Digital Technical Journal
Volume 6, Number 3

VTX Version of the KAP Paper

The KAP Parallelizer for DEC Fortran and DEC C Programs

by

Robert H. Kuhn, Bruce Leasure, and Sanjiv M. Shah

ABSTRACT

The KAP preprocessor optimizes DEC Fortran and DEC C programs to achieve their best performance on Digital Alpha systems. One key optimization that KAP performs is the parallelization of programs for Alpha shared memory multiprocessors that use the new capabilities of the DEC OSF/1 version 3.0 operating system with DECthreads. The heart of the optimizer is a sophisticated decision process that selects the best loop to parallelize from the many loops in a program. The preprocessor implements a robust data dependence analysis to determine whether a loop is inherently serial or parallel. In engineering a high-quality optimizer, the designers specified the KAP software architecture as a sequence of modular optimization passes. These passes are designed to restructure the program to resolve many of the apparent serializations that are artifacts of coding in Fortran or C. End users can also annotate their DEC Fortran or DEC C programs with directives or pragmas to guide KAP's decision process. As an alternative to using KAP's automatic parallelization capability, end users can explicitly identify parallelism to KAP using the emerging industry-standard X3H5 directives.

INTRODUCTION

The KAP preprocessor developed by Kuck & Associates, Inc. (KAI) is used on Digital Alpha systems to increase the performance of DEC Fortran and DEC C programs. KAP accomplishes this by restructuring fragments of code that are not efficient for the Alpha architecture. Essentially a superoptimizer, KAP performs optimizations at the source code level that augment those performed by the DEC Fortran or DEC C compilers.[1]

To enhance the performance of DEC Fortran and DEC C programs on Alpha systems, KAI engineers selected two challenging aspects of the Alpha architecture as KAP targets: symmetric multiprocessing (SMP) and cache memory. An additional design goal was to assist the compiler in optimizing source code for the reduced instruction set computer (RISC) instruction processing pipeline and multiple functional units.

This paper discusses how the KAP preprocessor design was adapted to parallelize programs for SMP systems running under the DEC

OSF/1 version 3.0 operating system. This version of the DEC OSF/1 system contains the DECthreads product, Digital's POSIX-compliant multithreading library. The first part of the paper describes the process of mapping parallel programs to DECthreads. The paper then discusses the key techniques used in the KAP design. Finally, the paper presents examples of how KAP performs on actual code and mentions some remaining challenges. Readers with a compiler background may wish to explore Optimizing Supercompilers for Supercomputers for more details on KAP's techniques.[2]

In this paper, the term directive is used interchangeably to mean directive, when referring to DEC Fortran programs, and pragma, when referring to DEC C programs. The term processor generally represents the system component used in parallel processing. In discussions in which it is significant to distinguish the operating system component used for parallel processing, the term thread is used.

THE PARALLELISM MAPPING PROCESS

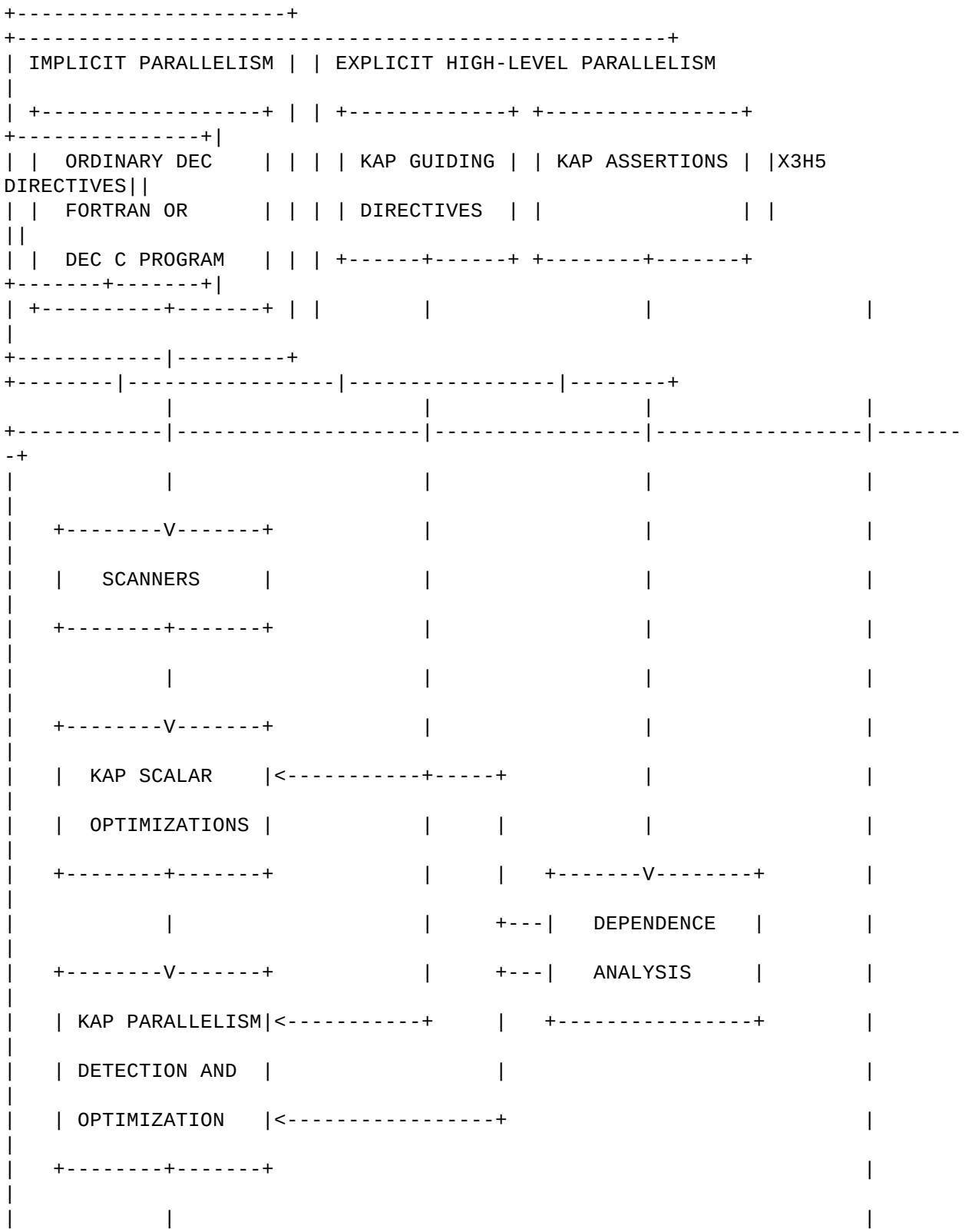
Figure 1 shows the input modes and major phases of the compilation process. Parallelism is represented at three levels in programs using the KAP preprocessor on an Alpha SMP system. The first two are input to the KAP preprocessor; the third is the representation of parallelism that KAP generates. The three levels of parallelism are

