data locality' determines if loop data behavior has affinity or not. 'Loop boundary' determines if it must be known at compile time. 'Loop level' determines if nested loop is profitable to algorithms. 'Loop carried dependence' is classified into DOALL and DOACROSS. 'Easiness' describes if the implementation of algorithm is easy. 'Factor' means the variables, which can dynamically influence the performance due to loop information and system states. The overheads of synchronization, communication and thread management are roughly classified into four levels, none, light, normal, or heavy. 'Start time' determines whether all each processor starting time need to be equal or not.

In many parallel loop-scheduling algorithms, there are some attributes, such as factors, which influence the performance of executing program. For example, the adaptive hybrid scheduling algorithm has two factor, β and γ, determining the fetching processor whether or not to fetch more iterations form work queue in dynamic level after executing the iterations coming from static level. These two factors, β and γ, should be adjusted by the programmers according to the properties of parallel computers. However, how to select appropriately the value of β and γ on different system is difficult. If we can refine values of the factors in the loop-scheduling algorithm by using the profile information derived from the record of executing process of the program, it is obvious that the new factors cyclically modified by refining procedure will make the parallelism of program more clear and make the performance better. And we say this method stated above has feedback-learning ability and is intelligent. In the paper, a refining system based upon the profile

Table 2: The attributive table for UMA models.

UMA Model									
	DOALL								DOACROSS
	Static	SS	CSS	GSS	TSS	Factoring	AHS	SSS	Enhanced CSS
UMA/NUMA	UMA	UMA	UMA	UMA	UMA	UMA	UMA/NUMA	UMA/NUMA	UMA
No of Processor	X	X	X	X	X	X	X	X	X
Memory Access Rate	1:10:200	1:10:200	1:10:100	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200
Loop Style	U, D, I	X	U, R	U, I, R	X	X	X	X	U
Program Size	X	X	Large	X	X	Large	X	X	Large
Loop Type	1-10	X	1-2, 5-7, 9-10	2-3, 7-8	1, 3-4, 7, 11	3-4, 7-8	X	1-3, 5-11	1, 6-7
Data Locality	X	No	No	No	No	No	Yes	Yes	X
Loop Boundary	Yes	X	No	X	X	X	Yes	Yes	X
LCD	DOALL	DOALL	DOALL/(0,1)	DOALL	DOALL	DOALL	DOALL	DOALL	Doacross (>1)
Easiness	X	X	X	No	X	No	No	No	No
Factor	-	-	k	-	N_S, N_F	$x = 2$	β, γ	α, k	K
Thread Overhead	l, n	h	l, n	h	n	n	n, h	n, h	l, n
Comm. Overhead	X	l	l	l	l, n	l	l, n	1, n	l
Sync. Overhead	X	1	2, 3, 4	2	3, 4	3, 4	4, 5	4, 5	3, 4, 5
Start Time	Yes	X	Yes	X	X	X	Yes	X	Yes

Table 3: The attributive table for NUMA models.

NUMA Model								
	DOALL							
	AFS	MAFS	CAFS	LAFS	DAFS	LDS	GDCS	ASS
UMA/NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
No of Processor	S, M	S, M	S, M, L	S, M, L	S	S, M	S, M	S
Memory Access Rate	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200	1:10:200
Loop Style	X	D, I, R	X	X	X	X	X	X
Program Size	-	-	-	X	-	-	-	-
Loop Type	2-3, 9-10	2, 4-5	1, 4, 8	1, 4, 8	2 , 5, 9, 11	1, 5, 6, 8, 10	2, 5, 9, 11	2, 4, 5, 6
Data Locality	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Loop Boundary	X	X	X	X	X	X	Yes	X
Loop Level	X	X	X	X	X	X	X	X
LCD	DOALL	DOALL	DOALL	DOALL	DOALL	DOALL	DOALL	DOALL
Easiness	No	No	No	No	No	No	No	No
Factor	$0.5 < \alpha < 1$	-	K	k	-	B	α, β	$\alpha = \frac{N}{P^2}$
Thread Overhead	l, n	l	l, n	l, n	l	l, n	l, n	l
Comm. Overhead	l	l	l	l	l	l, n, h	l, n, h	h
Sync. Overhead	3, 4	5	3, 4	3, 4	5	3, 4	4, 5	3, 4
Start Time	X	X	X	X	X	X	X	X

information consisting of the following seven items will be included into our model.

- The number of iteration
- Maximal time of iteration
- Minimal time of iteration
- Total time of program
- The number of synchronization
- The number of remote memory access
- The workload distribution of each processor

How to refine attributes and not to modify rules in the knowledge base is a problem, which is solved in our refining system by storing attribute data into a file called Attri_file and using data type of structure (record) as condition testing of antecedent of if statement in rules. When a loop is executed and profile information is generated, the refining system will input profile information to modify the attributes in

Attri_file; therefore, the rules in knowledge base does not need to be changed and the inference engine does not need to be recompiled.

There are several situations at which the refining system is suggested. Firstly, when IPLS is constructed completely, maybe the attributes in knowledge base are crude that an optimal loop-scheduling algorithm to transform a sequential program into an efficient multithread program cannot be selected. Secondly, when IPLS is ported on a new environment, some attributes about system states, such as memory access rate, need to be changed to influence the selection of scheduling method. In addition, perhaps an appropriate loop-scheduling algorithm is selected by inference engine, but the bad values of factors in algorithm, such as chunk size in CSS, will result in larger execution time. The factors had better be refined to reduce the wasted considerable execution time if the executable code will be executed repeatedly. It seems that the overhead from refining the attributes can be neglected because of its advantage. After all, to increase the accuracy is to increase efficiency. The flow chart of refining system is shown in Figure 11. The programmer can determine whether to use refining system before deriving an ideal loop-scheduling algorithm for the program or not. When using the refining system, the programmer can also decide the number of loop-scheduling algorithms selected by inference engine.

