JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 169-200 (2000)

An Intelligent Parallel Loop Scheduling for Parallelizing Compilers*

YUN-WOEI FANN, CHAO-TUNG YANG+,
SHIAN-SHYONG TSENG AND CHANG-JIUN TSAI
Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.
+ROCSAT Ground System Section
National Space Program Office
Hsinchu, Taiwan 300, R.O.C.

In this paper we propose a knowledge-based approach to solving loop-scheduling problems. A rule-based system, called IPLS, is developed by combining a repertory grid and an attribute ordering table to construct a knowledge base. IPLS chooses an appropriate scheduling algorithm by inferring some features of loops and assigning parallel loops to multiprocessors to achieve significant speedup. Because more attributes are proposed, the accuracy of selection of an appropriate scheduling method is improved. In addition, the refined IPLS system can automatically adjust the attributes in the knowledge base according to profile information; therefore, IPLS has the capability of feedback learning. The experimental results show that our approach can achieve greater speedup on multiprocessor systems than can others.

Keywords: parallelizing compiler, parallel loop scheduling, knowledge-based system, multiprocessor systems, speedup

1. INTRODUCTION

Parallel processing has been one of the most important technologies in modern computing for several decades. Many powerful multiprocessor hardware systems have been built to exploit parallelism for concurrent execution. Two models are classified below according to their memory organization, addressing schemes, and inter-processor communication mechanisms:

- The uniform memory access (UMA) model: Most UMA systems as shown in Fig. 1(a) are shared-memory environments, in which all processors use the common memory and have equal time to access all memory words.
- The non-uniform memory access (NUMA) model -In NUMA system shown in Fig. 1 (b), the cost of accessing memory increases with the distance between the accessing processor and target memory.

Received March 31, 1999; revised June 16 & July 27, 1999; accepted September 20, 1999.
Communicated by Shang-Rong Tsai.
*This work was supported in part by the NSC of the ROC under Grant No. NSC87-2213-E-009-023. A preliminary version of this paper, entitled "IPLS: An Intelligent Parallel Loop Scheduling for Multiprocessor Systems," appeared in *Proceedings of the 1998 International Conference on Parallel and Distributed Systems* (*ICPADS '98*), Taiwan, pp. 775-782, 1998.

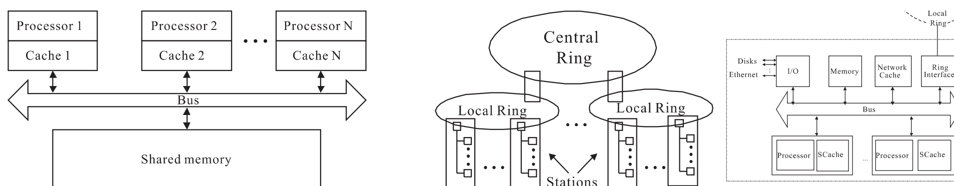


Fig. 1. The UMA and NUMA systems.

In view of the difference between these two architectures, the memory access and interprocessor communication overheads should be taken into consideration when accessing the shared variables. To speed up multiprocessor systems, it is likely that tasks must be decomposed into several sub-tasks and executed in parallel on different processors. Parallelizing compilers and parallel programming tools that can translate ordinary programs into parallel codes have been proposed. Parallelizing compilers can analyze sequential programs to detect hidden parallelism. The information is used to automatically restructure sequential programs into parallel sub-tasks for multiprocessors using scheduling algorithms. Therefore, it is important to design and implement efficient parallelizing compilers that can extract the maximum amount of parallelism for multiprocessors.

An efficient approach to extracting potential parallelism is to concentrate on the parallelism available in the loops. Since the body of a loop may be executed many times, loops often comprise a large portion of a program's parallelism. By definition, a loop is called a DOALL loop if there is no cross-iteration dependence in the loop; i.e., all the iterations of the loop can be executed in parallel. If all the iterations of a DOALL loop are distributed among different processors as evenly as possible, a high degree of parallelism can be exploited. Parallel loop scheduling is a method that schedules a DOALL loop on multiprocessor systems as evenly as possible.

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 [1, 2, 4-6, 9-11, 14, 19-21]. However, it has the drawback of creating load imbalances when the loop style is not uniformly distributed, when the loop bounds cannot be known at compile-time, or when locality management cannot be exercised. In contrast, dynamic scheduling is more appropriate for load balancing; however, the runtime overhead must be taken into consideration. In general, parallelizing compilers distribute loop iterations by using only one kind of scheduling algorithm, which maybe static or dynamic. However, a program may have different loop styles, including a uniform workload, an increasing workload, a decreasing workload, and a random workload. Every scheduling algorithm can achieve good performance with some loop styles and different system states because the loop style and the system environment influence the selection of scheduling strategies. For example, it is not appropriate to apply GSS on a NUMA system because the communication cost of accessing remote memory is too high.

The scheduling performance of a DOALL loop on shared-memory multiprocessor systems is usually dependent upon the amounts of overhead that arise from four main factors: load imbalances among processors, the synchronization overhead, the communication overhead, and the thread management overhead. They are described as follows:

- Load imbalances: If some processors are idle, we cannot take full of advantage of their multiprocessing capacity. A good scheduling algorithm tries to spread the workload around to multiprocessor systems as evenly as possible. To avoid uneven assignment of work units to processors, many loop-scheduling algorithms use a central work queue for remainder iterations.
- Synchronization overhead: This type of overhead arises from simultaneous accesses made by different processors of a set of shared variables that contain the iteration indices.
- Communication overhead: This is non-uniform data access time among multiprocessor systems and is often caused by having to access remote memory for non-local data. Data locality management policies attempt to minimize the communication overhead by allocating iterations close to their data.
- Thread management overhead: This refers to the time required to create, detach and schedule multiple concurrent lightweight processes to express the concurrency on multithreaded operating systems. It is well known that the system performance will drop if too many threads are created for execution.

It is difficult to balance the tradeoff among these factors; therefore, this paper concentrates on how to distribute parallel loop iterations on a shared-memory multiprocessor system not only as evenly as possible, but also with the lowest overhead. For example, the block size should be large enough to reduce the synchronization overhead, but load imbalance and the communication overhead may become extreme. As described above, none of the scheduling algorithms is best for all cases. That is, all of the algorithms are only appropriate for some cases, and none can manage all these features well. Therefore, finding a good trade-off among them is not an easy task. Suppose the parallelizing compiler can analyze loop attributes, such as the loop style, loop bound, data locality, etc.; an appropriate scheduling algorithm for this particular case can be determined. This leads to selecting scheduling algorithms based on a knowledge-based system approach.

The scheme proposed in [19] is called Knowledge-Based Parallel Loop Scheduling (KPLS). KPLS uses knowledge-based techniques to select an adequate loop-scheduling algorithm for a loop according to the attributes of the loop behaviors and system states. However, in KPLS, some attributes that influence the accuracy of selecting an adequate loop-scheduling algorithm are neglected. Therefore, in this paper, we propose a new approach, called the Intelligent Parallel Loop Scheduling (IPLS) algorithm. It considers more attributes so as to make up for the shortcoming of KPLS. IPLS also integrates more existing loop-scheduling algorithms than KPLS does for UMA and NUMA systems to make good use of their advantages in loop parallelism. Because more attributes are proposed, the accuracy of selection of an appropriate scheduling method is improved. In addition, the refining system in IPLS can automatically adjust the attributes in the knowledge base according to profile information. Therefore, it has a feedback-learning mechanism. The experimental results show that our approach can achieve more speedup on multiprocessor systems than others can. Furthermore, our approach is obviously superior to others in terms of system maintenance and extensibility. Once a new scheduling algorithm or techniques is proposed, it can be easily integrated into IPLS by adding knowledge and rules to improve the power of IPLS.

2. BACKGROUND

The major source of parallelism in a program is loops. If the loop iterations can be distributed to different processors as evenly as possible, the parallelism within loop iterations can be exploited. Parallel loop scheduling is used to achieve this goal by determining how to assign the DOALL loops onto each processor in a balanced fashion so as to effect a high level of parallelism with the least amount of overhead. In a shared-memory multiprocessor system, two kinds of parallel loop scheduling decisions can be made either statically at compile-time or dynamically at run-time. In the rest of this section, we will review the various scheduling algorithms. We use N and P to denote, respectively, the number of iterations and the number of processors, and we set the size of the ith partition to K_i.

2.1 Static Scheduling

Static scheduling [6] makes a scheduling decision at compile-time and uniformly distributes loop iterations onto processors. It is applied when each loop iteration takes roughly the same amount of time, and the compiler knows how many iterations must be run and how many processors are available for use at compile-time. It has the advantage of incurring the minimum scheduling overhead, but load imbalances may occur. However, static scheduling may perform unacceptably when the loop style is not uniformly distributed or the loop bounds can not be known at compile-time. In the following, the different static scheduling methods with and without consideration of data locality are reviewed.

Block Scheduling In block scheduling [6], N iterations are divided into $\left\lceil \frac{N}{P} \right\rceil$ rounds. Each round consists of consecutive iterations and is assigned to one processor. This is only suitable for uniformly distributed loop iterations.

Cyclic Scheduling Instead of assigning to a processor a consecutive block of iterations, iterations are assigned to different processors in a cyclic fashion [6]; i.e., iteration i is assigned to processor $i \bmod P$. This method may produce a more balanced schedule than block scheduling for some non-uniformly distributed parallel loops.

Block-D Scheduling In NUMA systems, managing data locality is important due to the increased cost of accessing remote memory. For good performance, it is essential that the loop partitioning match the data partitioning. If both the data partitioning and loop scheduling occur in blocks, the static scheduling is called Block-D scheduling [6].