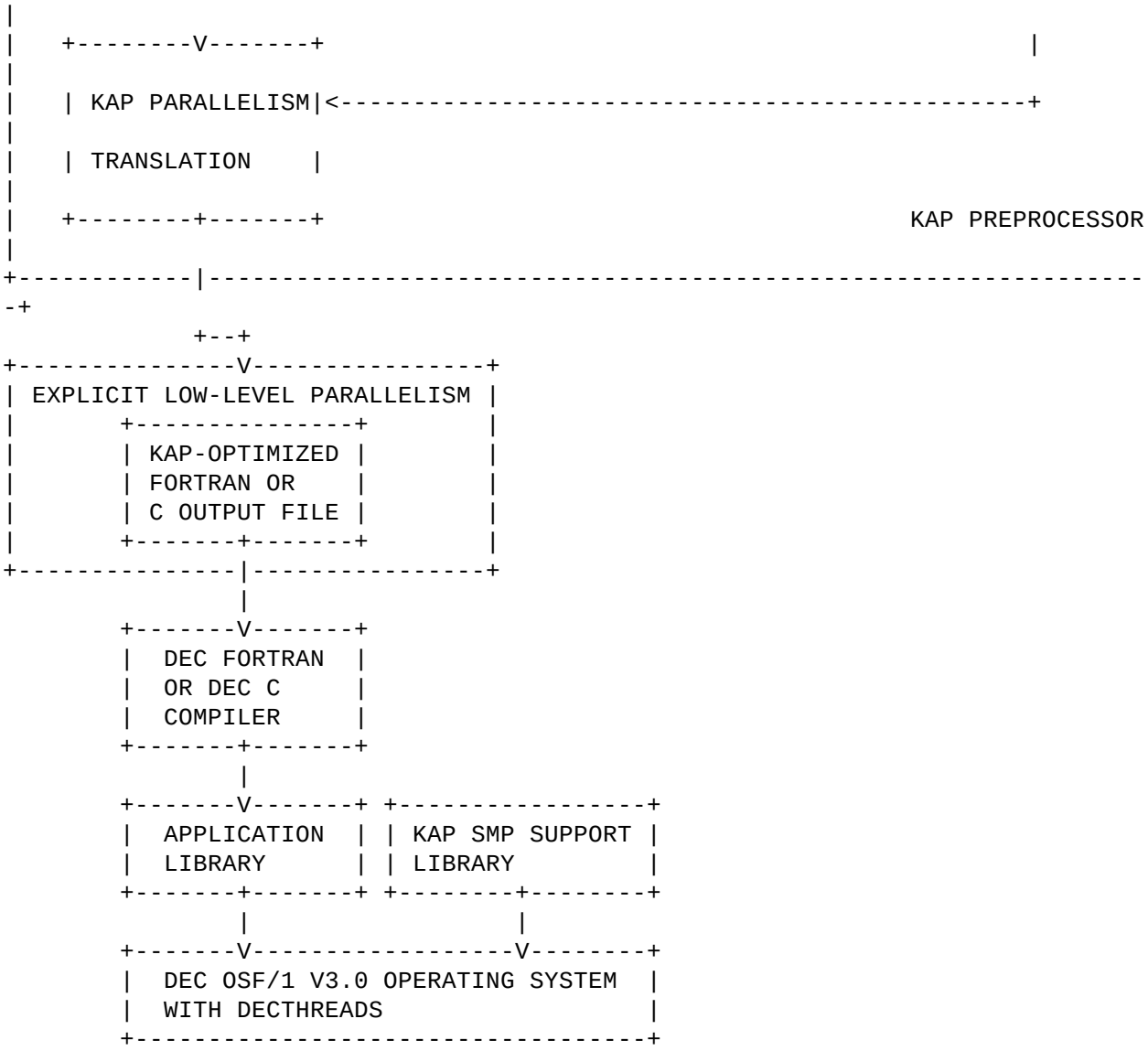
1. Implicit parallelism. Starting from DEC Fortran or DEC C programs, KAP automatically detects parallelism.

2. Explicit high-level parallelism. As an advanced feature, users can provide any of three forms: KAP guiding directives, KAP assertions, or X3H5 directives. KAP guiding directives give KAP hints on which program constructs to parallelize. KAP assertions are used to convey information about the program that cannot be described in the DEC Fortran or DEC C language. This information can sometimes be used by KAP to optimize the program. Using X3H5 directives, the user can force KAP to parallelize the program in a certain way.[3]

3. Explicit low-level parallelism. KAP translates either of the above forms to DECthreads with the help of an SMP support library. (The user could specify parallelism directly, using DECthreads; however, KAP does not perform any optimization of source code with DECthreads. Therefore, the user should not mix this form of parallelism with the others.)

Figure 1 Parallelism Mapping Process





Because the user can employ parallelism at any of the three levels, a discussion of the trade-offs involved with using each level follows.

From DEC Fortran or DEC C Programs

The KAP preprocessor accepts DEC Fortran and DEC C programs as input. Although starting with such programs requires the compilers to intelligently utilize a high-performance SMP system, there are several reasons why this is a natural point at which to start.

- Lots of software. Since DEC Fortran and DEC C are de facto standards, there exists a large base of applications that can be parallelized relatively easily and inexpensively.

- Ease of use. Given the high rate at which hardware costs are decreasing, every workstation may soon have multiple processors. At that point, it will be critical that programming a multiprocessor be as easy as programming a single processor.

- Portability. Many software developers with access to a multiprocessor already work in a heterogeneous networking environment. Some systems in such an environment do not support explicit forms of parallelism (either X3H5 or DECthreads). The developers would probably like to have one version of their code that runs well on all their systems, whether uniprocessor or multiprocessor, and using DECthreads would cause their uniprocessors to slow down.

- Maintainability. Using an intricate programming model of parallelism such as X3H5 or DECthreads makes it more difficult to maintain the software.

KAP produces KAP-optimized DEC Fortran or DEC C as output. This fact is important for the following reasons:

- Performance. Users can leverage optimizations from both Digital's compilers and KAP.

- Integration. Users can employ all of Digital's performance tools.

- Ease of use. Expert users like to "tweak" the output of KAP to fine-tune the optimizations performed.