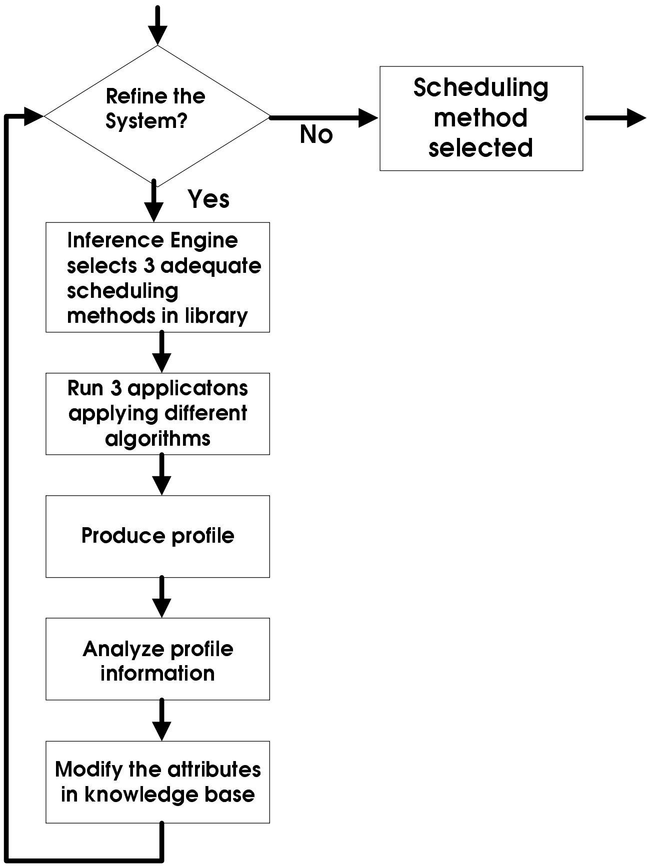


Figure 11: Flow chart of refining system

3.3 Run-Time Parallelization

The method of parallelizing our general inspector is to partition the entire range of iterations into consecutive segments, after which each segment is assigned to a different thread. Each thread produces a valid parallel schedule for iterations in its segment and ignores any dependence with other iterations outside of its segment. After all segments have finished, we have a schedule for each segment. Every such schedule, called a *sub-schedule*, is a mapping from the iterations in the corresponding segment to the wavefronts of that segment. The overall schedule is formed by concatenating the sub-schedules according to the order of the segments in entire range of iterations. The number of segments can be set appropriately, and intuitively, we will set the number of processors to this number. However, if the number of segments is larger, then it will increase the total number of wavefronts (i.e., *depth*) in overall schedule. A larger number of wavefronts implies that there are fewer iterations in each wavefront, and then it will decrease the speedup of run-time parallelization.

The executor performs the overall schedule extracted by the general parallel inspector. As a rule of thumb, the executor performs the sub-schedule of each segment in order, i.e., visits the first wavefront from first to last in a segment, then does for next segment's first wavefront closely, and continues until the last wavefront of the last segment have been visited. Every wavefront is sequentially executed, and, ideally, all iterations in the same wavefront are executed concurrently. In practice, iterations in the same wavefront are partitioned into equal-sized chunks and every chunk is enclosed in one thread. The number of threads is automatically adapted according to the number of iterations in each wavefront by calling function `auto-adapted`, and then the threads scheduled by OSF/1 operating system can be executed in parallel manner.

Now, we will explain why the `auto-adapted` function is used in the executor phase. In the inspector phase, the number of iterations in each wavefront will vary according to the distribution of data dependence in a loop. Therefore not all wavefronts will have great quantities of iterations to obtain good parallelism. In practice, if the executor engages large number of threads for a wavefront which has fewer iterations, then it will incur additional run-time overheads or even let the iterations execute in sequential manner (when number of threads > number of iterations in wavefront). Conversely, if the executor engages fewer threads for a wavefront which has more iterations, then we cannot obtain the deserved parallelism.

Table 4: The `auto-adapted` mapping function.

Number of iterations	1	2	∼8	∼16	∼32	∼64	∼128	∼
Number of threads	1	2	3	4	5	6	8	12

For the sake of efficiency, we propose a strategy to solve above the problems, to get a tailored number of threads for each wavefront; threads are dynamically engaged to a wavefront according to the number of iterations in it. The `auto-adapted` function is a mapping function that maps number of iterations in a wavefront to number of threads. For example, the `auto-adapted` function for our system (4 processors) is shown in Table 4. It should be noted that this mapping function would be modified according to the system environment.

We will now compare the methods described in this paper with several other techniques that have been proposed to analyze and schedule DO loops at run-time. Most of this work has concentrated on developing inspectors. A high level comparison of the various methods is given in Table 5. Since the process of the inspector uses to find the wavefronts can be parallelized fully without any synchronization. Our executor can execute the loop iterations concurrently.

Table 5: Comparison of the characteristics of several methods. The superscripts have the following meanings: 1. Our parallel inspector perhaps cannot get an optimal schedule. 2. The `bit-vector` atomic operation must be applied to avoid the use of global synchronization. Since most of parallel machines don't provide this operation, the performance of this run-time method is degraded.

Methods	Get optimal schedule	No sequential portions	No global syn.	No restrict type of loops	No merge between pro.	No large local mem. required	Integrate in compiler
Our Method	No[1]	Yes	Yes	Yes	Yes	Yes	Yes
Zhu and Yew	No	Yes	No	Yes	Yes	Yes	No
Midkiff and Padua	Yes	Yes	No	Yes	Yes	Yes	No
Chen *et al.*	No	Yes	No	Yes	Yes	Yes	Yes
Rauchwerger *et al.*	Yes	Yes	Yes	Yes	No	No	Yes
Saltz *et al.*	Yes	No	No	No	Yes	Yes	Yes
Leung and Zahorjan	Yes	Yes	No	No	Yes	Yes	Yes
Sheng *et al.*	Yes	Yes	Yes[2]	Yes	Yes	Yes	No

4 Experimental Results

4.1 Performance of PPD

To evaluate the performance of PPD for PFPC, experiments were performed using both practical and contrived data. The practical data included two numerical packages, LINPACK and EISPACK, while the contrived data included several examples that appeared in other papers. Another program parallelization restructurer, Parafrase-2, was also applied to the same testing data, and the results compared with those from our design. LINPACK and EISPACK are two well-known numerical packages. LINPACK is a collection of FORTRAN subroutines that analyze and solve various systems of simultaneous linear algebraic equations, while EISPACK is a collection of subroutines for evaluating the eigenvalues of matrices. Because of their systemization and representatives, the packages have been widely adopted as benchmark programs [16]. There is total of 256 DO loops distributed across the 52 subroutines in LINPACK. PPD was able to exploit 51 DOALL loops and 0 DOACROSS loops, as was Parafrase-2. In the experiments using LINPACK, we have examined all the DOALL loops detecting by PPD and Parefrase-2 carefully. PPD was able to exploit the same 51 DOALL loops as Parafrase-2 was. Because there is no DO loop that can be translated into DOACROSS loop by using our algorithm in the experiments of LINPACK. So, we show the other experiments for demonstrating the DOACROSS loops detected by using PPD.