Cyclic-D Scheduling As mentioned above, if both the data partitioning and loop scheduling occur in a cyclic fashion, the static scheduling is called Cyclic-D scheduling [6].

2.2 Dynamic Scheduling

Dynamic scheduling adjusts the schedule during execution whenever it is uncertain how many iterations to expect or when each iteration will take a different amount of time due to a branching statement inside the loop. Although it is more suitable for load balancing between processors, runtime overhead and memory contention must be considered.

Dynamic scheduling algorithms, such as SS [12], fixed-size chunking [3, 11], GSS [8], factoring [2], and TSS [11], share the following same characteristics. They always maintain a global queue containing indices of iterations. At runtime, when a processor is idle, it issues synchronous operations to the global queue and fetches some iterations for execution.

Pure Self-Scheduling (SS) This is the easiest and most straightforward dynamic loop scheduling algorithm [12]. Whenever a processor is idle, an iteration is allocated to it. This algorithm achieves good load balancing but also introduces excessive overhead.

Chunk Self-Scheduling (CSS) Instead of allocating one iteration to an idle processor as in self-scheduling, CSS allocates k iterations each time, where k, called the chunk size, is fixed and must be specified by either the programmer or the compiler [3, 11]. When the chunk size is one, this scheme is pure self-scheduling, as discussed above. If the chunk size is set to the bound of the parallel loop equally divided by the number of processors, the scheme becomes static scheduling. A large chunk size will cause load imbalancing while a small chunk is likely to produce too much scheduling overhead. For different partitioning schemes, we adapted CSS/l, which is a modified version of CSS, where l means the number of chunks.

Enhanced Chunk Self-Scheduling (ECSS) When all the dependent iterations of a loop are assigned to the same processor, the dependent relation is satisfied and it does not need synchronization operations to keep the executing order of the loop. Let chunk size be K. If a loop exist loop carried dependence distance D, and if every time each processor gets K iterations that mutually are a distance D from work queue, then the dependent relation of K iterations are reserved without a synchronization operation. For the loop whose LCD distance is larger than one, ECSS can develop more parallelism than CSS and reduce the amount of synchronization overhead.

Guided Self-Scheduling (GSS) This algorithm can dynamically change the number of iterations assigned to each processor [8]. More specifically, the next chunk size is determined by dividing the number of remaining iterations of a parallel loop by the number of available processors. The property of decreasing chunk size implies an effort is made to achieve load balancing and to reduce the scheduling overhead. By allocating large chunks at the beginning of a parallel loop, one can reduce the frequency of mutually exclusive accesses to shared loop indices. The small chunks at the end of a loop partition serve to balance the workload across all the processors.

Multilevel Interleaved Guided Self-Scheduling (MIGSS) This is a hybrid scheduling scheme blended with run-time scheduling techniques and compile-time loop restructuring [15]. Its run-time scheduling is based on guided self-scheduling. A compile-time loop transformation method is used to enhance the parallel execution performance. The basic idea of MIGSS is to split a hybrid perfectly nested loop into several independent loops and then apply high-level spreading to the generated loops. Since the resulting loops are independent, loop splitting must be applied to outer DOALL loops only. As in the GSS scheme, loop interchange and loop coalescing can be applied to the resulting loops to reduce the number of synchronization points.

Factoring In some cases GSS might assign too much work to the first few processors, so that the remaining iterations are not time-consuming enough to balance the workload. This situation arises when the initial iterations in a loop are much more time-consuming than later iterations. The factoring algorithm addresses this problem [2]. 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 loads better than GSS does when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of Factoring is not significantly larger than that of GSS.

Trapezoid Self-Scheduling (TSS) This approach tries to reduce the need for synchronization while still maintaining a reasonable load balance [11]. TSS(N_s, N_f) assigns the first N_s iterations of a loop to the processor starting the loop and the last N_f iterations to the processor performing the last fetch, where N_s and N_f are both specified by either the programmer or the parallelizing compiler. This algorithm allocates large chunks of iterations to the first few processors and successively smaller chunks to the last few processors. Tzen and Ni proposed TSS($N/2P$, 1) as a general selection. In this case, the first chunk is of size $\frac{N}{2P}$, and consecutive chunks differ in size $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in TSS whereas it is a decreasing function in GSS and in Factoring.

Self-Adjusting Scheduling (SAS) Hamidzadeh and Lilja introduced the SAS technique, which is capable of improving the performance of programs on NUMA systems by employing an on-line optimization technique on a dedicated processor [1]. The object of this algorithm is to address the tradeoffs between three interrelated factors in dynamic scheduling, namely the remote memory access delay, load imbalancing, and scheduling costs, to compute schedules that result in minimum total loop execution times on a multiprocessor system. SAS is an on-line branch-and-bound algorithm that searches through a space of all possible partial and complete schedules. The overlapping scheduling and execution, along with self-adjustment of the durations of partial scheduling periods reduces scheduling and synchronization costs significantly. To satisfy load balancing and locality management, SAS introduces a unified cost model that accounts for both of these factors simultaneously.

Safe Self-Scheduling (SSS) The basic idea behind SSS is to assign to each processor the largest number, m, of consecutive iterations having a cumulative execution time just exceeding the average processor workload $\frac{E}{P}$ [5]; i.e., $\sum_{i=s}^{s+m-1} e(i) < \frac{E}{P} < \sum_{i=s}^{s+m} e(i)$, where $E = \sum_{i=1}^{N} e(i)$ and s is some starting iteration number of the chore. m is the best choice of chore size, which is the *smallest critical chore size*. SSS is similar to Factoring but improves the shortcoming of Factoring. In the implementation of SSS, $\alpha \times \frac{N}{P}$ iterations are given to each processor at compile time, where $0 < \alpha \le 1$. (For most of the applications we have encountered, $0.5 \le \alpha < 1$.) At run time, an idle processor fetches more unscheduled iterations. The ith fetching processor is assigned a chunk of $\max\left\{(1-\alpha)^{\lceil \frac{i}{P} \rceil} \times \frac{N}{P} \times \alpha, k\right\}$, where k is used to control granularity.

Adaptive Hybrid Scheduling (AHS) One solution to workload imbalance on every processor is to adopt a hybrid scheduling mechanism. This mechanism distributes the workload as much as possible at compile-time based on not breaking the load balance among all processors, and it schedules the left workload at run time. Let E_{max}, E_{min} and E_{avg} be the maximum, average and minimum execution times of every iteration, respectively; obviously, $E = N \times E_{avg}$. A block is defined as a number of successive iterations. The jth block is denoted as B_j, and the number of iterations included in B_j is called the block size, denoted as $|BS_j|$. In AHS, the first block B_1 is assigned to processor p_1, the second block B_2 is assigned to processor p_2, and so on. Every m consecutive blocks forms a round. A large number, $|BS_0| = (N/P) \times \beta + E/(E_{\max} \times p) \times \gamma = (N/P) \times \omega$, of iterations is distributed in round 0, where β is the probability not to fetch again and γ is the probability to fetch again, obviously $\beta + \gamma = 1$. β and γ are selected by programmers according to the properties of parallel computers. The other iterations are left for later rounds. $|BS_i| = (N_r/P) \times \beta + E/(E_{\max} \times p) \times \gamma = (N_r/P) \times \omega$, N_r is the number of remaining iterations.

Table 1. Various loop scheduling algorithms.

Scheme	Formulas
SS	$K_i = 1$
CSS(k)	$K_i = k$
CSS/l	$K_i = \left\lceil \frac{N}{l} \right\rceil$
GSS	$K_i = \left\lceil \frac{R_i}{P} \right\rceil, R_0 = N, R_{i+1} = R_i - K_i$
Factoring	$K_i = \left(\frac{1}{2}\right)^{\left\lceil \frac{i}{p} \right\rceil} \frac{N}{P}$
TSS(f,l)	$K_i = f - i\delta, I = \left\lceil \frac{2N}{f+l} \right\rceil, \delta = \frac{f-l}{I-1}$
SSS	$K_i = \max\left\{(1-\alpha)^{\left\lceil \frac{i}{P} \right\rceil} \times \frac{N}{P} \times \alpha, k\right\}$
AHS	$K_0 = (N/P) \times \beta + E/(E_{max} \times p) \times \gamma = (N/P) \times \omega$ $K_i = \max\{R_i/P \times \omega, k\}, R_{i+1} = R_i - K_i, R_0 = N$

Table 2. Sample partition sizes.

Scheme	$N = 1000$ and $P = 4$
SS	111111111111...
CSS(125)	125 125 125 125 125 125 125 125
CSS/4	250 250 250 250
GSS	250 188 141 106 79 59 45 33 25 ...
Factoring	125 125 125 125 62 62 62 62 31 ...
TSS(88,12)	88 84 80 76 72 68 64 60 ... 12
SSS	188 188 188 188 47 47 47 47 12 12 ...
AHS	188 188 188 188 16 16 16 16 11 11 ...

Formulas for calculating K_i in different algorithms are listed in Table 1. Table 2 gives sample partition sizes for SS, CSS(125), CSS/4, GSS, Factoring, TSS(88, 12), SSS, and AHS, when $N = 1000$ and $P = 4$.

2.3 Affinity Scheduling

As modern shared memory multiprocessor systems have high and non-uniform memory access costs, these costs gradually dominate the source of a parallel application's execution. To reduce the remote memory access cost, affinity scheduling algorithms partition and schedule the loop iterations to the local work queue of each processor. The data for iteration is placed on the cache of some dedicated processor to be used repeatedly. Therefore, they are more suitable for a NUMA machine that takes data locality into account, but load balancing between the processors, runtime overhead and memory access rate must be considered.

