Chapter 15. The Auto−Parallelizing Option (APO)

Note: APO is licensed and sold separately from the MIPSpro C/C++ compilers. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your sales representative.

The Auto−Parallelizing Option (APO) enables the MIPSpro C/C++ compilers to optimize parallel codes and enhances performance on multiprocessor systems. APO is controlled with command line options and source directives.

APO is integrated into the compiler; it is not a source−to−source preprocessor. Although run−time performance suffers slightly on single−processor systems, parallelized programs can be created and debugged with APO enabled.

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the compiler analyzes and restructures the program with little or no intervention by you. With APO, the compiler automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques.

APO integrates automatic parallelization with other compiler optimizations, such as interprocedural analysis (IPA), optimizations for single processors, and loop nest optimization (LNO). In addition, run−time and compile−time performance is improved.

15.1 C/C++ Command Line Options That Affect APO

Several `cc(1)` and `CC(1)` command line options control APO's effect on your program. For example, the following command line invokes APO and requests aggressive optimization:

```
CC −apo −O3 zebra.c  
```

The following subsections describe the effects that various C/C++ command line options have on APO.

Note: If you invoke the loader separately, you must specify the `−apo` option on the `ld(1)` command line.

15.1.1 `−apo`

The `−apo` option invokes APO. When this option is enabled, the compiler automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so. Specifying `−apo` also enables the `−mp` option, which enables recognition of the parallel directives inserted into your code.

15.1.2 `−apokeep` and `−apolist`

The `−apokeep` and `−apolist` options control output files. Both options generate *`file`*`.list`, which is a listing file that contains information on the loops that were parallelized and explains why others were not parallelized.

When `−apokeep` is specified, the compiler writes *`file`*`.list`, and in addition, it retains *`file`*`.anl` and *`file`*`.m`. The ProMPF tools use *`file`*`.anl`. For more information on ProMPF, see the *ProDev ProMP User's Guide*. *`file`*`.m` is an annotated version of your source code that shows the insertion of multiprocessing directives.

When `−IPA` is specified with the `−apokeep` option, the default settings for IPA suboptions are used with the exception of `-IPA:inline`, which is set to `OFF`.

For more information on the content of *file*.list, *file*.anl, and *file*.m, see Section 15.2.

> **Note:** Because of data conflicts, do not specify the `-mplist` or `-CLIST` options when `-apokeep` is specified.

15.1.3 -CLIST:...

This option generates a C/C++ listing and directs the compiler to write an equivalent parallelized program in *file*.w2c.c. For more information on the content of *file*.w2c.c, see Section 15.2.

15.1.4 -IPA:...

Interprocedural analysis (IPA) is invoked by the `-ipa` or `-IPA` command line option. It performs program optimizations that can only be done by examining the whole program, not parts of a program.

When APO is invoked with IPA, only those loops whose function calls were determined to be safe by the APO are optimized.

If IPA expands functions inline in a calling routine, the functions are compiled with the options of the calling routine. If the calling routine is not compiled with `-apo`, none of its inlined functions are parallelized. This is true even if the functions are compiled separately with `-apo` because with IPA, automatic parallelization is deferred until link time.

When `-apokeep` or `-pcakeep` are specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption, which is set to `OFF`.

For more information on the effect of IPA, see Section 15.5.1.2. For more information on IPA itself, see the `ipa(5)` man page.

15.1.5 -LNO:...

The `-LNO` options control the Loop Nest Optimizer (LNO). LNO performs loop optimizations that better exploit caches and instruction−level parallelism. The following LNO options are of particular interest to APO users:

- `-LNO:auto_dist=on`. This option requests that APO insert data distribution directives to provide the best memory utilization on Origin2000 systems.
- `-LNO:ignore_pragmas=`*setting*. This option directs APO to ignore all of the directives and assertions described in Section 15.4.
- `-LNO:parallel_overhead=`*num_cycles*. This option allows you to override certain compiler assumptions regarding the efficiency to be gained by executing certain loops in parallel rather than serially. Specifically, changing this setting changes the default estimate of the cost to invoke a parallel loop in your run−time environment. This estimate varies depending on your particular run−time environment, but it is typically several thousand machine cycles.