There are a total of 657 DO loops distributed across the 77 subroutines in EISPACK. PPD was able to exploit 185 DOALL loops and 7 DOACROSS loops, while Parafrase-2 was able to exploit only the 185 DOALL loops. If there is a constant dependence distance in the loop, PPD will record the information for generating the synchronization statements, and translate that loop into DOACROSS loop during codegen phase. In our version, Parafrase-2 cannot detect the DOACROSS loops, so PPD was able to exploit 7 DOACROSS loops in the experiments using EISPACK, while Parafrase-2 was not. Comparative results are shown in Table 8. PPD translated the loops into DOALL or DOACROSS loops conservatively. So, it is not possible that PPD mistakenly marks non-DOALL loops as DOALL or non-DOACROSS loops as DOACROSS.

The practical data, as shown in Figure 12(a), is a program segment that computing the transitive closure of an adjacency matrix. Only loop `K` could be transformed into a DOALL loop by PPD, as shown in Figure 12(b), while loops `J` and `K` are both transformed into DOALL loops by Parafrase-2, as shown in Figure 12(c), but, parallelizing loop `J` seems wrong. We expose the mistake as follows: Suppose that the iteration vector of the accesses `A(I, K)` is (J, I, K), and the one of the access `A(J, K)` is (J', I', K'). If loop `J` is a DOALL loop, then we cannot ensure execution order between, say $(J, I, K) = (2, 3, 4)$ and $(J', I', K') = (3, 2, 4)$, with the result that the anti-dependence at memory location `A(3, 4)` can not be preserved.

```
      DO 10 J=1, 1000
        DO 20 I=1, 1000
          IF (A(J, I) .EQ. .TRUE.) THEN
            DO 30 K=1, 1000
              IF (A(I, K) .EQ. .TRUE.) THEN
                A(J, K)= .TRUE.
              END IF
30          CONTINUE
          END IF
20      CONTINUE
10    CONTINUE
```

(a)

```
      DO 10 J=1, 1000
        DO 20 I=1, 1000
          IF (A(J, I) .EQ. .TRUE.) THEN
C           DOALL 30 K=1, 1000
            DO 30 K=1, 1000
              IF (A(I, K) .EQ. .TRUE.) THEN
                A(J, K)= .TRUE.
              END IF
30          CONTINUE
C           ENDALL 30
          END IF
20      CONTINUE
10    CONTINUE
```

(b)

```
      CDOALL 10 J=1, 1000
        DO 20 I=1, 1000
          IF (A(J,I) .EQ. .TRUE.) THEN
            CDOALL 30 K=1, 1000
              IF (A(I,K) .EQ. .TRUE.) THEN
                A(J,K)= .TRUE.
              END IF
30          CONTINUE
          END IF
20      CONTINUE
10    CONTINUE
```

(c)

Figure 12: The transitive closure program.

4.2 Performance of IPLS

To demonstrate the performances of IPLS, there are two experimentations on UMA system and NUMA system, the first one concerns each execution time and speedup of above ten applications, and the other is a combined program, including ten applications. Under the implementation on UMA system, which is 2-processor machine, the execution time and the corresponding speedup are shown in Table 6.

Table 6: The execution time (ms)/speedup of 11 applications applying different scheduling algorithms.

Applications	SERIAL	CSS/2	GSS	TSS	Factoring	SSS	AHS	IPLS
Adj_Con	20104/1	15042/1.337	15055/1.335	10398/1.933	13974/1.439	12359/1.627	12352/1.628	as TSS
Gauss_Eli	365359/1	256945/1.422	197157/1.853	202922/1.8	195016/1.873	208055/1.756	196852/1.856	as Factoring
Gauss_Jor	7765/1	4245/1.829	5587/1.39	5599/1.387	5266/1.475	4333/1.792	4391/1.768	as CSS/2
Jacobi_Iter	14047/1	10109/1.39	12836/1.094	12656/1.11	13125/1.07	9802/1.433	9758/1.44	as AHS
LU	40995/1	28094/1.459	33521/1.223	34356/1.193	33071/1.24	28505/1.438	28432/1.442	as CSS/2
Matrix_Mul	23453/1	12281/1.91	12095/1.939	12229/1.918	12214/1.92	12187/1.924	12203/1.922	as CSS/2
Radj_Con	27235/1	21274/1.28	14719/1.85	15587/1.747	15255/1.785	14336/1.9	15477/1.76	as SSS
SOR	109062/1	76891/1.418	82594/1.32	83943/1.299	86742/1.257	77376/1.41	77680/1.404	as CSS/2
Spath	63063/1	57032/1.106	58867/1.071	43146/1.462	61547/1.025	38126/1.654	38797/1.625	as SSS
Tran_Clos	479188/1	298312/1.606	308844/1.552	325430/1.472	310469/1.543	295922/1.619	296078/1.618	as SSS
If_Then	17125/1	9682/1.769	9693/1.767	8595/1.992	8667/1.976	8656/1.978	8620/1.987	as AHS

GSS performs poorly for Adjoint Convolution because the workload of iterations is decreasing, and TSS is the most efficient algorithm for Adjoint Convolution. CSS/2 is suitable for the applications like Gauss Jordan Elimination with random unbalanced workload, LU Decomposition with decreasing unbalanced workload, and SOR with uniform balanced workload respectively. Factoring scheduling algorithm is suitable for Gauss Elimination with random balanced workload. SSS is suitable for the applications like Reverse Adjoint Convolution with increasing unbalanced workload, All Pairs Shortest Paths with random balanced workload, and Transitive Closure with random unbalanced workload respectively. AHS is suitable for Jacobi Iteration with random unbalanced workload. We can find that none of six scheduling algorithms on UMA system is suitable for all applications. Alternatively, IPLS can choose an appropriate scheduling algorithm and get good performance for most applications except Matrix Multiplication and If_Then application. In the case of Matrix Multiplication, IPLS does not apply the optimal approach, GSS, but chooses CSS/2, because the workload of iterations in this program is uniform. Whereas the number of processor, 2, is so small that CSS cannot exploit the ability fully. In the case of If_Then application, IPLS does not apply the optimal approach, TSS, but chooses AHS, because AHS is suitable for random workload of iterations in the program. The reason for not selecting an optimal approach is like the case of Matrix Multiplication. Although the selection of scheduling algorithms is not absolute accurate, we can solve the problem by refining the attributes causing the error. Refining system in IPLS will be used afterwards. Traditionally, once a scheduling algorithm is used, it will be used through the entire program. But in IPLS, it can always choose an appropriate scheduling algorithm according to the behaviors of the loops among one program. As the second experiment, IPLS chooses different scheduling algorithms for each loop in the combined program integrated from the above eleven applications. For example, according to the loop behaviors, IPLS selects TSS for the Adjoint Convolution part of the combined program and factoring for the Gauss Elimination part, instead of only one scheduling method. Table 7 shows the experimental execution time and the corresponding speedup for the combined program.

Table 7: The execution time (ms)/speedup of the combined program for different scheduling algorithms.