Affinity Scheduling (AFS) Most of the dynamic loop scheduling methods work well only in UMA share-memory systems; in NUMA systems, the overhead of communication for accessing remote memory is too heavy and more important. While most existing dynamic scheduling algorithms fail to take locality into account, there is one method, called AFS [7], that gives us a useful alternative. Markatos and LeBlanc proposed this algorithm, which takes locality into account [7]. AFS consists of two phases: *initialization* and *execution*. In the initialization phase, AFS divides the iterations of a loop into chunks of size $\left\lceil \frac{N}{P} \right\rceil$. The ith chunk of iterations is always placed in the local work queue of processor i. During the execution phase, when a processor is idle, the processor removes $\frac{1}{k}$ (in general, we assume that $k = P$) of the remaining iterations from its local work queue and executes them. If a processor's work queue is empty, the scheduling process finds the most loaded processor, removes $\left\lceil \frac{1}{P} \right\rceil$ of the remaining iterations from that processor's work queue, and executes them.

Modified Affinity Scheduling (MAFS) Because the migration strategy of AFS is too conservative, a mortified policy called MAFS [13] to fix the migration algorithm has been. The main difference between AFS and MAFS is in the migration policy. MAFS assigns a more appropriate quantum than AFS when migrating work between processors. For an idle processor, instead of taking $\frac{1}{P}$ iterations from the most loaded processor, i, it removes $\min(N_i^1, N_i^{most} - N_i^1)$ iterations, where N_i^1 equals $\left\lfloor \frac{N_i}{P} \right\rfloor$, N_i^{most}, is the most loaded processor's remaining iterations, and N_i is the total iterations of all processors at time t_i. This scheme combines the advantages of GSS and AFS, reduces the communication cost, avoids using global queues to alleviate contention, and provides better load balancing.

Locality-Based Dynamic Scheduling (LDS) Most of the loops scheduling methods work well only in share-memory systems, but in NUMA systems, the overhead of network communication is too heavy and more important. While most existing dynamic scheduling algorithms fail to take locality into account, there is one method called LDS [6] that offers a good solution to this kind of problem. LDS partitions loops into $\left\lceil \frac{n}{2P} \right\rceil$ sub-tasks, where n is the number of remaining unscheduled iterations. The subtask size is half of GSS, but the iterations are distributed with locality. For example, if the data locality is cyclic and proces-

sor p executes a subtask from 1 to $S1$, then LDS executes the iterations as follows: $p + P$, $p + 2P$, ..., $p + P \times S1$. On the other hand, if the data locality is sequential, the subtask will be executed in the $p \times B + 1, p \times B + 2, \ldots, p \times B + S1$ order, where B is the block size with a number of continuous iterations. When all the local iterations have been executed completely, non-local iterations are acquired from the processor with the most unscheduled iterations.

Dynamic Partitioned Affinity Scheduling (DAFS) The basic idea behind this approach is to dynamically change the partitioned size of AFS scheduling by using the traced record of previous executed iterations [10]. There are three distinct phases in this method. First is the loop initialization phase: it partitions iterations with size $\frac{N}{P}$ to each processor, and this is only done for the first execution of the loop. Second is the loop execution phase: a processor removes $\frac{1}{P}$ iterations from its local work queue and executes them. If a processor's local work queue is empty, the idle processor finds the most heavily loaded processor, migrates $\frac{1}{P}$ of the remaining iterations of that processor and executes them. Every processor keeps track of the actual number of iterations that it has executed. The first two steps are the same as in AFS. Third is the re-initialization phase: before executing the next iteration loop, the loop partition for each processor is re-initialized. Each processor runs the size of iterations, which is last time they were executed. DAFS can dynamically change the partition size in each processor initialization, and the proposed algorithm is more capable of handling workloads that are unbalanced with respect to the amount of computation represented by each iteration.

Clustered Affinity Scheduling (CAFS) A new algorithm called clustered affinity scheduling (CAFS) has been proposed to improve AFS on cluster NUMA machines [16]. In the initialization phase of CAFS, the iterations are divided into C clusters, and a cluster consists of $S = \frac{P}{C}$ processors, where C is about $\lceil \sqrt{P} \rceil$ and P is the number of processors, respectively. Processors 1, 2, ..., and C are assigned to clusters 1, 2, ..., and C in sequence, respectively. But processors $(C + 1)$, $(C + 2)$, ..., and $2C$ are assigned to the clusters in reverse order, and so on.

In the execution phase of CAFS, each time, a processor performs $\lceil \frac{1}{S} \rceil$ of the remaining iterations from its local queue until the local queue is empty, where S is the number of processors in each cluster. If no imbalance occurs, then migration is not needed. When imbalance occurs, the first idle processor migrates $\lceil \frac{1}{S} \rceil$ iterations from the processor with the largest number of non-executing iterations in its local cluster. Instead of searching the other $P-1$ processor of AFS, CAFS searches only the other processors of its cluster.

Localized Affinity Scheduling (LAFS) A new algorithm called localized affinity scheduling (LAFS) has been proposed to improve AFS on a cluster NUMA machines [17]. In the initialization phase of LAFS, the iterations are divided into $\frac{N}{P}$ chunks, where N is the total number of loop iterations and P is the number of processors. If there are C clusters in the NUMA system, then chunk 1, chunk 2, ..., and chunk C are assigned, respectively, to the first

processor of cluster 1, cluster 2, ..., and cluster C in order. Sequentially, chunk $C + 1$, chunk $C + 2$, ..., and chunk $2C$ are assigned to the second processor of cluster 1, cluster 2, ..., and cluster C in sequence, and so on.

In the execution phase of LAFS, each time, a processor performs $\left\lceil \frac{1}{S} \right\rceil$ of the remaining iterations form its local queue until the local queue is empty, where S is the number of processor in each cluster. If no imbalance occurs, then migration is not needed. When imbalance occurs, the first idle processor migrates $\left\lceil \frac{1}{S} \right\rceil$ iterations from the processor with the largest number of non-executing iterations in its local cluster. If the queues of processors in local cluster are all empty, the idle processor migrates $\left\lceil \frac{1}{P} \right\rceil$ iterations from the processor with the largest number of non-executing iterations in other clusters.

Global Distributed Control Scheduling (GDCS) The basic idea behind GDCS is to decentralize the scheduling task among all the processors [4]. The scheme logically organizes P processors in a ring topology. Initially, GDCS assigns $\frac{N}{P}$ iterations to each processor using the static scheduling scheme in the hope that each processor will receive the same size workload. Each processor executes these assigned iterations from its local queue. When a processor P_i becomes idle, it requests extra iterations from successive processors on the ring, P_{i+1}, P_{i+2}, ..., P_N, P_1, ..., P_{i-1}, until it finds an active processor that still has more than β iterations. The active node, processor P_j, dispatches α iterations to this request, where α and β are two threshold values. The processor P_i remembers that processor P_j was the last node from which a request was satisfied. The next time P_i becomes idle, it starts requesting work from processor P_j, by-passing processors between P_i and P_j. An additional feature of this scheme is that if, in the meantime, processor P_j becomes idle and processor P_j remembers that it received tasks from processor P_k, then node P_i will jump from P_j directly to P_k without asking for work from processors between P_j and P_k. Thus, it can avoid unnecessary task searching overheads.

2.4 Knowledge-Based Approach

There are four aspects to the problem of loop scheduling on shared-memory multiprocessors: the synchronization overhead, communication overhead, threaded management overhead, and load balancing. In order to reduce the synchronization overhead, the block size should be large, but load imbalance and the communication overhead may become very serious. Therefore, finding a good trade off between them is not easy. That is, all of the algorithms are only suitable in some cases, and none can do well in all these four aspects.

A rule-based system, called Knowledge-based Parallel Loop Scheduling (KPLS) [19], was proposed to make use of the advantages of typical loop scheduling algorithms for parallelism. KPLS can choose an appropriate scheduling by analyzing the characteristics of the input program and can apply the selected scheduling to assign parallel loops on multiprocessor systems to achieve a high level of speedup.

Because KPLS achieves good performance by selecting an appropriate scheduling algorithm for each executed application, KPLS is very similar to other scheduling algorithms. We show the hierarchy of loop-scheduling algorithms in Fig. 2.

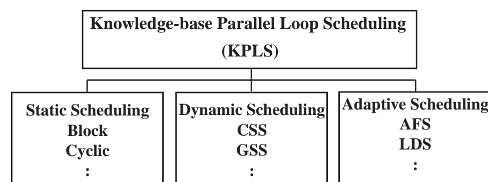


Fig. 2. The hierarchy of loop scheduling algorithms.

3. A NEW APPROACH TO PARALLEL LOOP SCHEDULING

Suppose the parallelizing compiler can analyze a loop's attributes, such as the loop style, loop bound, data locality, etc.; a suitable scheduling algorithm for this particular case can be determined. This leads us to select scheduling algorithms by using a knowledge-based system approach to get reasonable execution results. Below, we shall propose a new approach which uses knowledge-based techniques to construct an intelligent parallel loop scheduling (IPLS).

In this section, we will further propose two methods to enhance the functionality of IPLS. One includes additional attributes that influence the selection of appropriate scheduling algorithm for a parallel loop. If there are more attributes that can be used to increase the accuracy of selecting an appropriate algorithm, the functionality of IPLS can be improved. The other is that a refining system, a new component is developed to improve the inferrence accuracy of IPLS.