You can view the transformed code in the original source language after LNO performs its transformations. Two translators, integrated into the compiler, convert the compiler's internal representation into the original source language. You can invoke the desired translator by using the `CC -CLIST:=on` option. For example, the following command creates an `a.out` object file and the C/C++ file `test.w2c.c`:

```
CC -O3 -CLIST:=on test.c
```

Because it is generated at a later stage of the compilation, this `.w2c.c` file differs somewhat from the `.w2c.c` file generated by the `-apokeep` option (see Section 15.1.2). You can read the `.w2c.c` file, which is a compilable C/C++ representation of the original program after the LNO phase. Because the LNO is not a preprocessor, recompiling the *file*.w2c.c can result in an executable that differs from the original compilation of the `.c` file.

15.1.6 *-O3*

To obtain maximum performance, specify `-O3` when compiling with APO enabled. The optimization at this level maximizes code quality even if it requires extensive compile time or relaxes the language rules. The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating–point results due to the relaxation of operation–ordering rules. Floating–point optimization is discussed further in Section 15.1.7.

15.1.7 *-OPT:...*

The `-OPT` command line option controls general optimizations that are not associated with a distinct compiler phase.

The `-OPT:roundoff=`*n* option controls floating–point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules.

When `-O3` is in effect, the default rounding setting is `-OPT:roundoff=2`. This setting allows transformations with extensive effects on floating–point results. It allows associative rearrangement across loop iterations and the distribution of multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round–off errors for a wide range of floating–point operands.

At `-OPT:roundoff=2` or `3`, APO can change the sequence of a loop's floating–point operations in order to parallelize it. Because floating–point operations have finite precision, this change can cause slightly different results. If you want to avoid these differences by not having such loops parallelized, you must compile with `-OPT:roundoff=0` or `-OPT:roundoff=1`.

Example. APO parallelizes the following loop when compiled with the default settings of `-OPT:roundoff=2` and `-O3`:

```
float a, b[100];  
for(i=0; i<100; i++)  
    a = a + b[i];
```

At the start of the loop, each processor gets a private copy of `a` in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of `a` can be different from the value generated by a version of the loop that is not parallelized.

15.1.8 *-pca, -pcakeep, -pcalist*

The `-pca` option invokes APO. For the O32 ABI, the `-pca` option invokes Power C. The `-pcakeep` and `-pcalist` options control output files.

When `-IPA` is specified with the `-pcakeep` option, the default settings for IPA suboptions are used with the exception of `-IPA:inline`, which is set to `OFF`.

> **Note:** These options are outmoded. The preferred way of invoking APO is through the `-apo` option, and the preferred way to obtain a listing is through the `-apolist` option. For more information on these options, see Section 15.1.1, and Section 15.1.2.

15.1.9 file

Your input file.

For information on files used and generated when APO is enabled, see Section 15.2.

15.2 Files

APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. You can use these listings to identify constructs that inhibit parallelization. When you remove these constructs, you can often improve program performance dramatically.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the WorkShop Pro MPF Parallel Analyzer View described in *ProDev ProMP User's Guide*.

The following sections describe the content of the files generated by APO.

15.2.1 The `file.list` File

The `-apolist` and `-apokeep` options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is provided.

Example. The following function resides in file `testl.c`:

```
void sub(double arr[], int n)  
{  
    extern void foo(double);  
    int i;  
    for(i=1; i<n; i++) {  
      arr[i] += arr[i-1];  
    }  
    for(i=0; i<n; i++) {  
      arr[i] += 7.0;  
      foo(arr[i]);  
    }  
    for(i=0; i<n; i++) {  
      arr[i] += 7.0;  
    }  
}
```

File `testl.c` is compiled with the following command:

```
cc -O3 -n32 -mips4 -apolist -c testl.c
```

APO produces file `testl.list`:

```
Parallelization Log for Subprogram sub  
    5: Not Parallel  
         Array dependence from arr on line 6 to arr on line 6.  
  
    8: Not Parallel  
         Call foo on line 10.  
  
   12: PARALLEL (Auto) __mpdo_sub1  
```

Note the message for line 12. Whenever a loop is run in parallel, the parallel version of the loop is put in its own function. The MIPSpro profiling tools attribute all the time spent in the loop to this function. The last line indicates that the name of the function is `__mpdo_sub1`.