Applications	Serial	CSS/2	GSS	TSS	Factoring	SSS	AHS	IPLS
All	1167396/1	789907/1.477	750968/1.554	754511/1.547	755346/1.545	709657/1.645	700640/1.666	693687/1.683

4.3 Performance of PFPC on Windows NT

Finally, we demonstrate the performance of PFPC, ten examples are used for DOALL loop parallelization of S2m. Figures 13, 14 and 15 show the speedups of adjoint convolution, Gaussian elimination, matrix multiplication, reverse adjoint convolution, transitive closure, SOR, Jacobi iteration, Gaussian-Jordan elimination, LU decomposition, and all pairs shortest paths by using different program sizes loop. In the experiments, CSS is used to partition every example and obtain their corresponding performances. Obviously, speedup of parallel version is always higher than serial version. So, the S2m translator used in Windows NT can perform high speedup on multiprocessor systems. Particularly, for the loop with uniform workload, such as matrix multiplication shown in Figure 13 (c), it can achieve higher speedup, since the CSS is suitable for the uniform workload loop.

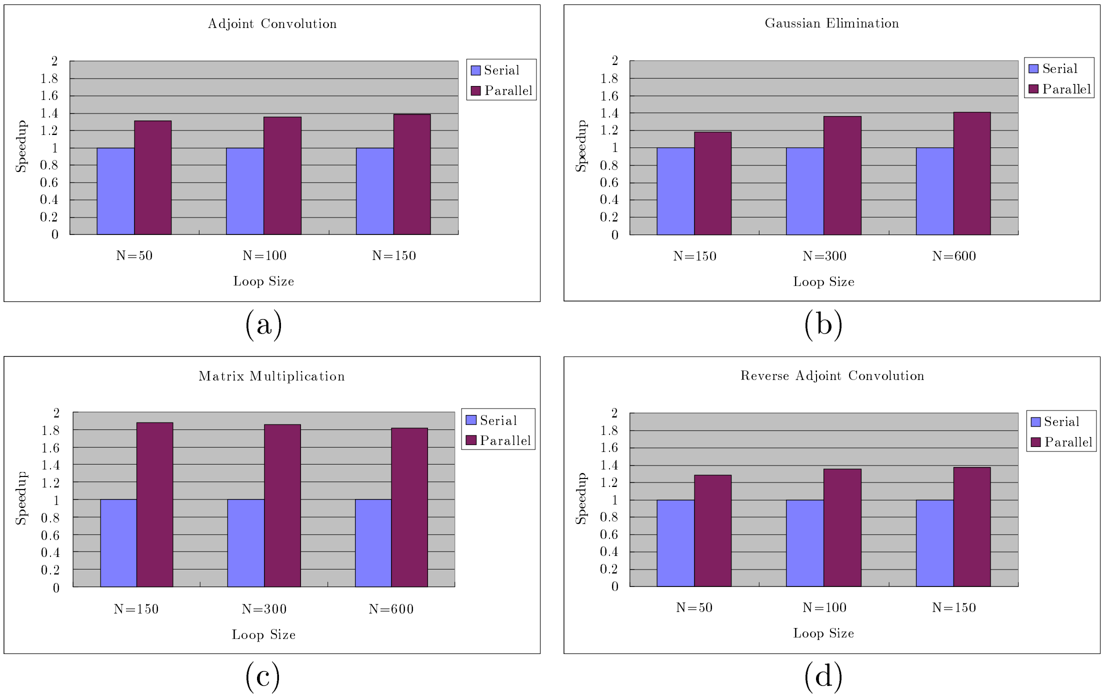


Figure 13: Part I: partial results of DOALL examples.

We examine the characteristics of adjoint convolution and reverse adjoint convolution. In Figure 16 (a), because adjoint convolution is with decreasing workload, we distribute 1/3 of all workload to the first thread and 2/3 of workload to the second thread. As a result of workload is balanced, the speedup is raised. Moreover, in Figure 16 (b), because reverse adjoint convolution is with increasing workload, we distribute 2/3 of all workload to the first thread and 1/3 of workload to the second thread. As a result of workload is balanced, the speedup is raised. Furthermore, for loop of increasing workload, GSS can distribute workload more balanced, so its speedup is raised again.

In order to compare performances of different loop partition algorithms, we examine five representative applications. Figure 17 shows speedup when applications was run with different loop-partitioning algo-