3.1 Some Attributes Affecting Scheduling Performance

Four main overheads, the processor load imbalances, synchronization overhead, communication overhead, and thread management overhead, influence the performance of parallel loop scheduling on multiprocessor systems. For example, the block size should be large enough to reduce the synchronization overhead, but the load imbalance and communication overhead may become extreme. As described above, there is no scheduling algorithm that is best for all cases. That is, all of the algorithms are only suitable for some cases, and none can manage all these aspects well. Therefore, finding a trade-off among them is not an easy task. According to the analysis of numerous related researches, eighteen attributes influencing the selection of an adequate loop-scheduling algorithm can be classified into three categories: the system architecture, loop information, and scheduling method. They are discussed in detail below.

3.1.1 The attributes of the system architecture

- Number of processors: The number of 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 a scheduling decision is made at compile time. For dynamic scheduling methods, except CSS, the factors can be omitted.

- Machine model: Because of the difference between UMA and NUMA, the memory access and inter-processor communication overheads should be taken into consideration when accessing shared variables. If a loop has strong data locality, it seems best

to adopt a AFS on NUMA system so that the number of remote memory accesses will be reduced. For a well load-balanced loop, we can consider to using CSS on a UMA system so that the synchronization overhead and communication cost can be reduced.

- Cache size: When a block of data is retrieved and placed in the cache, not only the desired word, but also some adjacent words are retrieved. As the block size increases, the hit ratio will at first increase because of the principle of locality: there is great probability that data in the vicinity of a referenced word will be referenced in the near future. As the block size increases, more useful data is brought into the cache. The relationship between the block size and hit ratio depends on the locality characteristics of a particular program, and no definitive optimum value has been found.

- Memory access rate: The memory access cost for a UMA system and a NUMA System is different because NUMA will spend much more on memory access if the data is missed in cache. When a migration policy is used for AFS, we must set the value of the transfer_limit by considering the memory access rate to avoid unnecessary migration of chunks.

3.1.2 The attributes of loop information

- Loop style: Loops can be roughly divided into four kinds as shown in Fig. 3: uniform workload, increasing workload, decreasing workload, and random workload loops are the most common ones in programs, and should cover most cases. In general, increasing workload and decreasing workload are called uniform workload loops. The static scheduling method is only suitable for uniform workload loops. In contrast, dynamic scheduling is suitable for all loop styles except that GSS is not good for the third kind. It is possible GSS will initially allocate too many iterations to a processor when the third kind of loop is present; that is, GSS may cause load imbalancing.

- Program size: For Enhanced CSS and Factoring, the size of a loop affects the fitness about the adopted scheduling algorithm. If the size of the loop is large, the performance will be improved.

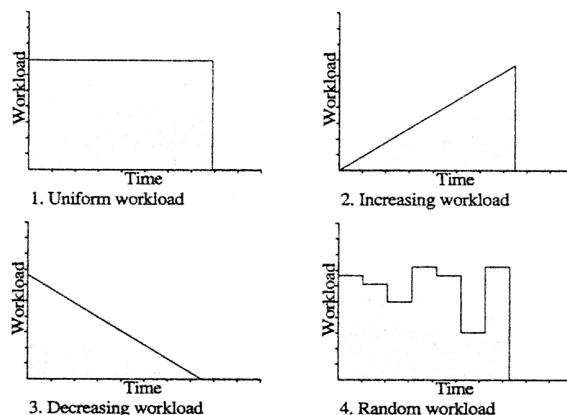


Fig. 3. Four kinds of loops.

- Loop type: The typical applications listed in Table 3 have been studied and implemented in related researches. Some features of these applications, such as the load balance, loop style and affinity, are different. Therefore, each application listed in Table 3 can be set as a specific type of loop. If the experience used to correctly choose an adequate scheduling algorithm for each loop type can be applied to other programs, then the accuracy of the inference engine in IPLS will be enhanced.

- Loop boundary: It is necessary to know the boundary of a loop at compile time before the compiler applies the static scheduling method so that the synchronization overhead can be reduced. On the other hand, this is not necessary for the dynamic scheduling method because the boundary will be known at runtime.

Table 3. 11 types of loops.

No.	Applications	Load Balance	Loop Style	Affinity
1	All-Pairs Shortest Paths	Large	R	Large
2	Transitive Closure	Little	R	Little
3	Reverse Adjoint convolution	Little	I	No
4	Adjoint Convolution	Little	D	No
5	Jacobi Iteration	Little	R	Large
6	Successive Over-Relaxation	Large	U	Large
7	Matrix Multiplication	Large	U	No
8	Gauss Elimination	Large	R	Large
9	Gauss Jordan	Little	R	Large
10	LU Decomposition	Little	D	Large
11	A Loop with Condition	Little	–	–

- Data locality: Among all the scheduling algorithms, only AFS, MAFS, LDS and DAFS consider data locality. They work well when there is strong data locality in the loop. Strong data locality is common in many applications, particularly those that employ iterative algorithms wherein a parallel loop is nested within a sequential loop. Take the program segment of Successive Over-Relaxation shown in Fig. 4 as an example. The ith iteration of the parallel loop always accesses the ith row of the matrix. Thus data locality is the only effect that can be exploited.

- Loop level: If a loop is nested, usually the level of parallelism is large, especially when a parallel loop is embedded within a sequential loop. This is because the more the execution times of iterations of outer loop, the more the occurrence of data locality needed by inner parallel loop in the cache of processors on a UMA machine. Usually, the needed data are the elements of the matrix that are at different locations. This is especially

```
for (i = 1; i <= MAXITERATIONS; i++)//SEQUENTIAL
    for (j = 1; j <= N; j++)//PARALLEL
        for (k = 1; k <= N; k++)//SEQUENTIAL
            A(J, k) = UPDATE(A, j, k)
```

Fig. 4. The program segment of SOR.

obvious on a NUMA machine. For example, the re-initialization phase in DAFS needs nested loops to dynamically change the partition size, so that it can reassign the workload of each processor as balanced a manner as possible.

3.1.3 The attributes of scheduling method

- Start time: If the start times of processors are unequal, static scheduling may not perform well. In contrast, dynamic methods can work well when this unequal processor start time condition is applied.

- Loop carried dependence: The attribute decides if the class of data dependence of a loop is DOALL or DOACROSS. This will influence the parallelism of the program and an adequate scheduling algorithm must be adopted. When the LCD distance of a loop is larger than one, we can apply ECSS to schedule the loop so as to reduce the synchronization overhead.

- Thread overhead: The overhead for creating and executing a thread has to be considered. Large numbers of threads can balance imbalanced workloads but may introduce an overhead that worsens performance. This is a tradeoff. The results indicate that workloads must be as evenly distributed as possible, and that a minimum number of threads should be used.

- Communication cost: In UMA systems, the effect of network overhead has never been considered. But in NUMA systems, the network traffic exerts a very important influence on the scheduling of decisions. The right decision will save message-passing time and network communication. This attribute has the same effect that data locality has, in that the LDS method is a good example of this kind of problem may be handle. The network communication overhead can be roughly divided into three levels:light traffic, normal traffic and heavy traffic.

- Synchronization overhead: The synchronization primitives provided by a system are related to the synchronization overhead introduced by scheduling algorithms. If a system provides few synchronization primitives, the synchronization overhead will be high. Thus, we divide the synchronization overhead into four levels. Level one is no overhead, level two is slight overhead, level three is moderate overhead, and level four is high overhead. SS works well when there is no synchronization overhead since it may introduce many overheads when shared variables are accessed. GSS does not work well when the overhead is moderate or high.

- Easie of: If two scheduling methods are suitable for a particular loop, the one that is easier to implement should be chosen. Therefore, ease of implementation should also be considered in our expert system approach.

- Factor: In some scheduling strategies, the values of factors, such as allocation factor in SSS [5], β and γ in AHS, the chunk size in CSS, etc, deterministically affect the execution time of the program. Therefore, they should not be neglected if we want to get optimal factors in order to achieve reasonable performance.

- Prefetch: To maximize memory performance, prefetching of multiple single-word blocks on a miss reduces the miss ratio by approximately 5% to 30% compared to a system with no prefetching. The adaptive prefetching strategy tends to further reduce the miss ratio and the network traffic. In addition, the average memory delay in a multiprocessor system using this adaptive prefetching will be reduced. The relation between a loop-scheduling algorithm and the method used to prefetch the data is not clear and will not be discussed in this paper. The memory performance appears to be relatively insensitive to whether the specific loop-scheduling strategy is GSS, or to whether a single chunk of iterations is assigned to each processor at the start of each parallel loop.

Tables 4 and 5 show the relationships between seventeen attributes and parallel loop scheduling algorithms in UMA and NUMA models, respectively. The features mentioned above are the attributes based upon which we constructed our attribute grid. 'Machine model' has two categories: UMA and NUMA. 'Memory access ratio' means the speed ratio of the 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 loops: 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 a nested loop is profitable to algorithms. 'Loop carried dependence' is classified as DOALL and DOACROSS. 'Ease of implementation' describes if implementation of the algorithm is easy. 'Facto' 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, and heavy. 'Start time' determines whether all each processor starting time need to be equal or not.

3.2 The Anatomy of the IPLS System

In this paper, we propose a new system, called intelligent parallel loop scheduling (IPLS) (Fig. 5), which uses knowledge-based techniques to select an appropriate loop-scheduling algorithm. The approach makes good use of the advantages of algorithms to improve loop parallelism. By the resulting algorithms for assigning parallel loop on multiprocessor systems, it is believed that the applications can save execution time and achieve a high level of speedup.