With KAP Guiding Directives, KAP Assertions, or X3H5 Directives

Although the automatic detection of parallelism is frequently within the range of KAP capabilities on SMP systems, in some cases, as described below, users may wish to specify the parallelism.

- In the SMP environment, coarse-grained parallelism is sometimes important. The higher in the call tree of a program a preprocessor (or compiler, as well) operates, the more difficult it is for a preprocessor to parallelize automatically. Even though the KAP preprocessor performs both inlining and interprocedural analysis, the higher in the call tree KAP operates, the more likely it is that KAP will conservatively assume that the parallelization is invalid.

- Sometimes information that is available only at run time precludes the preprocessor from automatically finding parallelism.

- Occasionally, experts can fine-tune the parallelism to get the highest efficiency for programs that are run frequently.

- For software that is more portable between systems, it is sometimes important to get repeatable parallel performance or to indicate where parallelism has been applied. In such cases, explicit parallelism may be preferable.

Three mechanisms are available to the user for directing KAP to parallelism. The first mechanism uses KAP guiding directives to guide KAP to the preferred way to parallelize the program. The second mechanism uses KAP assertions. The third mechanism uses X3H5-compliant directives to directly describe the parallelism. The first two mechanisms differ significantly from the third. With the first two, KAP analyzes the program for the feasibility of parallelism. With the third, KAP assumes that parallelism is feasible and restricts itself to managing the details of implementing parallelism. In particular, the user does not have to be concerned with either the scoping of variables across processors, i.e., designating which are private and which are shared, or the synchronization of accesses to shared variables.[4] KAP guiding directives will not be discussed in this paper. KAP assertions and how they are implemented are discussed later in the section Advanced Ways to Affect Dependences. A description of the X3H5 directives follows.

The X3H5 model of parallelism is well structured; all operations have a begin operation--end operation format. The parallel region construct identifies the fork and join points for parallel processing. Parallel loops identify units of work to be distributed to the available processors. The critical section and one processor section constructs are used to synchronize processors where necessary. Table 1 shows the X3H5 directives as implemented in KAP.

Table 1 X3H5 Directives As Implemented in KAP

Function	X3H5 Directives
To specify regions of parallel execution	C*KAP* PARALLEL REGION C*KAP* END PARALLEL REGION
To specify parallel loops	C*KAP* PARALLEL DO C*KAP* END PARALLEL DO
To specify synchronized sections of code such that all processors synchronize	C*KAP* BARRIER
To specify that all processors execute sequentially	C*KAP* CRITICAL SECTION C*KAP* END CRITICAL SECTION
To specify that only the first processor executes	C*KAP* ONE PROCESSOR SECTION C*KAP* END ONE PROCESSOR SECTION

To the DEC OSF/1 Operating System with DECthreads

Although KAP does not optimize programs that use DECthreads directly, there may be some benefits to specifying parallelism explicitly using DECthreads.

- DECthreads allows a user to construct almost any model of parallel processing fairly efficiently. The high-level approaches described above are limited to loop-structured parallel processing. Some applications obtain more parallelism by using an unstructured model. It can even be argued that for some cases, unstructured parallelism is easier to understand and maintain.

- A user who invests the time to analyze exactly where parallelism exists in a program may wish to forego the benefits mentioned above and to capture the parallelism in detail with DECthreads. In that manner, no efficiency is lost because the preprocessor misses an optimization.

- The POSIX threads standard to which DECthreads conforms is available on several platforms. Because this standard is broadly adopted and language independent, it is only slightly less portable than implicit parallelism.

The KAP preprocessor translates a program in which KAP has detected implicit parallelism or a program in which the user explicitly directs parallelism to DECthreads. KAP performs this translation in two steps. First, it translates the internal representation into calls to a parallel SMP support library. Second, the support library makes calls to DECthreads.

The SMP support library implements various aspects of X3H5 notation, as can be seen by comparing Tables 1 and 2.

Table 2 KAP SMP Support Library

C Entry Point Name	Fortran Name	Function	OSF/1 DECthreads Subroutines Used
__kmp_enter_csec	mppecs	To enter a critical section	pthread_mutex_lock
__kmp_exit_csec	mppxcs	To exit a critical section	pthread_mutex_unlock
__kmp_fork	mppfrk	To fork to several threads	pthread_attr_create, pthread_create
__kmp_fork_active	mppfkd	To inquire if already parallel	(none)
__kmp_end	mppend	To join threads	pthread_join, thread_detach
__kmp_enter_onepsec	mppbop	To enter a single processor section	pthread_mutex_lock, pthread_mutex_unlock
__kmp_exit_onepsec	mppeop	To exit a single processor section	pthread_mutex_lock, pthread_mutex_unlock
__kmp_barrier	mppbar	To execute a barrier wait	pthread_mutex_lock, pthread_cond_wait, pthread_mutex_unlock

In the parallelism translation phase, KAP significantly restructures a program by moving the code in a parallel region to a separate subroutine. A call to the SMP support library replaces the parallel region. This call references the new subroutine. KAP examines the scope of each variable used in the parallel region and, if possible, converts each variable to a local variable of the new subroutine. Otherwise, the variable becomes an argument to the subroutine so that it can be passed back out of the parallel region.

Converting variables to local variables makes accessing these variables more efficient. A variable that is referenced outside the parallel region cannot be made local and must be passed as an argument.

Shared Memory Multiprocessor Architecture Concerns

Given its parallelism model, the KAP preprocessor requires operating system and hardware support from the system for efficient parallel execution. There are three areas of concern: thread creation and scheduling, synchronization between threads, and data caching and system bus bandwidth.

Thread Creation and Scheduling.

Thread creation is the most expensive operation. The X3H5 standard minimizes the need for creating threads through the use of parallel regions. The SMP support library goes further by reusing threads from one parallel region to the next. The SMP support library examines the value of an environment variable to determine how many threads to use. The appropriate scheduling of threads onto hardware processors is extremely important for efficient execution. The support library relies on the DECthreads implementation to achieve this. For the most efficient operation, the library should schedule at most one thread per processor.

Synchronization between Threads.

In the KAP model of parallelism, threads can synchronize at

- A point where loop iterations are scheduled
- A point where data passes between iterations (for collection of local reduction variables only)
- A barrier point leaving a work-sharing construct
- Single processor sections

Two versions of the SMP support library have been developed: one

with spin locks for a single-user environment and the second with mutex locks for a multiuser environment. Either library works in either environment; however, using the spin lock version in a single-user environment yields the most efficient parallelism.