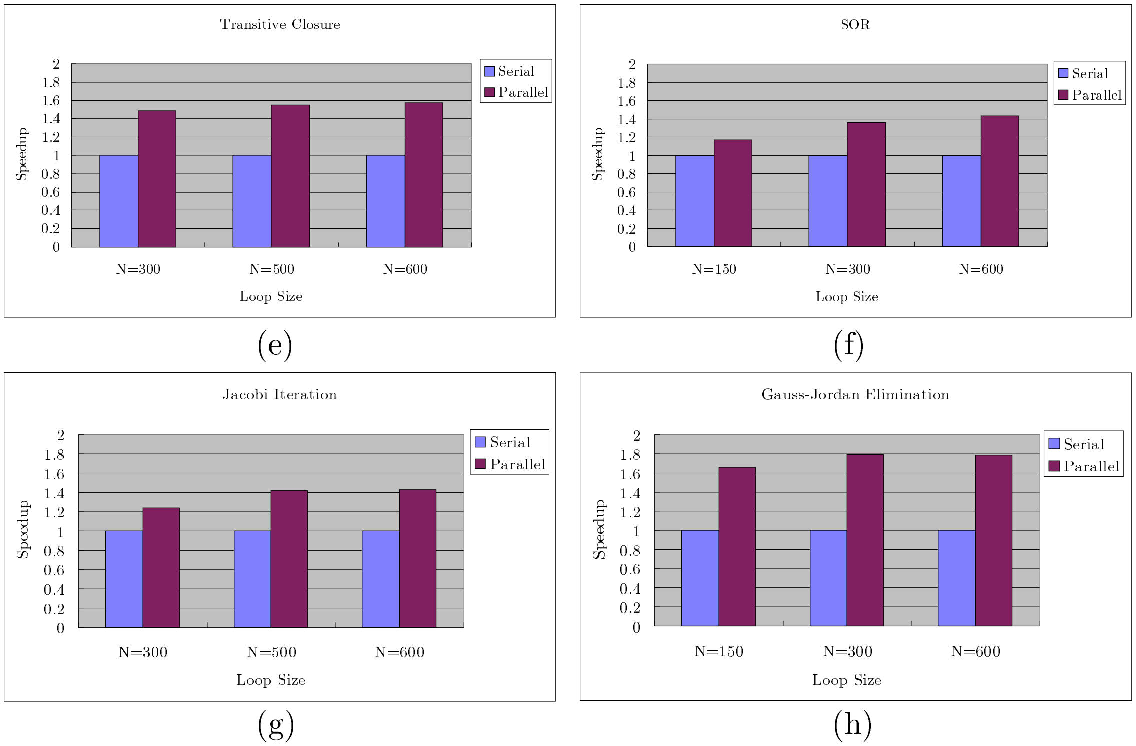


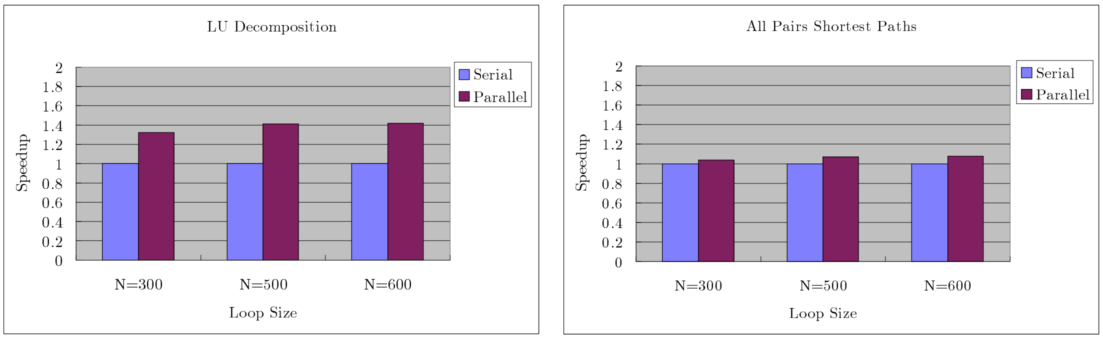
Figure 14: Part II: partial results of DOALL examples.

rithms and arguments. For adjoint convolution and reverse adjoint convolution, Factoring obtains highest performance. For matrix multiplication and transitive closure, CSS/2 obtains highest performance. For Gaussian elimination, TSS obtains highest speedup.

We compared the performance of applications that run in various operating systems, such as OSF/1 and Windows NT. Due to machine architectures are different, such as the number of CPUs. In OSF/1, the machine has four CPUs. In Windows NT, our target machine has two CPUs. Speedup is compared between OSF/1 and Windows NT, and speedup which obtained in OSF/1 must be divided by 2 before comparing each other. We still use five representative applications. The program size of adjoint convolution and reverse adjoint convolution are 150×150. The program size of Gaussian elimination is 750×750. The program size of matrix multiplication is 600×600. The program size of transitive closure is 1000×1000. The speedup of applications which run in the two operating system show in Figure 18.

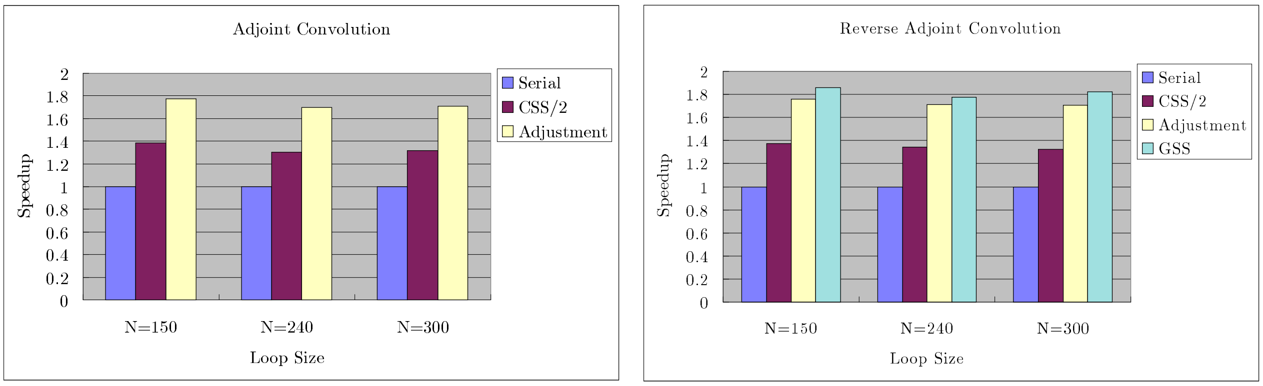
Figure 19 (a) shows the speedup of loop with distance 4. By using doacross scheduling, parallel version can obtain higher performance than serial version. Figure 19 (b) shows the speedup of loop with anti-dependence and distance 2. Due to synchronization and scheduling overhead is large, the speedup is not good as we expected. But parallel version is still better than serial version.

In Figure 20, four examples with different types of loops are given to show the performance obtained by using CSS scheduling method and the experimental environment described in the previous. In Figure 20 (a), every iteration of the outer loop has a constant workload. Figure 20 (b) is with an increase workload and Figure 20 (c) is with a random workload. Both of them are originally obtained from the Perfect



(i) (j)

Figure 15: Part III: partial results of DOALL examples.



(a) (b)

Figure 16: Results of adjusted adjoint and reverse adjoint convolution.

Club identified by other researchers as being parallel but not made parallel by current compilers [1, 7]. In Figure 20 (d) is a synthetic loop with non-constant dependence distance and also has an increase workload.

In runtime, parallel version is obviously better than serial version. Although the speedup of a program with parallel inspector is better than with sequential inspector, the experimental results are not satisfactory. The results can be improved by using various scheduling strategy or adding the number of CPU.

4.4 Related Work

The Parafrase-2 compiler, a famous parallelizing compiler developed by Illinois University, aims at developing a source-to-source multilingual restructuring compiler and provides a reliable, portable, easy to extend, and powerful research tool for exploring program transformation about parallelizing compiler. Normally, the parallelizing compiler consists of two principal components, the *front-end* and the *back-end*. The front-end of parallelizing compiler consists of the *scalar analysis* and *data dependence analysis*, which detect dependence relations between procedures or statements and extract the parallelizable code segments for back-end to generate parallel executable codes. The task of the back-end is to generate parallel machine codes for some multiprocessor systems from its intermediate representation by using the analysis results