Table 4. 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	R,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)
Ease of implementation	X	X	X	No	X	No	No	No	No
Factor	—	—	k	—	Ns,NF	x=2	b,g	a,k	K
Thread Overhead	l,n	h	l,n	h	n	n	n,h	n,h	l,n
Comm. Overhead	X	1	1	1	l,n	1	l,n	l,n	1
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 5. The attributive table for NUMA models.

	NUMA Model							
	DOALL							
	AFS	MAFS	CAFS	LAFS	DAFS	LDS	GDCS[4]	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
Ease of implementation	No	No	No	No	No	No	No	No
Factor	$0.5<\alpha<1$	—	k	K	—	β	α,β	$A=\frac{N}{p^2}$
Thread Overhead	l, n	1	l,n	l,n	1	l,n	l,n	1
Comm. Overhead	1	1	1	1	1	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

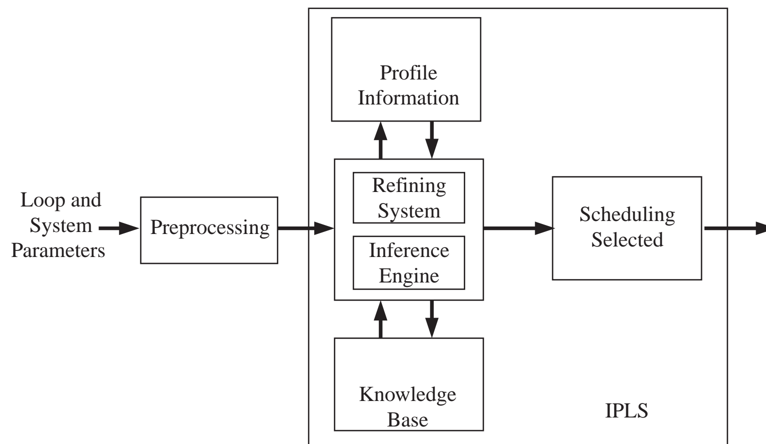


Fig. 5. The system architecture of IPLS.

A 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 separate from the inference component. This makes it convenient to append new knowledge or update existing knowledge easily. The rule-based approach is one of the forms commonly used in many knowledge-based systems. The primary difficulty in building a knowledge base is in acquiring the desired knowledge. To ease acquisition of knowledge, one popular technique 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 says that the symptoms of cold are headache, coughing and sneezing, he may have those symptons. However, in RGA, a person is not considered to catch a cold unless that he has all of the symptoms. 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. The runtime environment contains five components, which are briefly described as follows:

- Knowledge Base: This component contains knowledge required to solve the problem of determining an appropriate parallel loop-scheduling algorithm to be applied. The knowledge is constructed as a rule base. This type of system uses knowledge encoded in the form of production rules, i.e., If ... Then ... rules.

- 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 the machine architecture and programmers to draw a conclusion, an appropriate parallel loop-scheduling algorithm. The inference engine attempts to find connections between the input attributes explained in section three and the selected loop-scheduling algorithm according to RGA and AOT. An example of applying RGA/AOT is shown in Table 6. ‘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 to apply the scheduling algorithm. For those entries that are not labeled 'X' or 'D', integer numbers are used to represent the relative degree of importance for an attribute that does not dominate the object but is of some degree of importance relative to other attributes. A larger integer number implies that the attribute is 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
We illustrate the inference of IPLS with an example. Input data: hungry
Rule 1: If (thirsty) Then (drink water)
Rule 2: If (hungry) Then (eat sandwich)
Inference: Rule 2 is matched because of the input data, so a result is obtained, that is, "eat a sandwich".

Table 6. 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

- Scheduling Algorithm Library: In the library, there are twelve representative scheduling algorithms that are classified as UMA or NUMA models. Whenever any new typical scheduling strategy is developed, the rule can be modified easily, and the new strategy is added into the library.

- Profile Information: After the program applying the selected loop scheduling algorithm is executed, some information about the number of iterations, the maximal time of iterations, the minimal time of iterations, the total time used by the program, the number of synchronizations, the 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 in order to modify the attributes by means of the refining system.

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

3.3 Refining System

In many parallel loop-scheduling algorithms, there are some attributes, such as factors, which influence the performance of an executed program. For example, the adaptive hybrid scheduling algorithm has two factors, β and γ, determining the fetching processor whether or

not to fetch more iterations form the work queue in the dynamic level after executing the iterations from the static level. These two factors, β and γ, should be adjusted by the programmers according to the properties of parallel computers. However, appropriately selecting the values of β and γ on different systems is difficult. If we can refine the values of the factors in the loop-scheduling algorithm by using the profile information derived from the record of the execution process of the program, it is obvious that the new factors cyclically modified by the refining procedure will make the parallelism of program more clear and improve the performance. This method stated also has feedback-learning ability and is intelligent. In this paper, a refining system based upon profile information consisting of the following seven items will be included in our model.

- the number of iteration;
- the maximal time of iterations;
- the minimal time of iterations;
- the total time of program;
- the number of synchronizations;
- the number of remote memory accesses;
- the workload distribution of each processor.

Refining attributes without modifying rules in the knowledge base is a problem, but it is solved in our refining system by storing attribute data into a file called Attri_file and using the data type of the structure (record) as a 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 the knowledge base do not need to be changed, and the inference engine does not need to be recompiled.

There are several situations in which the refining system is suggested to be used. Firstly, when IPLS is constructed completely, some attributes in the knowledge base maybe crude, which may prevent an appropriate loop scheduling from being selected. Secondly, when IPLS is ported to a new system environment, some attributes of the computer system, such as the memory access rate, need to be changed to reflect the selection of scheduling method. In addition, the non-optimal values of features in the knowledge base may cause an appropriate loop-scheduling algorithm to consume much execution time. Therefore, these features shall be refined to reduce the execution time of the program if the executable code is executed repeatedly. The programmer can determine whether or not to use the refining system before deriving an ideal loop-scheduling algorithm for the program. When the refining system is used, the programmer can also decide how many loop-scheduling algorithms are to be selected by the inference engine. The flow chart of the refining system is shown in Fig. 6.

3.4 The Algorithm of IPLS and an Example

In this section, we will describe our algorithm for the intelligent parallel loop scheduling system (IPLS). The algorithm consists of four phases. The following attributes will be obtained from the input file and the parameters of the computer system.

1. How many processors are in the system?

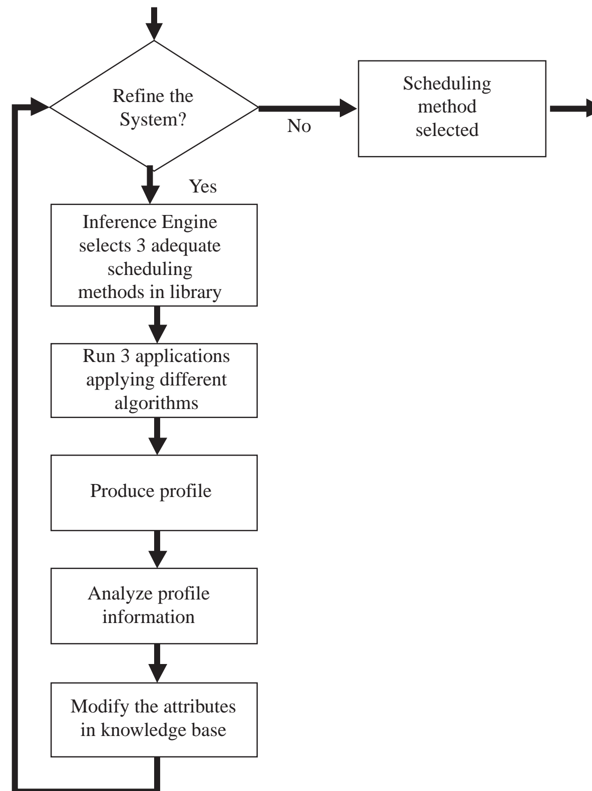


Fig. 6. Flow chart of the refining system.

2. Which machine model of the system architecture? (UMA/NUMA)
3. What is the memory access rate (the speed ratio of the cache, memory and network)?
4. What kind of loop type is present? (1-11; 11 types)
5. What is the level of program size?
6. What kind of loop style is used? (1-4; 4 style methods)
7. Is there strongly affinity? (yes/no)
8. Is the loop bound known during compiler time? (yes/no)
9. Is the loop nested? (yes/no)
10. How large loop carries dependence distance?

A certainty factor (CF) value for each question expresses the importance of that question.

Output: If there is more than one suggestion, the one with the optimal evaluation number among the thread overhead, communication overhead and synchronization overhead will be chosen. There are four phases in the process for constructing the algorithm.

- Phase 1: Get the loop attributes from the input file.
- Phase 2: Call the inference engine to draw a conclusion using the rules; that is, find the most suitable loop scheduling method.

- Phase 3: Apply the Single-to-multiple thread translator (s2m) [20] to adopt the appropriate scheduling so as to partition the loop on multiprocessor systems.
- Phase 4: While the loop is being executed, the profile information is generated and referenced in order to modify the attributes.

For example, the program segment of Adjoint Convolution is shown in Fig. 7. The attributes of the system, the number of processors, machine model, and memory access rate, are detected by the system automatically. Let us describe the four phases used in this example.

```
n = 1000;
for (i = 1; i <= n * n; ++i)
   for (k = i; k <= n * n; ++k)
      A[i-1] + = x * b [k-1] * c [n * n + i - k - 1];
```

Fig. 7. The program segment of Adjoint Convolution.