Using spin locks in a multiuser environment may waste processor cycles when there are other users who could use them. Using mutex locks for a single-user environment creates unnecessary operating system overhead. In practice, however, a system may shift from single-user to multiuser and back again in the course of a single run of a large program. Therefore, KAP supports all lock-environment combinations.

Data Caching and System Bus Bandwidth. Multiprocessor Alpha systems support coherent caches between processors.[5] To use these caches efficiently, as a policy, KAP localizes data as much as possible, keeping repeated references within the same processor. Localizing data reduces the load on the system bus and reduces the chances of cache thrashing.

When all the processors simultaneously request data from the memory, system bus bandwidth can limit SMP performance. If optimizations enhance cache locality, less system bus bandwidth is used, and therefore SMP performance is less likely to be limited.

KAP TECHNOLOGY

This section covers the issues of data dependence analysis, preprocessor architecture, and the selection of loops to parallelize.

Data Dependence Analysis---The Kernel of Parallelism Detection

DEC Fortran and DEC C have standard rules for the order of execution of statements and expressions. These rules are based on a serial model of program execution. Data dependence analysis allows a compiler to see where this serial order of execution can be modified without changing the meaning of the program.

Types of Dependence. KAP works with the four basic types of dependence:[6]

1. Flow dependence, i.e., when a program writes a variable before it reads the variable

2. Antidependence, i.e., when a program reads a variable before it writes the variable

3. Output dependence, i.e., when a program writes the same

variable twice

4. Control dependence, i.e., when a program statement depends on a previous conditional

Because dependences involve two actions on the same variable, for example, a write and then a read, KAP uses the term dependence arc to represent information flow, in this example from the write to the read.

Since these dependences can prevent parallelization, KAP uses various optimizations to eliminate the different dependences. For example, an optimization called scalar renaming removes some but not all antidependences.

Loop-related Dependences. When dependences occur within a loop, the control flow relations are captured with direction vector symbols tagged to each dependence arc.[2] The transformations that can be applied to a loop depend on what dependence direction vectors exist for that loop. The symbols used in KAP and their meanings are

`=` The dependence occurs within the same loop iteration.

`>` The dependence crosses one or several iterations.

`<` The dependence goes to a preceding iteration of the loop.

`*` The dependence relation between iterations is not clear.

or a combination of the above, for example,

`<>` The dependence is known not to be on the same iteration.

When a dependence occurs in a nested loop, KAP uses one symbol for each level in the loop nest. A dependence is said to be carried by a loop if the corresponding direction vector symbol for that loop includes <, >, or *.

In the following program segment

```
1  for (i=1; i<=n; i++) {
2     temp = a[i];
3     a[i] = b[i];
4     b[i] = temp; }
```

there is a flow dependence from statement 2 to statement 4. There is an antidependence from statement 2 to statement 3 and from statement 3 to statement 4. There are control dependences from statement 1 to statements 2, 3, and 4 because executing 2, 3, and 4 depends on the i<=n condition. All these dependences are on the same loop iteration; their direction vector is =.

Some dependences in this program cross loop iterations. Because temp is reused on each iteration, there is an output dependence from statement 2 to statement 2, and there is an antidependence from statement 4 to statement 2. These two dependences are carried by the loop in the program segment and have the direction vector >.

Data Dependence Analysis. The purpose of dependence analysis is to build a dependence graph, i.e., the collection of all the dependence arcs in the program. KAP builds the dependence graph in two stages. First, it builds the best possible conservative dependence graph.[7] Then, it applies filters that identify and remove dependences that are known to be conservative, based on special circumstances.

What does the phrase "best possible conservative dependence graph" mean? Because the values of a program's variables are not known at preprocessing time, in some situations it may not be clear whether a dependence actually exists. KAP reflects this situation in terms of assumed dependences based on imperfect information. Therefore, a dependence graph must be conservative so that KAP does not optimize a program incorrectly. On the other hand, a dependence graph that is too conservative results in insufficient optimization.

In building the best possible dependence graph, KAP uses the following optimizations: constant propagation, variable forward substitution, and scalar expansion. KAP does not, however, leave the program optimized in this manner unless the optimizations will improve performance.

Advanced Ways to Affect Dependences. When there are assumed dependences in the program, KAP may not have enough information to decide on parallelism opportunities. KAP implements two techniques to mitigate the effects of imperfect information at preprocessing time: assertions and alternate code sequences.

Assertions, which are similar to directives in syntax, are used to provide information not otherwise known at preprocessing time. KAP supports many assertions that have the effect of removing assumed dependences. Table 3 shows KAP assertions and their effects.[8,9] When the user specifies an assertion, the information contained in the assertion is saved by a data abstraction called the oracle. When an optimization requests that a data dependence graph be built for a loop, the dependence analyzer inquires whether the oracle has any information about certain arcs that it wants to remove.

Table 3 KAP Assertions

Assertion	Specifiers	Primary Effect
[NO] ARGUMENT ALIASING		Removes assumed dependence arcs
[NO] BOUNDS VIOLATIONS		Removes assumed dependence arcs
CONCURRENT CALL		Removes assumed dependence arcs
DO (<specifier>)	SERIAL, CONCURRENT	Guides selection of loop order strongly
DO PREFER (<specifier>)	SERIAL, CONCURRENT	Guides selection of loop order loosely
[NO] EQUIVALENCE HAZARD		Removes assumed dependence arcs (Fortran only)
[NO] LAST VALUE NEEDED (<specifier>)	Variable names for which [no] last value is needed	Tunes the parallel code and sometimes removes assumed dependences
PERMUTATION (<specifier>)	Names of permutation variables	Removes assumed dependence arcs
NO RECURRENCE (<specifier>)	Names of recurrence variables	Removes assumed dependence arcs
RELATION(<specifier>)	Relation loop index known to be true	Removes assumed dependence arcs
NO SYNC		Tunes the parallel code which is produced

When accurate information is not known at compile time, a few KAP optimizations generate two versions of the source program loop: one assumes that the assumed dependence exists; the other assumes that it does not exist. In the latter case, KAP can apply subsequent optimizations, such as parallelizing the loop. KAP applies the two-version loop optimizations selectively to avoid dramatically increasing the size of the program. However, the payback of parallelizing a frequently executed loop warrants their use.

For example, the KAP C pointer disambiguation optimization is employed in cases in which C pointers are used as a base address and then incremented in a loop. Neither the base address of a pointer nor how many times the pointer will be incremented is usually known at compile time. At run time, however, they can be computed in terms of a loop index. KAP generates code that checks the range of the pointer references at the tail and at the head of a dependence. If the two ranges do not overlap, the dependence does not exist and the optimized code is executed.