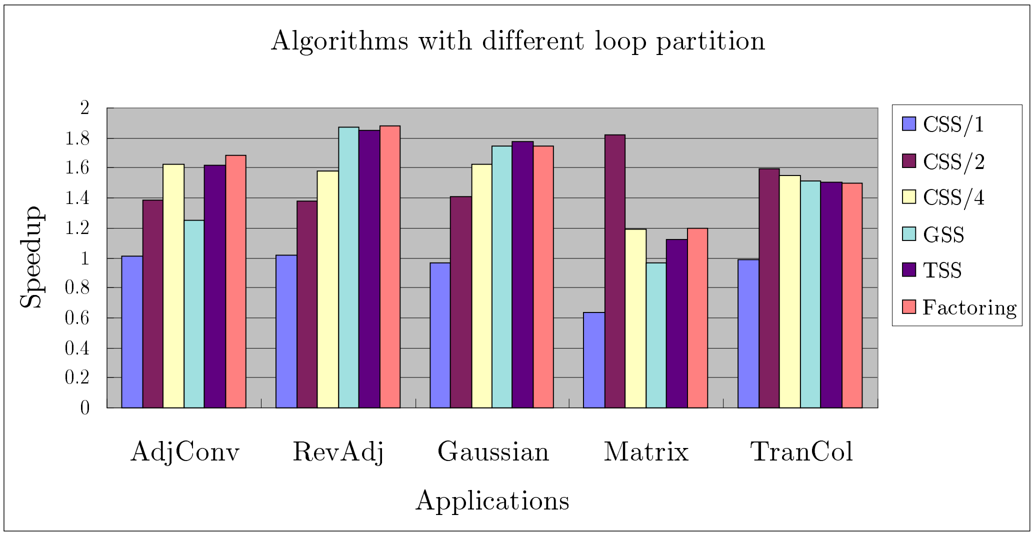


Figure 17: Result of different loop-partitioning algorithms.

gathered in the front-end [5, 6]. This compiler supports preprocessor and postprocessor for C and FORTRAN languages together with an intermediate representation. The preprocessor is used to transform each input source code to the intermediate representation together with its corresponding set of data structures, and the postprocessor is used to recreate the source program. The Parafrase-2 compiler is executed by a *pass list file* consisting of several passes, each of which operates on the data structures and transforms the input into some suitable form for subsequent execution. Parafrase-2 compiler coded by C language has a convenient user interface, provides a means for user interaction at several levels during the transformation processes, and also has high portability.

A new parallelizing compiler, called Polaris, is developed at the Center for Supercomputing Research and Development (CSRD) in University of Illinois [1]. Polaris includes a powerful basic infrastructure for manipulating FORTRAN programs and a number of improved analysis and transformation passes, notable subroutine inline expansion, symbolic analysis, induction and reduction variable recognition, data dependence analysis, array privatization, and runtime analysis. The most important techniques implemented in Polaris resulted from a study of the effectiveness of commercial Fortran parallelizers. They compiled the Perfect Benchmarks, a collection of conventional Fortran programs representing the typical workload of high-performance computers, for the Alliant FX/80, an eight-processor multiprocessor popular in the late 1980s. For each program, they measured the quality of the parallelization by computing the speedup the ratio of a program's sequential execution time to the execution time of the automatically parallelized version. Their study showed that extending the four most important analysis and transformation techniques traditionally used for vectorization leads to significant increases in speedup. However, it is important to note that Polaris' innovation is in improved recognition of parallelism, which is a necessary step for porting programs to any parallel machine available today.

Because independently developing an entire infrastructure is prohibitively expensive, compiler re-

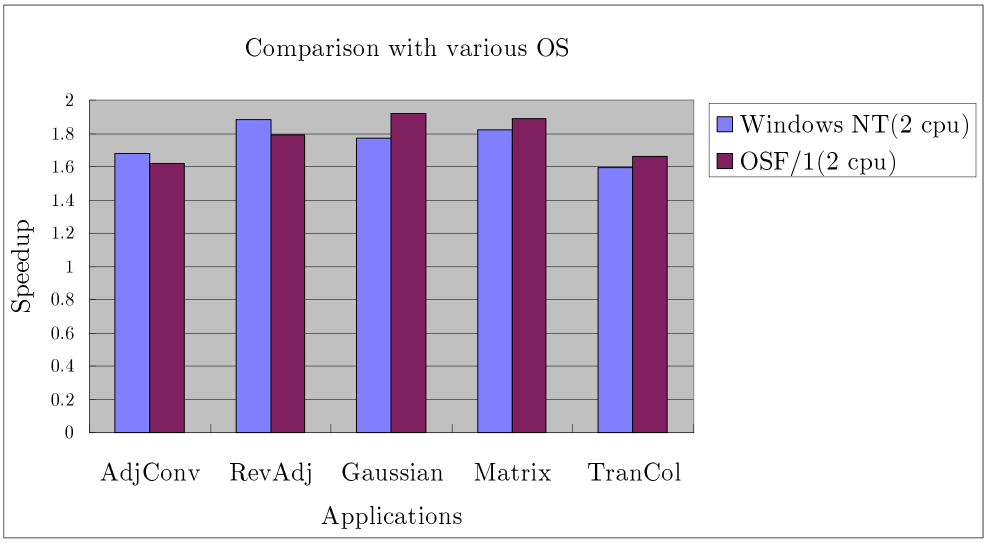


Figure 18: Comparison of speedup between OSF/1 and Windows NT.

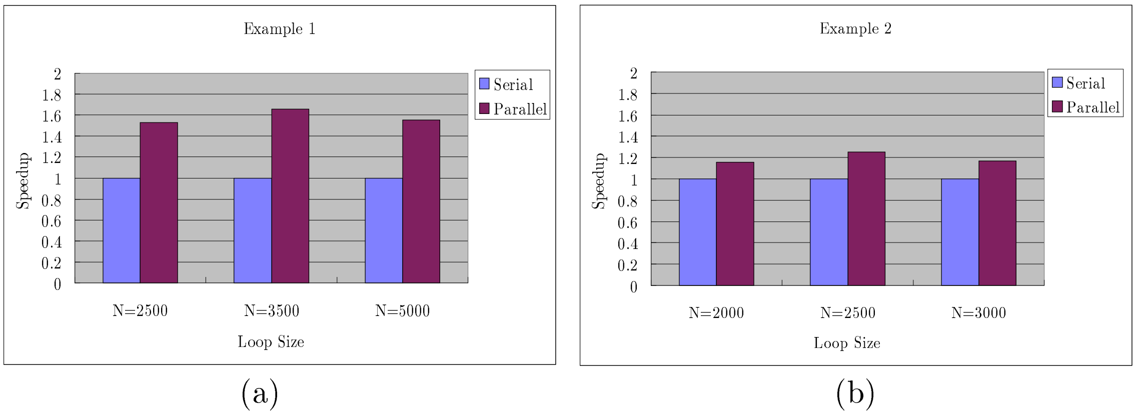


Figure 19: Results of DOACROSS examples.