- Phase 1: The following attributes can be obtained from the input file.
  1. The loop is the 4th type (AC).
  2. The program size is large.
  3. The loop has decreasing workload, which means it is the 3rd style.
  4. There is slight affinity in this example, that means no.
  5. The loop bound is known during compiler time.
  6. The loop is nested.
  7. Loop carried dependence distance is zero, that means DOALL.
- Phase 2: Post to inference, IPLS finds that the constraints of TSS are satisfied and determines that TSS is the most suitable algorithm.
- Phase 3: The TSS is invoked to transform the loop into a multithread program as shown as Fig. 8. The TSS algorithm partitions and schedules the iterations of Adjoint Convolution.
- Phase 4: While the loop is being executed, the profile information is generated and referenced so as to modify the attributes in the knowledge base.

```
Void FORALL1 (loop)
Struct loop_args * loop;
{
int i, k, n;
n = 1000;
for (i = loop->begin; i <= loop->end; ++i)
   for (k = i; k <= n * n; ++k)
      A[i-1] += x * b [k-1] * c[n*n+i-k-1];
}
```

Fig. 8. The multithreaded program segment of Adjoint Convolution.

4. EXPERIMENTAL RESULTS

In this section, two parts of an experiment will be examined for IPLS. The first part of the experiment is based on a UMA system. The other is simulation of a NUMA system by using a UMA system.

4.1 Experimental Environment

Our target machine was a two Pentium-133 CPU multiprocessor system, running the Windows NT multithreaded OS that supports Win32 API functions. The system included a 512K external cache, 64MB shared-memory, and a 64-bit high-speed frame bus. Our programming tool was visual C++, which provides Win32 API function call. Due to the symmetric architecture, computation tasks could be easily distributed to any available processor.

4.2 Applications

Ten popular application programs were chosen and executed in UMA and NUMA environments. These applications were discussed in section 3 and represent different loop types. We tested these applications as follows to show the feasibility and generalization of IPLS.

1. Adjoint Convolution: The loop in this application has a decreasing workload; only the outer loop can be parallelized and the ith iteration takes $O(N^2 - i)$ time. It has no data affinity to exploit.
2. Reverse Adjoint Convolution: This program has an increasing workload. As the loop bound i increases from 1 to N^2, the workload also increases from O(1) to $O(N^2)$.
3. Gaussian Elimination: This program has a small load imbalance across iterations and has some data affinity between matrix references. The inner loop can be parallelized.
4. Matrix Multiplication: This, the most commonly used program in parallel processing, has a uniform workload. Since Matrix A rows and Matrix B columns are referenced constantly, and the elements of both Matrices are not modified, and a local cache, if available, will be very useful to the system.
5. All Pairs Shortest Paths: The workload of the ith iteration of the inner parallel loop depends on A[i][k], and it takes O(1) or $O(N)$ times to complete the work. Because each processor's work queue initially contains about N/P consecutive iterations, the total loads for all the processors are about the same. Load imbalance is not significant. The ith iteration always accesses the ith row of the matrix. Therefore, the application has to exploit the affinity effect.
6. Successive Over-Relaxation (SOR): All the iterations of the SOR parallel loop take about the same time to execute, and each iteration always accesses the same set of data. Exploiting processor affinity may improve performance better than balancing the workload can. In this application, each parallel iteration has a locality rate of one and a data set of N array elements. The computational granularity of each parallel iteration is $O(N)$.
7. Jacobi Iteration: In the JI program, the top 20% of the rows of elements in the non-singular matrix A are nonzero elements, which are generated by a random number generator. The iterations of the parallel loop have a different workload, which is determined by the distribution of nonzero elements in A, so exploiting load imbalance will improve the performance. However, the workload of each parallel iteration is not changed when it is executed repeatedly.

8. LU Decomposition: This consists of an outer sequential loop and a parallel loop. In the innermost loop, one of the rows of the matrix A is modified based on the pivot row *k*.
9. Gauss-Jordan Elimination: The iteration granularity of Gauss-Jordan Elimination is small and is independent of the program size. The amount of variance in the iteration length is small, too. Programs of this kind are more suitable for static scheduling schemes than self-scheduling schemes. To outperform the static scheduling scheme problems, self-scheduling schemes must be able to achieve load balance with very small scheduling overhead.
10. Transitive Closure: The characteristic of this program is that the workload depends on the input data. Each iteration takes either O(1) or O(*N*) time. Since the input data affects the amount of execution time, the workload is randomly.

4.3 Experimental Results on a UMA System

Because the restriction of the operating system to programmers, a system function call for binding any thread onto a specific processor is not available and only dynamic scheduling is allowed to use on this experiment. There were two kinds of experiments on a UMA system and a NUMA system. One studied the execution time and speedup of above ten applications, and the other examined a combined case that included ten applications in a program.

4.3.1 The implementation on the UMA system

Let us examine the implementation on the UMA system, which was a 2-processor machine; the execution time and the corresponding speedup are shown in Table 7 and Fig. 9, respectively.

Table 7. The execution time (ms)/speedup of 11 applications obtained by 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
Saor	109062/1	76891/1.418	82594/1.32	83943/1.299	86742/1.257	38126/1.41	77680/1.404	as CSS/2
Spath	63063/1	57032/1.106	58867/1.071	43146/1.462	310469/1.543	295922/1.619	296078/1.618	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

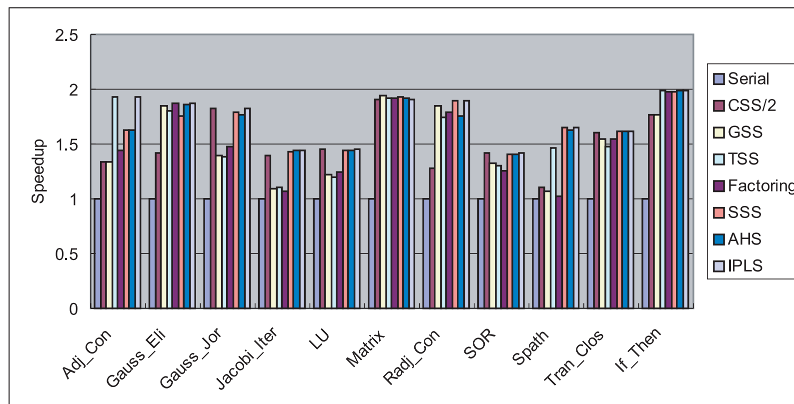


Fig. 9. The speedup of 11 applications by applying different loop scheduling.

GSS performed poorly for Adjoint Convolution because the workload of the iterations is decreasing and TSS is the most efficient algorithm for Adjoint Convolution (shown in Fig. 5.1(a)). CSS/2 was suitable for the applications like Gauss Jordan Elimination with a random unbalanced workload, LU Decomposition with a decreasing imbalanced workload, and SOR with a uniform balanced workload (shown in Fig. 5.1(c), (e), (h)) respectively. The factoring scheduling algorithm wass suitable for Gauss Elimination with a random balanced workload (shown in Fig. 5.1 (b)). SSS was suitable for applications like Reverse Adjoint Convolution with an increasing imbalanced workload, All Pairs Shortest Paths with a random balanced workload, and Transitive Closure with a random unbalanced workload (shown in Fig. 5.1(g), (i), (j)), respectively. AHS was suitable for Jacobi Iteration with a random unbalanced workload (shown in Fig. 5.1(d)). We can find that none of six scheduling algorithms under UMA system was suitable for all applications. IPLS could choose an appropriate scheduling algorithm and get good performance for most of the applications except Matrix Multiplication and If_Then. In the case of Matrix Multiplication, IPLS did not apply the optimal approach, GSS, but chose CSS/2 because the workload of iterations in this program is uniform. However, the number of processors, 2, was so small that CSS could not exploit the ability fully. In the case of the If_Then application, IPLS did not apply the optimal approach, TSS, but chose AHS because AHS is suitable for a random workload of iterations in the program. The reason for not selecting an optimal approach was similar to that for the case of Matrix Multiplication. Although the selection of scheduling algorithms was not absolutely accurate, we could solve the problem by refining the attributes causing the error. The refining system in IPLS will be used afterwards. Traditionally, once a scheduling algorithm is used, it will be used through out the entire program. But IPLS can always choose an appropriate scheduling algorithm according to the behaviors of the loops among one program. A comparison of results obtained using KPLS and IPLS on a UMA system is shown in Table 8.

In the second experiment, IPLS chose different scheduling algorithms for each loop in the combined program integrated from the above eleven applications. For example, according to the loop behaviors, IPLS selected TSS for the Adjoint Convolution part of the combined program and chose factoring for the Gauss Elimination part, instead of choosing only one scheduling method. Table 9 and Fig. 10 show, respectively, the experimental execution time and the corresponding speedup for the combined program.

Table 8. A comparisons of results obtained by applying KPLS and IPLS.

Applications	Best Loop Scheduling	KPLS Result	IPLS Result
Adj_Con	TSS	as TSS	as TSS
Gauss_Eli	Factoring	as Factoring	as Factoring
Gauss_Jor	CSS/2	as CSS/2	as CSS/2
Jacobi_Iter	AHS	as TSS	as AHS
LU	CSS/2	as CSS/2	as CSS/2
Matrix_Mul	GSS	as CSS/2	as CSS/2
Radj_Con	SSS	as Factoring	as SSS
SOR	CSS/2	as CSS/2	as CSS/2
Spath	SSS	as TSS	as SSS
Tran_Clos	SSS	as TSS	as SSS
If_Then	AHS	as TSS	as AHS

Table 9. 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

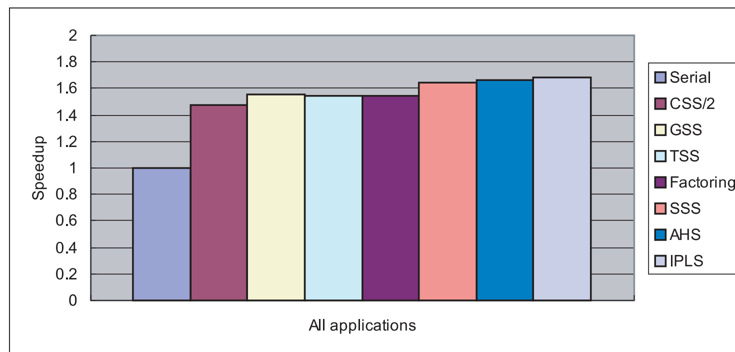


Fig. 10. The speedup of the combined program for different kinds of loop scheduling.