KAP Preprocessor Architecture

A controversial control architecture decision in KAP is to organize the preprocessor as a sequence of passes, generally one for each optimization performed. This design decision was controversial because of the following concerns:

- Run-time inefficiency would occur in processing programs because each pass would sweep through the intermediate representation for the program being processed, causing some amount of virtual memory thrashing.

- Added software development cost would be incurred because the KAP code that loops through the intermediate representation would be repeated in each pass.

The second concern has been dispelled. The added modularity of KAP, provided by its multipass structure, has saved development time as KAP has grown from a moderately complex piece of code to an extremely complex piece of code.

The KAP preprocessor uses more than 50 major optimizations. The pass structure has helped to organize them. In some cases, such as cache management, one optimization is broken into several passes. KAP performs some basic optimizations, e.g., deadcode elimination, more than once in different ways. In some cases, such as scalar expansion, KAP performs an optimization to uncover other optimizations and then performs the reverse optimization to tighten up the program again.

The run-time efficiency issue is still of interest. There is always some benefit to making the preprocessor smaller and faster.

Selecting Loops to Parallelize

Parallelizing a loop can greatly enhance the performance of the program. Testing whether a loop can be parallelized is actually quite simple, given the data dependence analysis that KAP performs. A loop can be parallelized if there are no dependence arcs carried by that loop. The situation, however, can be more complicated. If the program contains several nested loops, it is important to pick the best loop to parallelize. Additionally, it may be possible not only to parallelize the loop but also to optimize the loop to enhance its performance. Moreover, the loops in a program can be nested in very complex structures so that there are many different ways to parallelize the same program. In fact, the best option may be to leave all the loops serial because the overhead of parallel execution may outweigh the performance improvement of using multiple processors.

The KAP preprocessor optimizes programs for parallelism by searching for the optimum program in a set of possible configurations, i.e., ways in which the original program can be transformed for parallel execution. (In this regard, KAP optimizes programs from a classical definition of numerical optimization.) There is an objective function for evaluating each configuration. Each member of the set of configurations is called a loop order. The optimum program is the loop order whose objective function has the highest performance score, as discussed later in this section.

Descriptions of loop orders, the role of dependence analysis, and the objective function, i.e., how each program is scored, follow.

Loop Orders.

A loop order is a combination of loop transformations that the KAP preprocessor has performed on the program. The loop transformations that KAP performs while searching for the optimal parallel form are

- Loop distribution
- Loop fusion
- Loop interchange

Loop distribution splits a loop into two or more loops. Loop fusion merges two loops. Loop fusion is used to combine loops to increase the size of the parallel tasks and to reduce loop overhead.

Loop interchange occurs between a pair of loops. This

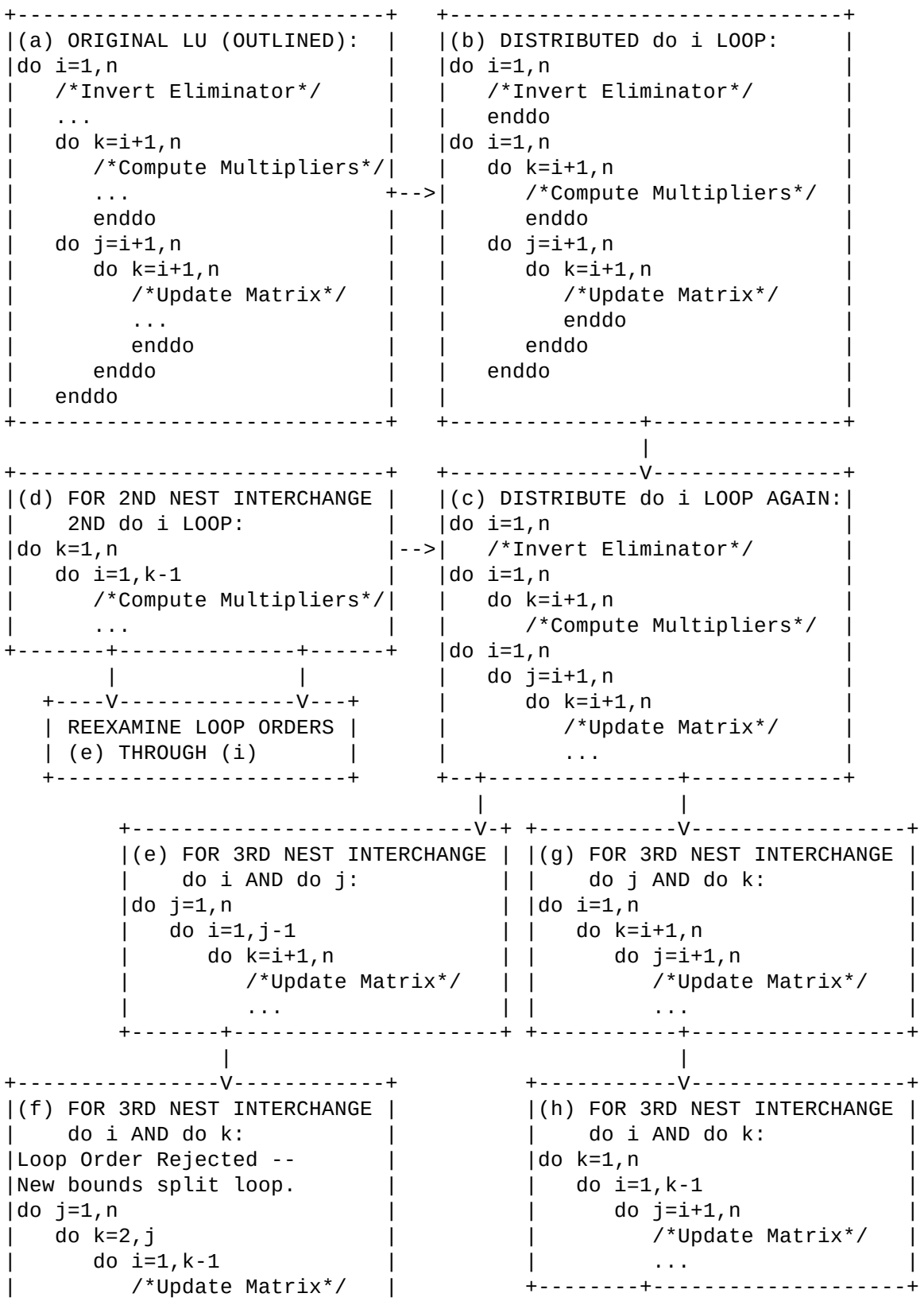
transformation takes the inner loop outside the outer loop, reversing their relation. If a loop is triply nested, there are three factorial (3!), i.e., six, different ways to interchange the loops. Each order is arrived at by a sequence of pairwise interchanges.