searchers would benefit greatly from sharing investments in infrastructure development. Toward that end, they are making the SUIF (Stanford University Intermediate Format) compiler system available to others. They have developed SUIF as a platform for their research on compiler techniques for high-performance machines. It is powerful, modular, flexible, clearly documented, and complete enough to compile large benchmark programs. Their group has successfully used SUIF to perform research on topics including scalar optimizations, array data dependence analysis, loop transformations for both locality and parallelism, software prefetching, and instruction scheduling. Ongoing research projects using SUIF include global data and computation decomposition for both shared and distributed address space machines, communication optimizations for distributed address space machines, array privatization, interprocedural parallelization, and efficient pointer analysis. The SUIF toolkit contains a variety of compiler passes. Fortran and ANSI C front ends are available to translate source programs into SUIF. The system includes a parallelizer that

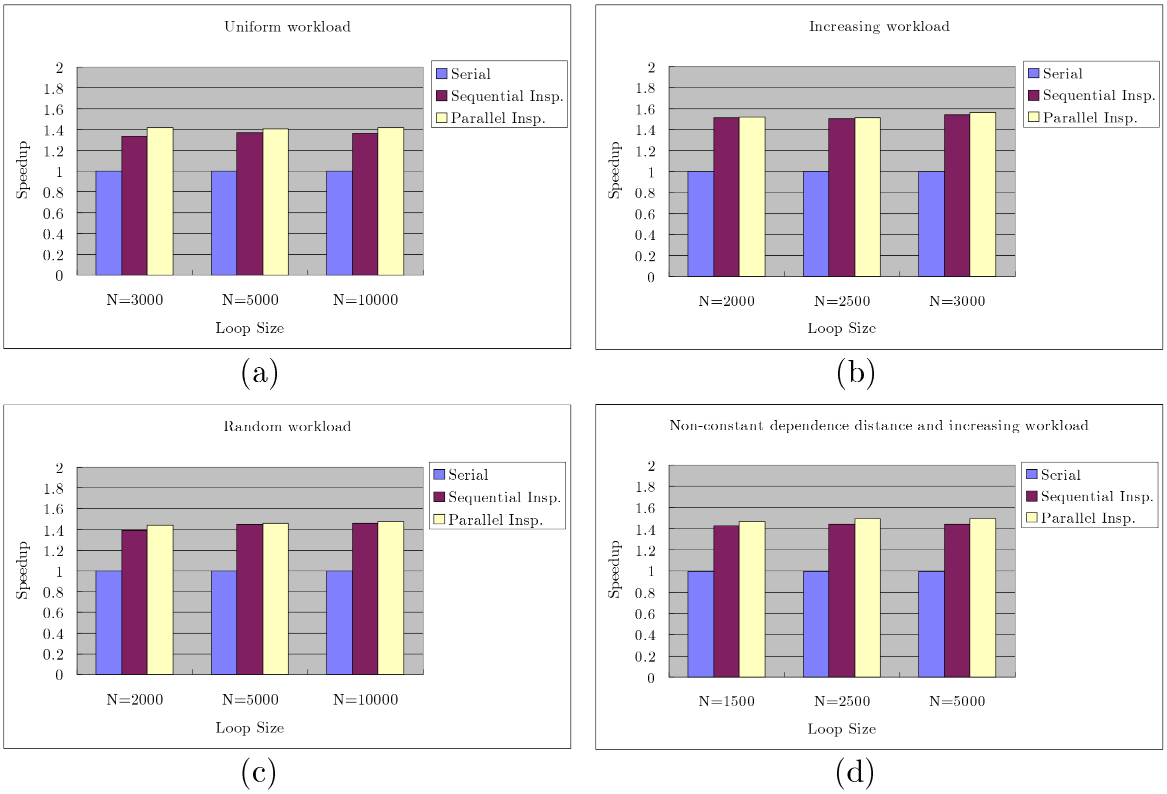


Figure 20: Results of runtime parallelization.

can automatically find parallel loops and generate parallelized code. A SUIF-to-C translator allows us to compile the parallelized code on any platform to which our parallel run-time library has been ported. The system provides many features to support parallelization: data dependence analysis, reduction recognition, a set of symbolic analyses to improve the detection of parallelism, and unimodular transformations to increase parallelism and locality. Scalar optimizations such as partial redundancy elimination and register allocation are also included.

In our PPD, the ZIV/I test is used for checking if the linear equation formed by array subscript has an appropriate integer solution. Besides, we also proposed two *ad hoc* techniques that look for the trivial contradiction on direction vectors to improve the drawbacks of traditional subscript-by-subscript testing mechanism. PPD could detect the DOALL loops and DOACROSS loops which include synchronization directives. In Parafrase-2, only GCD Test and Banerjee Test are employed on the builddep pass. The accuracy of GCD and Banerjee tests is less than that uses the I test. Besides, Parafrase could not detect the DOACROSS loops in a source program.

Traditionally, the parallelizing compiler dispatches the loop by using only one scheduling algorithm, either static or dynamic. However, programs have different kinds of loops, including uniform workload, increasing workload, decreasing workload, and random workload, loops, and every scheduling algorithm can achieve good performance on a different loop style. To reduce the overhead and enhance load balancing, the knowledge-based approach is feasible solution for parallel loop scheduling. An approach that integrates existing static and dynamic scheduling algorithms and makes good use of their advantages is proposed in the paper. We can use this approach to choose an appropriate scheduling algorithm base on some features that include the loop style, loops bound, system status, data locality, and synchronization overhead, and then apply the resulting algorithm to schedule the DOALL loop on processors. In this paper, we concentrate on the fundamental phase, parallel loop scheduling, in parallelizing compilers running on multiprocessor systems. A new model exploiting loop parallelization using knowledge-based techniques is proposed. The knowledge-based approach integrates existing loop schedules to make good use of their abilities in extracting more parallelism. Experimental results show that the high speedup obtained by using IPLS on multiprocessors is obvious. Furthermore, for system maintenance and extensibility, our approach is obviously superior to others. In addition, a run-time technique based on the inspector-executor scheme is proposed to find available parallelism on loops. Our inspector can determine the wavefronts by building a DEF-USE table for each loop of a program. The process of the inspector for finding the wavefronts can be parallelized fully without any synchronization. Our executor can execute the loop iterations concurrently. Additionally, our compiler is highly modularized so that porting to other platforms is very easy, and it can partition parallel loops into multithreaded codes based on several loop-partitioning algorithms. The experimental results clearly show that the compiler achieves good speedup on Windows NT OS.