4.3.2 The simulation results on the UMA system

To overcome the problem of the limit of only 2 processors in the UMA system in our experiment, we simulated multiprocessor systems with 4, 8, 16 and 32 processors on our target machine with 2 processors, and the results are shown in Table 10.

From the simulation on the UMA system, some significant results were obtained: On the UMA system, generally speaking, the greater the number of processors, the better the performance of the applications with DOALL loops while applying each scheduling algorithm, and the number of processors also influenced the selection of loop scheduling algorithms.

Table 10. The execution time (ms) of 11 applications for different scheduling algorithms on the UMA system with 4, 8, 16 and 32 processors.

Adjoint Convolution								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	20104	8637	8684	4944	4942	6773	6921	as TSS
8	20104	4628	4662	2493	2471	3551	3637	as TSS
16	20104	2393	2392	1256	1237	1809	1844	as TSS
32	20104	1218	1216	652	621	916	939	as TSS
Gauss Elimination								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	365359	134685	95052	907644	95910	103430	103539	as GSS
8	365359	68619	50178	55819	51179	54171	54089	as GSS
16	365359	35075	27418	30796	28368	28966	28974	as GSS
32	365359	18771	16853	17521	17483	16843	16859	as GSS
Gauss-Jordan Elimination								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	7765	2153	2692	2453	2456	2618	2618	as CSS
8	7765	1207	1702	1460	1462	1615	1590	as CSS
16	7765	732	1159	941	956	1076	1061	as CSS
32	7765	501	862	705	716	777	775	as CSS
Jacobi Iteration								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	14047	3599	4216	4293	4518	5094	4246	as CSS
8	14047	1996	2468	2352	2692	2895	2490	as CSS
16	14047	1068	1581	1438	1676	1773	1650	as CSS
32	14047	631	1026	1058	1123	1060	1065	as CSS
LU Decomposition								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	40995	10475	11501	11717	11978	11227	11296	as CSS
8	40995	5771	6470	6148	6748	6355	6344	as CSS
16	40995	3357	3798	4165	3891	3473	3470	as CSS
32	30995	2172	2373	2408	2236	2336	2331	as CSS
Matrix Multiplication								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	23453	5909	5955	5846	5943	5946	5841	as CSS
8	23453	2975	3025	3118	2994	3168	3111	as CSS
16	23453	1489	1638	2107	1512	1778	1762	as CSS
32	23453	755	859	782	759	920	903	as CSS
Reverse Adjoint Convolution								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	27235	11934	6937	6832	6820	7110	7134	as GSS
8	27235	6360	3451	3542	3412	3585	3264	as GSS
16	27235	3381	1741	1877	1781	1859	1884	as GSS
32	27235	1695	879	946	899	945	959	as GSS
SOR								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	109062	21805	29204	29572	30108	29124	29166	as CSS
8	109062	14538	5616	15634	16169	15545	15572	as CSS
16	109062	7939	8565	9797	8822	8915	8865	as CSS
32	109062	4221	4693	4668	4713	5261	5297	as CSS
All Pairs Shortest Paths								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	63063	16095	17052	17223	17591	16898	17100	as CSS
8	63063	8292	9539	8759	9835	9507	9365	as CSS
16	63063	4468	5560	5802	5725	5455	5452	as CSS
32	63063	2790	3708	3698	3159	3723	3711	as CSS
Transitive Closure								
Processor number	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
4	479188	126258	122581	126527	124971	123302	122999	as SSS
8	479188	64412	64423	64318	65088	63336	63724	as SSS
16	479188	33470	34509	34121	35588	34331	34903	as SSS
32	479188	18206	19611	22358	19960	18974	19026	as SSS

In the case of Adjoint Convolution, if the number of processors was less than 8, TSS was the appropriate scheduling algorithm; otherwise, the factoring scheduling algorithm is the optimal choice whereas the difference of execution time between TSS and factoring was not distinct. In the case of Matrix Multiplication, if the number of processors was large enough, equal to or greater than 8, CSS was the optimal scheduling algorithm because of the uniform workload of iterations. In the case of Transitive Closure, when the number of processors was 4, the chunk size of the static phase in SSS was reasonable, so the execution time of the program while applying SSS was shorter than that of the other algorithms. When the number of processors was more than 4, CSS produced the optimal results. In addition, no matter how many processors were used, there was no distinct difference in the performance of Transitive Closure when different scheduling algorithms where used. For other applications like Gauss Elimination, Gauss-Jordan Elimination, Jacobi Iteration, LU Decomposition, SOR and All Pairs Shortest Paths, the influence of the number of processors on the selection of an appropriate scheduling algorithm was not obvious. When the number of processors was equal to or greater than 32, the execution times of Gauss Elimination, Matrix Multiplication, LU Decomposition, SOR, and Transitive Closure were almost the same. This shows that the selection of scheduling algorithms for the six applications has less influence on the performance if the number of processors is large enough on a UMA system. Although the selection of scheduling algorithms was not completely accurate, we could solve this problem by refining the attributes causing the error. The refining system in IPLS will be used afterwards.

4.4 The Simulation Result on the NUMA System

The memory access cost between a UMA system and a NUMA system is different because a NUMA system will consume more on memory access if the data is not in local memory. In a NUMA system, the hierarchy of memory can be divided into three levels: cache, local memory and remote memory. According to the experimental results, the memory access rate of the three levels was about 1:10:200. Each time, synchronization took 5 microseconds (ms), remote communication took 30 ms, and the thread management overhead waas 1.006 ms. In addition, since the transfer speed of the bus is much faster than network, the cost of communication and synchronization could be neglected. The simulations for clustering NUMA system with 16 and 32 processors were conducted on the UMA system. In order to understand the influence of architecture of the clustering NUMA model on the performance of program execution, there were two different numbers of processors in a cluster as shown in Table 11.

In this simulation, Gauss Elimination, Jacobi Iteration and SOR were selected because they have strong data locality and different loop styles. Seven kinds of loop-scheduling algorithms, including AFS [7], CAFS [16], LAFS [17], MAFS [13], DAFS [10], adaptive scheduling [18] and IPLS, were considered. Table 12 shows the execution times of three applications when different scheduling algorithms were applied.

In the simulations of Gauss Elimination, Jacobi Iteration and SOR, no matter what the structure of the clustering NUMA system was, AFS was not selected because it would result in many remote memory accesses and high synchronization cost. As shown in Table 12, no matter how many processors were in a cluster, CAFS was suitable for SOR and Jacobi Iteration. This was because if the workload of iterations was uniform, an idle processor only

Table 11. The number of processors in a cluster.

The number of processors on NUMA	16		32	
The number of processors in a cluster	2	4	4	8

Table 12. The executiom times (ms) for 3 applications on the NUMA model.

Gauss Elimination								
Configuration	Serial	AFS	CAFS	LAFS	MAFS	DAFS	Adaptive	IPLS
16(2)	367449	25668	33305	23226	23166	23194	21843	as adaptive
16(4)	367449	25468	33152	23100	23166	22742	21654	as adaptive
32(4)	362095	17151	16929	11441	11620	11547	13487	as LAFS
32(8)	362095	16951	15524	11404	11590	11324	13239	as DAFS
Jacobi Iteration								
Configuration	Serial	AFS	CAFS	LAFS	MAFS	DAFS	Adaptive	IPLS
16(2)	13436	1089	880	930	1047	1146	1211	as CAFS
16(4)	13436	1049	873	887	1027	1097	1178	as CAFS
32(4)	13378	479	431	485	511	604	635	as CAFS
32(8)	13378	439	430	438	506	572	603	as CAFS
SOR								
Configuration	Serial	AFS	CAFS	LAFS	MAFS	DAFS	Adaptive	IPLS
16(2)	108383	7712	6875	6980	7274	7085	7937	as CAFS
16(4)	108383	7562	6871	6859	7214	6932	7805	as LAFS
32(4)	108311	4588	3437	3491	4358	4435	4638	as CAFS
32(8)	108311	4378	3435	3452	4265	4356	4564	as CAFS

searched the most loaded processor in its local cluster instead of migrating iterations by means of remote memory accesses. When the number of processors in the whole system or in a cluster was larger, the execution time was shorter.

In the execution phase of LAFS, when imbalance occurred, if the queues of processors in a local cluster were all empty, an idle processor migrated iterations from the remote clusters. Therefore, LAFS could balance the uniform and non-uniform workload of iterations efficiently. When there were 4 processors in a cluster on a 16-processor system, LAFS was suitable for Gauss Elimination. Table 12 shows that when there were 4 processors in a cluster on a 32-processor system, LAFS was also suitable for SOR application. The more processors in a cluster, the more efficient was program execution because LAFS reduced the remote memory access times. DAFS could collect the information needed by the re-initialization phase about balancing the workload of each processor, but the large scale about processors will result in

more overheads. When there were 8 processors in a cluster on 32-processor system, DAFS was an appropriate scheduling algorithm for Gauss Elimination. Adaptive scheduling exploited the potential for using the dynamic execution history to adaptively adjust the chunk size so as to reduce synchronization and loop allocation overheads and to maintain a better load balance. Therefore, adaptive scheduling was not suitable for the multiprocessor system with a larger number of processors, especially more than 16, because of excessive overhead. As shown in Table 12, when the number of processors was 16, adaptive scheduling was suitable for Gauss Elimination. However, the architecture of clustering NUMA systems and the number of processors influenced the selection of a loop scheduling strategy for a loop. This was taken into consideration as shown in Table 5, so IPLS could correctly choose the appropriate scheduling algorithms for the three applications on clustering NUMA systems with different structures.