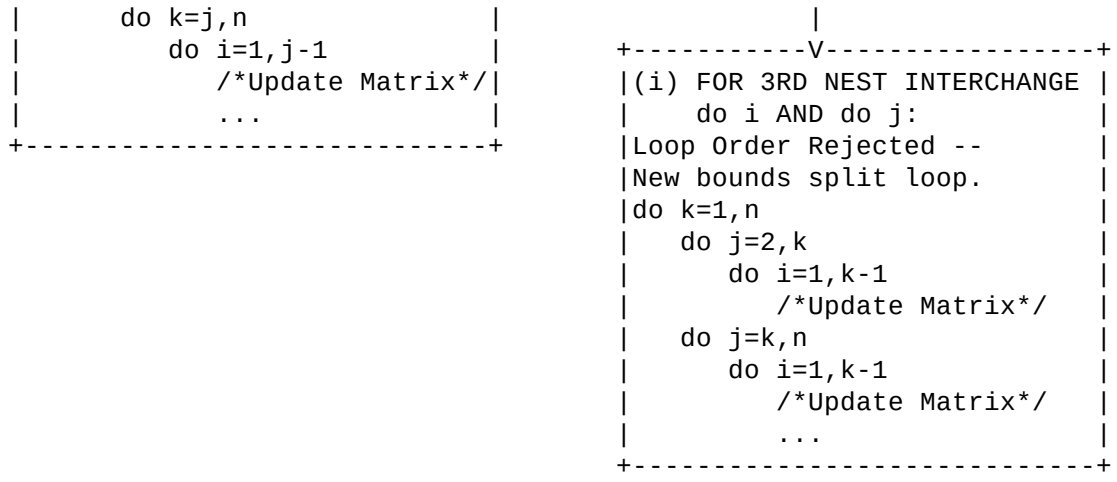
To increase the opportunities to interchange loops, KAP tries to make a loop nest into one that is perfectly nested. This means that there are no executable statements between nested loop statements. Loop distribution is used to create perfectly nested loops.

KAP examines all possible loop orders for each loop nest. Each loop nest is treated independently because no transformations between loop nests occur at this phase of optimization.

For example, an LU factorization program consists of one loop nest that is three deep and not perfectly nested. Figure 2 shows the loop orders. Loop order (a) is the original LU program. The KAP preprocessor first distributes the outer loop in loop orders (b) and (c). Next, KAP performs a loop interchange on the second loop nest which is two deep, as shown in loop order (d). Then, KAP interchanges the third loop nest in loop orders (e) through (i). Note that KAP eliminates some loop orders, (i) for example, when the loop-bound expressions cannot be interchanged. As explained above, there are six different loop orders because the nest is triply nested. Since the loop nest in (d) was originally nested with the triply nested loop at the outermost do loop, KAP will reexamine these six loop orders after the interchange in (d).

Figure 2 Loop Orders for LU Factorization




do k=j,n
do i=1,j-1
/*Update Matrix*/
...
(i) FOR 3RD NEST INTERCHANGE
do i AND do j:
Loop Order Rejected --
New bounds split loop.
do k=1,n
do j=2,k
do i=1,k-1
/*Update Matrix*/
do j=k,n
do i=1,k-1
/*Update Matrix*/
...

Dependence Analysis for Loop Orders. Before a loop order can be evaluated for efficiency, KAP determines the validity of the loop order. A loop order is valid if the resulting program would produce equivalent behavior. KAP tests validity by examining the dependences in the dependence graph according to the transformation being applied.

For example, the test for loop interchange validity involves searching for dependence direction vectors of a certain type. The direction vector (<,>) indicates that a loop interchange is invalid. The direction vectors (<,*), (*,>), or (*,*), if present, also indicate that the loop interchange may be invalid.

Evaluation of a Loop Order. After the KAP preprocessor determines that a loop order is valid, it scores the loop order for performance. KAP considers two major factors: (1) the amount of work that will be performed in parallel and (2) the memory reference efficiency.

The memory reference efficiency of a loop order can degrade performance so much that it outweighs the performance gained by executing a loop in parallel. On an SMP, if a processor references one word on a cache line, it should reference all the words contiguously on that line. In Fortran, a two-dimensional array reference, A(i,j), should be parallelized so that the j loop is parallel and each processor references contiguous columns of memory. If a loop order indicated that the i loop is parallel, this reference would score low. If a loop order indicated that the j loop is parallel, it would score high. The score for the loop order is the sum of the scores for all the references, and the highest-scoring loop order is preferred.

The score for a loop order depends on which loops in the order can be parallelized. For a given loop nest, there may be several (or no) loops that can be parallelized. The first step is to determine if any loops can be parallelized. If multiple loops can be parallelized, KAP selects the best one. KAP chooses at most one loop for parallel execution.

KAP tests loops to determine whether they can be executed in parallel by analyzing both the statements in the loop and the dependence graph. The loop may contain certain statements that block concurrentization. I/O statements or a call to a function or subroutine are examples. (Users can code KAP assertions to flag these statements as parallelizable.) Second, data dependence conditions may preclude parallelization. In general, a loop that carries a dependence is not parallelizable. (In some cases, the user may override the data dependence condition by allowing synchronization between loop iterations.) Finally, the user may give assertions that indicate a preference for making a loop parallel or for keeping it serial.

Barring data dependence conditions that would prevent parallelization, the amount of work that will be performed in parallel determines the score of parallelizing a loop. (The user can also specify with a directive that loops should not be parallelized unless they score greater than a specified value.) In this manner, KAP prefers to parallelize outer loops or loops that are interchanged to the outside because they contain the most work to amortize the overhead of creating threads for parallelism.

The actual parallelization process is even more complex than this discussion indicates. KAP applies a number of optimizations to improve the quality of the parallel code. If there is a reduction operation across a loop, KAP parallelizes the loop. Too much loop distribution can decrease program efficiency, so loop fusion is run to try to coalesce loops.

PERFORMANCE ANALYSIS

How does the KAP preprocessor perform on real applications? The answer is as complex as the software written for these applications. Consider the real-world example, DYNA3D, which demonstrates some KAP strengths and weaknesses.

DYNA3D is nonlinear structural dynamics code that uses the finite element analysis method. The code was developed by the Lawrence Livermore National Laboratory Methods Development Group and has been used extensively for a broad range of structural analysis problems. DYNA3D contains about 70,000 lines of Fortran code in more than 700 subroutines.