5 Conclusions

This paper describes the design and implementation of an efficient and parallelizing compiler to parallelize loops and achieve high speedup rates on multiprocessor systems. We first introduce how to design a portable FORTRAN parallelizing compiler (PFPC) on a multiprocessor system by multithreading operating system OSF/1. The main contribution of this paper is described as follows. A model of FORTRAN parallelizing compiler on multithreading OSF/1 was also proposed in this paper. This paper also reviewed the practical parallel loop detector (PPD) that was implemented in PFPC on finding the parallelism in loops as well. Furthermore, if DOACROSS loops are available, an optimization of synchronization statements are made. Experimental results showed that PPD was more reliable and accurate than previous approaches. In addition, a new model by using knowledge-based techniques was proposed to exploit more loop parallelisms in this paper. A new model exploiting loop parallelization using knowledge-based techniques is proposed. The knowledge-based approach integrates existing loop schedules to make good use of their abilities in extracting more parallelism. Experimental results show that the high speedup obtained by using IPLS on multiprocessors is obvious. Furthermore, for system maintenance and extensibility, our approach is obviously superior to others. In addition, a run-time technique based on the inspector-executor scheme is proposed to find available parallelism on loops. Our inspector can determine the wavefronts by building a DEF-USE table for each loop of a program. The process of the inspector for finding the wavefronts can be parallelized fully without any synchronization. One of the ultimate goals is to construct a high-performance and portable FORTRAN parallelizing compiler on shared-memory multiprocessor systems.

References

[1] W. Blume, R. Eigenmann, J. Hoeflinger, and D. Padua, P. Petersen, L. Rauchwerger, P. Tu, "Automatic detection of parallelism: A grand challenge for high-performance computing," *IEEE Parallel & Distributed Technology*, 2(3):37-47, Fall 1994.

[2] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, *Programming under Mach*, Addison Wesley, 1993.

[3] K. D. Cooper *et al.*, "The ParaScope parallel programming environment," *Proc. IEEE*, 81(2):244-263, Feb. 1993.

[4] Y. W. Fann, C. T. Yang, C. J. Tsai, and S. S. Tseng, "IPLS: An intelligent parallel loop scheduling for multiprocessor systems," *Proc. of ICPADS'98*, Tainan, Taiwan, pp. 7751-782, Dec. 1998.

[5] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, MA, 1988.

[6] C. D. Polychronopoulos, "Parafrase-2: An environment for paralleling, partitioning, synchronizing, and scheduling programs on multiprocessors," *Int'l. J. High Speed Computing*, 1(1):45-72, Jan. 1989.

[7] L. Rauchwerger, N. M. Amato, and D. Pauda, "Run-time methods for parallelizing partially parallel loops," in *Proc. 1995 Int'l. Conf. Supercomputing*, Barcelona, Spain, July 1995.

[8] R. P. Wilson *et al.*, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, 29(12):31-37, Dec. 1994.

[9] M. Wolfe, *High-Performance Compilers for Parallel Computing*, 137-162, Addison-Wesley Publishing, New York, 1995.

[10] C. T. Yang, S. S. Tseng, and C. S. Chen, "The anatomy of parafrase-2," *Proceedings of the National Science Council Republic of China (Part A)*, 18(5):450-462, Sep. 1994.

[11] C. T. Yang, S. S. Tseng, C. D. Chuang, and W. C. Shih, "Using knowledge-based techniques on loop parallelization for parallelizing compilers," *Parallel Computing*, 23(3):291-309, May 1997.

[12] C. T. Yang, S. S. Tseng, and M. C. Hsiao, "A model of parallelizing compiler on multithreading operating systems," *Int'l. J. of Modelling and Simulation*, 18(1):9-15, 1998.

[13] C. T. Yang, S. S. Tseng, M. H. Hsieh, and S. H. Kao, "An efficient run-time parallelization for do loops," *J. of Info. Sci. and Eng. — Special Issue on Compiler Techniques for High-Performance Computing*, vol. 14, no. 1, pp. 237-253, 1998, the previous version in *Proc. ICPADS'97*, Korea, pp. 308-313, Dec. 1997.

[14] C. T. Yang, S. S. Tseng, M. C. Hsiao, and S. H. Kao, "A portable parallelizing compiler with loop partitioning," to appear in *Proceedings of the National Science Council Republic of China (Part A)*, 1999.

[15] C. T. Yang, C. T. Wu, and S. S. Tseng, "PPD: A practical parallel loop detector for parallelizing compilers on multiprocessor systems," *IEICE Trans. Information and Systems*, vol. E79-D, no. 11, pp. 1545-1560, Nov. 1996, the previous version in *Proc. ICPADS'96*, 274-281, Japan, June 1996.

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

Table 8: The comparative result using EISPACK.

LINPACK subroutines list	Parafrase-2 no. of DOALL	PPD no. of DOALL	PPD no. of DOACR	LINPACK subroutines list	Parafrase-2 no. of DOALL	PPD no. of DOALL	PPD no. of DOACR
bakvec.f	2	2	1	balanc.f	5	5	0
balbak.f	1	1	0	bandr.f	8	8	0
bandv.f	3	3	0	bisect.f	1	1	0
bqr.f	5	5	0	cbabk2.f	1	1	0
cbal.f	5	5	0	cdiv.f	0	0	0
cg.f	0	0	0	ch.f	2	2	0
cinvit.f	5	5	0	combak.f	1	1	0
comhes.f	2	2	0	comlr2.f	12	12	0
comlr.f	4	4	0	comqr2.f	9	9	0
comqr.f	2	2	0	cortb.f	2	2	0
corth.f	2	2	0	csroot.f	0	0	0
elmbak.f	1	1	0	elmhes.f	2	2	0
eltran.f	4	4	0	epslon.f	0	0	0
figi2.f	1	1	0	figi.f	0	0	0
foo.f	1	1	0	hqr2.f	6	6	0
hqr.f	2	2	0	htrib3.f	3	3	0
htribk.f	3	3	0	htrid3.f	0	0	0
htridi.f	3	3	0	imtql1.f	0	0	1
imtql2.f	0	0	1	imtqlv.f	1	1	0
intvit.f	6	6	0	minfit.f	10	10	0
ortbak.f	2	2	0	orthes.f	2	2	0
ortran.f	4	4	0	otqlrat.f	2	2	1
pythag.f	0	0	0	qzhes.f	5	5	0
qzit.f	0	0	0	qzval.f	0	0	0
qzvec.f	2	2	0	ratqr.f	3	3	0
rebak.f	0	0	0	rebakb.f	0	0	0
reduc2.f	0	0	0	reduc.f	0	0	0
rg.f	0	0	0	rgg.f	0	0	0
rs.f	0	0	0	rsb.f	0	0	0
rsg.f	0	0	0	rsgab.f	0	0	0
rsgba.f	0	0	0	rsm.f	0	0	0
rsp.f	2	2	0	rst.f	2	2	0
rt.f	0	0	0	svd.f	14	14	0
tinvit.f	5	5	0	tql1.f	1	1	1
tql2.f	1	1	1	tqlrat.f	2	2	1
trbak1.f	1	1	0	trbak3.f	0	0	0
tred1.f	5	5	0	tred2.f	9	9	0
tred3.f	1	1	0	tridib.f	1	1	0
tsturm.f	6	6	0				