4.5 An Example Showing How to Refine Attributes

We found that IPLS could not choose an appropriate scheduling for the case of Matrix Multiplication on the UMA system with two processors as shown in Table 13.

To improve the poor selection ability, the refining system in IPLS was used to modify the attributes in the knowledge base. During the first iteration, first of all, the refining system selected three algorithms, CSS/2, TSS and factoring, in the scheduling library according to the attributes of Matrix Multiplication, and transformed the application into three multithreaded programs by applying three chosen algorithms. Secondly, the multithreaded programs were executed, and the profile information of each program was recorded. Then, the refining system analyzed the profile information to refine the attributes in the knowledge base. Finally, we found that factoring was better than the two other algorithms. Table 14 shows the results of using the refining system in three rounds. It is found that IPLS chose the most appropriate algorithm, GSS, after two rounds, and that the results after the third round had converged into stability. This shows the feasibility of using therefining system.

To summarize the above experimental and simulation results, IPLS can choose a suitable loop-scheduling algorithm for each kind of loop. Once IPLS gets enough information from the system environment and the loop, it can make a correct decision for that loop. Since none of loop-scheduling algorithms can fit all applications, our approach will be a good choice for a parallel compiler.

Table 13. The execution time (ms) of Matrix Multiplication when different scheduling algorithms were applied.

Application	Serial	CSS	GSS	TSS	Factoring	SSS	AHS	IPLS
Matrix_Mul	23453	12281	12095	12229	12214	12187	12203	as CSS/2

Table 14. The results of refining the IPLS in three rounds.

	1st round	2nd round	3rd round
Three algorithms selected by refining system	CSS/2	GSS	SSS
	Factoring	Factoring	GSS
	TSS	SSS	AHS
Result	Factoring	GSS	GSS

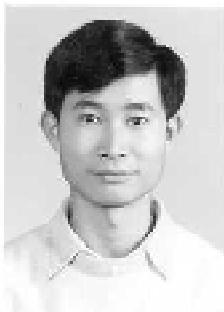
5. CONCLUSION AND FUTURE WORK

In the paper, we have proposed a new approach that uses knowledge-based techniques to select some appropriate loop-scheduling algorithms according to loop behaviors and system states. A rule-based system, called IPLS, has been developed that uses RGA and AOT to integrate existing loop scheduling algorithms for UMA and NUMA systems, and that makes good use of their advantages for loop parallelism. Based on the results for algorithms used to assign parallel loops on multiprocessor systems, it is believed that the program can save execution time and achieve a high level of speedup. In addition, the refined system of IPLS can automatically adjust the attributes in the knowledge base according to profile information so as to match the characteristics of the system environment, especially for NUMA systems. Therefore, IPLS has the feedback-learning ability. Whereas, the adjustment of attributes does not need to modify the rules and recompile the inferring engine. Experiments on a NUMA system could not be implemented due to the operating system constraints and the lack of hardware. In the near future, IPLS will be improved so that it can automatically detect the attributes of system states and loop information instead of requiring the programmer's input. Other directions of research are the implementation of prefetching, and exploration of the distinct relationship between the cache size and the selection of a scheduling method. In addition, finding a way to construct a precise algorithm for adjusting the factors of loop scheduling strategies so as to avoid overshoot is one of our future goals.

REFERENCES

1. B. Hamidzadeh and D. J. Lilja, "Self-adjusting scheduling: An on-line optimization techniques for locality managemant and load balancing," in *Proceedings of the 1994 International Conference on Parallel Processing*, Vol. II, pp. 39-46, CRC Press.
2. S. F. Hummel, E. Schonberg and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Communication of ACM*, Vol. 35, No. 8, 1992, pp. 90-101.
3. C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, Vol. 11, No. 10, 1985, pp. 1001-1016.
4. T. Y. Lee, C. S. Raghavendra and H. Sivaraman, "A practical scheduling scheme for non-uniform parallel loops on distributed memory parallel machines," in *Proceedings of Conference on System Sciences* Vol. 1, 1996, pp. 243-250.
5. J. Liu, V. A. Saletore and T. G. Lewis, "Safe self-scheduling: A parallel loop scheduling scheme for shared-memory multiprocessors," *International Journal of Parallel Programming*, Vol. 22, No. 6, 1994, pp. 589-616.
6. H. Li, S. Tandri, M. Stumm and K. C. Sevcik, "Locality and loop scheduling on NUMA multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, Vol. II, 1993, pp. 140-147.
7. E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel Distributed Systems*, Vol. 5, No. 4, 1994, pp. 379-400.
8. C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers," *IEEE Transactions on Computer*, Vol.

36, No. 12, 1987, pp. 1425-1439.
9. O. Plata and F. Rivera, "Combining static and dynamic scheduling on distributed-memory multiprocessors," in *Proceedings of the 1994 ACM International Conference on Supercomputing*, 1994, pp. 186-195.
10. S. Subramaniam and D. L. Eager, "Affinity scheduling of unbalanced workload," in *Proceedings of the 1994 Supercomputing, SC '94* , 1994, pp. 214-226.
11. T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel Distributed Systems*, Vol. 4, No. 1, 1993, pp. 87-98.
12. P. Tang and P. C. Yew, "Processor self-scheduling for multiple-nested parallel loops, " in *Proceedings of the 1986 International Conference on Parallel Processing*, 1986, pp. 528-535.
13. Y. M. Wang and R. C. Chang, "A minimal synchronization overhead affinity scheduling algorithm for shared-memory multiprocessors," *International Journal of High Speed Computing*, Vol. 7, No. 2, 1995, pp. 231-249.
14. M. Wolfe, *High-Performance Compilers for Parallel Computing*, Addison-Wesley Publishing, New York, 1995, pp. 137-162.
15. C. M. Wang and S. D. Wang, "A hybrid scheme for efficiently executing nested loops on multiprocessors," *Parallel Computing*, Vol. 18, No. 6, 1992, pp. 625-637.
16. Y. M. Wang, H. H. Wang and R. C. Chang, "Clustered affinity scheduling on NUMA shared-memory multiprocessors," in *Proceedings of 1st Workshop High Performance Multiprocessor Systems, HPMS '95*, 1995, pp. 168-175.
17. Y. M. Wang, H. H. Wang and R. C. Chang, "Loop scheduling on clustered NUMA multiprocessors," in *Proceedings of 1995 National Computer Symposium, NCS '95*, 1995, pp. 536-543.
18. Y. Yan, C. Jin and X. Zhang, "Adaptively scheduling parallel loops in distributed shared-memory systems," *IEEE Transactions on Parallel Distributed Systems*, Vol. 8, No. 1,1997, pp. 71-80.
19. 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*, Vol. 23, No. 3, 1997, pp. 291-309.
20. C. T. Yang, S. S. Tseng and M. C. Hsiao, "A model of parallelizing compiler on multithreading operating systems," *International Journal of Modelling and Simulation*, Vol. 18, No. 1, 1998, pp. 9-15.
21. H. P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing and ACM Press, New York, 1990.



Yun-Woei Fann (范雲偉) received the MS degree in Computer and Information Science from National Chiao-Tung University, Taiwan, R.O.C., in 1998, and is currently a teacher at an elementary school in Hsinchu. His research interests include expert system, parallel processing, and network.

Chao-Tung Yang (楊朝棟) received a BS degree in Information Science from Tung Hai University in 1990 and an MS degree in Computer and Information Science from National Chiao Tung University in 1992. Since August 1992, he had been a Ph.D. student with the Laboratory of Knowledge Engineering and Programming Languages in Computer and Information Science at National Chiao Tung University, and involved in applying knowledge-based techniques to development a parallelizing compiler. He had passed the first class of the National Higher Examination in Information Processing field in 1994. Then, he had finished Ph.D. program in Computer and Information Science in July 1996. He was the winner of 1996 Acer Dragon Award for outstanding Ph.D. Dissertation. Now, he is working on ROCSAT Ground System Section (RGS) at National Space Program Office (NSPO). His current research interests were in parallelizing compiler design, parallel and distributed computing, and satellite communications.

Shian-Shyong Tseng (曾憲雄) received his Ph.D. degree in Computer Engineering from the National Chiao Tung University in 1984. Since August 1983, he has been on the faculty of the Department of Computer and Information Science at National Chiao Tung University, and is currently a Professor there. From 1988 to 1991, he was the Director of the Computer Center at National Chiao Tung University. From 1991 to 1992 and 1996 to 1998, he acted as the Chairman of Department of Computer and Information Science. From 1992 to 1996, he was the Director of the Computer Center at Ministry of Education and the Chairman of Taiwan Academic Network (TANet) management committee. His current research interests include parallel processing, expert systems, computer algorithm, and internet-based applications.

Chang-Jiun Tsai (蔡昌均) received the MS degree in Computer and Information Science from National Chiao-Tung University, Taiwan, R.O.C., in 1997, and is currently a doctoral candidate at National Chiao-Tung University. His research interests include expert system, object-oriented techniques, and computer assisted learning.