When KAP is being used on a large program, it is sometimes preferable to concentrate on the compute-intensive kernels. For example, KAP developers ran six of the standard benchmarks for DYNA3D through a performance profiling tool and isolated two groups of three subroutines that account for approximately 75 percent of the run time in these cases. This data is shown in Table 4.

Table 4 Performance Profiles of Six DYNA3D Problems

Problem	Profile (First Two Initials of the Subroutine and Percent of Run Time)	Key Call Sequences*
NIKE2D Example	ST 19%, FO 15%, FE 12%, PR 10%, HG 7%, HR 5%	(a) and (b)
Cylinder Drop	ST 20%, FO 15%, FE 11%, PR 10%, HG 7%, HR 5%	(a) and (b)
Bar Impact interest	WR 17%, ST 7%, FE 6%	None of
Impacted Plate	SH 22%, TN 16%, TA 16%, YH 14%, BL 7%	(c)
Single Contact	YH 24%, SH 21%, TN 7%, TA 7%, BL 6%	(c)
Clamped Beam	EL 12%, SH 12%, TN 8%, TA 8%, BL 6%	(c)

*Call Sequences

(a) ST is called; ST calls PR; and then FE is called.
(b) HR is called; HR calls HG; and then FO is called.
(c) BL calls SH, then TA, and then TN.

KAP's performance on some of these key subroutines appears in Table 5. KAP parallelized all the loops in these subroutines. Since DYNA3D was designed for a CRAY-1 vector processor, it is perhaps to be expected that the KAP preprocessor would perform well. KAP, however, is intended for a shared memory multiprocessor rather than for a vector machine. For this reason, KAP does more than parallelize the loops. The entries in the column labeled "Number of Loops after Fusion" show how KAP reduced loop overhead by fusing as many loops together as it could. KAP fused the five loops in subroutine STRAIN into three loops and fused all nine loops in subroutine PRTAL.

Table 5 KAP's Performance on Key Subroutines

Subroutine	Number of Loops	Number of Loops Parallelized	Maximum Nest Depth	Number of Loops after Fusion
STRAIN	5	5	1	3
PRTAL	9	9	1	1
FELEN	6	6	1	1
FORCE	9	9	2	2
HRGMD	5	5	1	3
HGX	4	4	1	1

Another example of KAP's optimization for an SMP system is that in the doubly nested loop cases, such as subroutine FORCE (see Figure 3), the KAP preprocessor automatically selects the outer loop for parallel execution. In contrast, a vector machine such as the CRAY-1 prefers the inner loop.

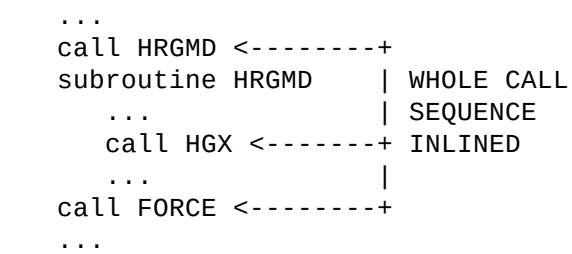
Figure 3 Parallel Loop Selection

```
    subroutine FORCE                / OUTER LOOP PARALLIZED
    ...                            /
    do 60 n = 1,nnc <--------------+
       lcn = lczc + n + nh12 - 1
       i0 = ia(lnc)
       i1 = ia(lcn + 1) - 1
    cdir$ ivdep
       do 50 i = i0,i1
          e(1,ix(i)) =
          e(1,ix1(i)) + ep11(i)
          ...
    50 continue
       ...
    60 continue
```

Because the kernels of DYNA3D code span multiple subroutines, cross compilation optimization is suggested. There are three ways to do this: inlining, interprocedural analysis, and directives specifying that the inner subroutines can be concurrentized. Using KAP's inlining capability gives KAP the most freedom to optimize the program because in this manner KAP can restructure code across subroutines.

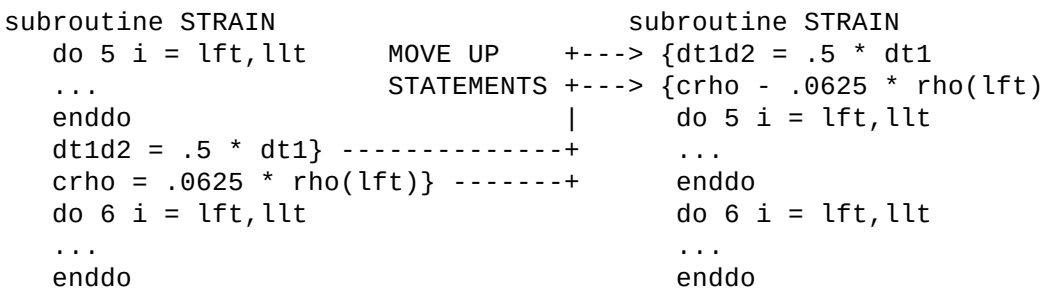
Figure 4 shows part of the call sequence of subroutine SOLDE. (Subroutine SOLDE contains call sequence (b) of Table 4.) Subroutine SOLDE calls subroutine HRGMD which calls subroutine HGX. Then subroutine SOLDE calls subroutine FORCE. KAP supports inlining to an arbitrary depth. Inlining in KAP can be automatic or controlled from the command line. In this case, we did not want to enable inlining automatically to depth two of subroutine SOLDE because it contains calls to many other subroutines that are not in the kernel. Here, the user specified the subroutines to inline on the command line. When the user specified inlining, KAP fused all the loops in subroutines HRGMD, HGX, and FORCE to minimize loop overhead, and then it parallelized the fused loop.

Figure 4 Inlining a Kernel



In some cases, the user can make simple restructuring changes that improve KAP's optimizations. Figure 5 shows a case in which fusion was blocked by two scalar statements between a pair of loops. The first loop does not assign any values to the variables used to create these scalars, so the user can move the assignments above the loop to enable KAP to fuse them.

Figure 5 Assisted Loop Fusion



Finally, the user can elect to specify the parallelism directly. Figure 6 shows subroutine STRAIN with X3H5 directives used to describe the parallelism. In this case, the user elected to keep the same unfused loop structure as in the original code. This case is not dramatically less efficient than the fused version because the parallel region causes KAP to fork threads only once.

Figure 6 X3H5 Explicit Parallelism

ALL c*kap* STATEMENTS
ARE X3H5 EXPLICIT
PARALLEL DIRECIVES.

```
subroutine STRAIN
c*kap* parallel region
c*kap*& shared(dxy,dyx,d1)
c*kap*& local(i,dt1d2)
c*kap* parallel do
   do 5 i = lft,llt
   dyx(i) = ...
5 continue
c*kap* end parallel do
c*kap* barrier
   dt1d2 = ...
c*kap* parallel do
   do 6 i = lft,llt
      d1 = dt1d2 * (dxy(i) + dyx(i))
6 continue
c*kap* end parallel do
c*kap* end parallel region
```

A very sophisticated example of KAP usage occurs when a user inputs a program to KAP that has already been optimized by KAP. This is an advantage of a preprocessor that does not apply to a compiler because a preprocessor produces source code output. In this case, the statements shown in Figure 6 were generated by KAP to illustrate X3H5 parallelism. A user may want to perform some hand optimization on this output, such as removing the barrier statement, and then optimize the modified program with KAP again.

CHALLENGES THAT REMAIN

Although the KAP preprocessor is a robust tool that performs well in a production software development environment, several challenges remain. Among them are adding new languages, further enhancing the optimization technology, and improving KAP's everyday usability.

As the popular programming languages evolve, KAP evolves also. KAI will soon extend KAP support for DEC Fortran to Fortran 90 and is developing C++ optimization capabilities.

In optimization technology, KAI's goal is to make an SMP server as easy to use as a single-processor workstation is today. "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing" contains a leading-edge analysis of parallelization technology.[10] The research reported shows that further developing current techniques can improve optimization technology. These techniques frequently involve the grand challenge of compiler optimization---whole program analysis.

In a much more pragmatic direction, the KAP preprocessor should be integrated with Digital's compiler technology at the intermediate representation level. Such integration would increase processing efficiency because the compiler would not have to reparse the source code. In addition, integration would increase the coordination between KAP and the compiler to improve performance for the end user.

Increasing the usability of the KAP preprocessor, however, benefits the end user directly. KAP engineers frequently talk to beta users and encourage feedback. The following are examples of user comments:

- Optimizing programs is difficult when no subroutine in the program takes more than a few percent of the run time. As its usability in this area improves, KAP will become a substantial productivity aid. If a program is generally slow, optimizing repeated usage patterns will allow the programmer to use a comfortable programming style and still expect peak system performance.

- Increasing feedback to the user would improve KAP's usability. When KAP cannot perform an optimization, often the user can help in several ways (e.g., by providing more information at compile time, by changing the options or directives, or by making small changes to the source code). KAP does not always make it clear to the user what needs to be done. Providing such feedback would improve KAP's usability.

- Integration with other performance tools would be useful. Alpha systems have a good set of performance monitoring tools that can provide clues about what to optimize in a program and how. The next release of the KAP preprocessor will provide some simple tools that a user can employ to integrate KAP with tools like prof and to track down performance differences.

On a final note, the fact that KAP does not speed up a program should not always be cause for disappointment. Some programs already run as fast as possible without the benefit of a KAP preprocessor.

ACKNOWLEDGMENTS

We wish to acknowledge the Lawrence Livermore National Laboratory Methods Development Group and other users for providing applications that give us insight into how to improve the KAP preprocessor. We would like to thank those at Digital who have been instrumental in helping us deliver KAP on the DEC OSF/1 platform, especially Karen DeGregory, John Shakshober, Dwight Manley, and Dave Velten. Everyone at Kuck & Associates participated in the making of this product but of special note are Mark Byler, Debbie Carr, Ken Crawford, Steve Healey, David Nelson, and Sree Simhadri.

REFERENCES

1. D. Blickstein et al., "The GEM Optimizing Compiler System," Digital Technical Journal, vol. 4, no. 4 (Special Issue 1992): 121-136.

2. M. Wolfe, Optimizing Supercompilers for Supercomputers (Cambridge, MA: MIT Press, 1989).

3. Parallel Processing Model for High Level Programming Languages, ANSI X3H5 Document Number X3H5/94-SD2, 1994.

4. P. Tu and D. Padua, "Automatic Array Privatization," Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, vol. 768 of Lecture Notes in Computer Science (New York: Springer-Verlag, 1993): 500-521.

5. B. Maskas et al., "Design and Performance of the DEC 4000 AXP Departmental Server Computing System," Digital Technical Journal, vol. 4, no. 4 (Special Issue 1992): 82-99.

6. R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," ACM Transactions on Programming Languages and Systems, vol. 9, no. 4 (October 1987): 491-542.

7. U. Banerjee, Dependence Analysis for Supercomputing (Norwell, MA: Kluwer Academic Publishers, 1988).

8. KAP for DEC Fortran for DEC OSF/1 AXP User Guide (Maynard, MA: Digital Equipment Corporation, 1994).

9. KAP for C for DEC OSF/1 AXP User Guide (Maynard, MA: Digital Equipment Corporation, 1994).

10. W. Blume et al., "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing," CSRD Report No. 1348 (Urbana, IL: Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1994).

TRADEMARKS

The following are trademarks of Digital Equipment Corporation:
DEC, DEC Fortran, DEC OSF/1, DECthreads, and Digital.

CRAY-1 is a registered trademark of Cray Research, Inc.
KAP is a trademark of Kuck & Associates, Inc.

BIOGRAPHIES

Robert H. Kuhn Robert Kuhn joined Kuck & Associates as the Director of Products in 1992. His functions are to formulate technical business strategy and to manage product deliveries. From 1987 to 1992, he worked at Alliant Computer Systems, where he managed compiler development and application software for parallel processing. Bob received his Ph.D. in computer science from the University of Illinois at Champaign-Urbana in 1980. He is the author of several technical publications and has participated in organizing various technical conferences.

Sanjiv M. Shah Sanjiv Shah received a B.S. in computer science and mathematics (1986) and an M.S. in computer science and engineering (1988) from the University of Michigan. In 1988, he joined Kuck & Associates' KAP development group as a research programmer. He has since been involved in researching and developing the KAP Fortran and C products and managing the KAP development group. Currently, Sanjiv leads the research and development for parallel KAP performance.

Bruce Leasure Bruce Leasure, one of three founders of Kuck & Associates in 1979, serves as Vice President of Technology and is the chief scientist for the company. As a charter member and executive director of the Parallel Computing Forum (PCF), a standards-setting consortium, he was a leader in efforts to standardize basic forms of parallelism. The PCF subsequently became the ANSI X3H5 committee for Parallel Program Constructs for High-level Languages, which he chaired. Bruce received B.S. and M.S. degrees in computer science from the University of Illinois at Champaign-Urbana.