15.2.2 The `file.w2f.c` File

File *`file`*`.w2c.c` contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the original code were not parallelized. You can use this information to change the original program.

The compiler creates *`file`*`.w2c.c` by invoking the appropriate translator to turn the compiler's internal representations into C/C++. In most cases, the files contain valid code that can be recompiled, although compiling *`file`*`.w2c.c` without APO enabled does not produce object code that is exactly the same as that generated when APO is enabled on the original source.

The -apolist option generates *file*.w2c.c. Because it is generated at an earlier stage of the compilation, *file*.w2c.c from -apolist is more easily understood than *file*.w2c.c generated from -CLIST:=on option. On the other hand, the -CLIST option shows more of the optimizations that were performed. The parallelized program in *file*.w2c.c uses OpenMP directives.

Example. File testw2.c is compiled with the following command:

```
cc -O3 -n32 -mips4 -c -apo -apolist -c testw2.c
```

```
void trivial(float a[])  
{  
    int i;  
    for(i=0; i<10000; i++) {  
      a[i] = 0.0;  
    }  
}
```

Compiling testw2.c generates an object file, testw2.o, and listing file testw2.w2c.c, which contains the following code:

```
/*******************************************************  
 * C file translated from WHIRL Wed Oct 28 14:03:23 1998  
 *******************************************************/  
/* Include file-level type and variable decls */  
#include "testw2.w2c.h"  
  
  
void trivial(  
  _IEEE32(*a0)[])  
{  
  register _INT32 i0;  
  
  /* PARALLEL DO will be converted to SUBROUTINE __mpdo_trivial1 */;  
#pragma parallel  
  {  
#pragma pfor  
#pragma local(i0)  
#pragma shared(a0)  
    for(i0 = 0; i0 <= 9999; i0 = i0 + 1)  
    {  
      (*a0)[i0] = 0.0F;  
    }  
  }  
  return;  
} /* trivial */  
```

Note: WHIRL is the name for the compiler's intermediate representation.

As explained in Section 15.2.1, parallel versions of loops are put in their own functions. In this example, that function is __mpdo_trivial_1. #pragma omp parallel is an OpenMP directive that specifies a parallel region containing a single DO directive.

15.2.3 About the .m and .anl Files

The -apokeep option generates *file*.list. It also generates *file*.m and *file*.anl, which are used by Workshop Pro MPF.

file.m is similar to the *file*.w2c.c file but is more like original source code; it is based on OpenMP and mimics the behavior of the program after automatic parallelization.

WorkShop Pro MPF is a Silicon Graphics product that provides a graphical interface to aid in both automatic and manual parallelization for C/C++. The WorkShop Pro MPF Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. For more information, see the *ProDev ProMP User's*

Guide and the *Developer Magic: Performance Analyzer User's Guide.*

SpeedShop, another Silicon Graphics product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop includes a set of commands and a number of libraries to support the commands. For more information, see the *SpeedShop User's Guide*.

15.3 Running Your Program

Running a parallelized version of your program is no different from running a sequential one. The same binary output file can be executed on various numbers of processors. The default is to have the run−time environment select the number of processors to use based on how many are available.

You can change the default behavior by setting the `OMP_NUM_THREADS` environment variable, which tells the system to use an explicit number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

The `OMP_DYNAMIC` environment variable allows you to control whether the run−time environment should dynamically adjust the number of threads available for executing parallel regions to optimize system resources. The default value is `ON`. If `OMP_DYNAMIC` is set to `OFF`, dynamic adjustment is disabled.

For more information on these and other environment variables, see the `pe_environ(5)` man page.

15.4 Compiler Directives

APO works in conjunction with the OpenMP C/C++ API directives and with the Origin series directives. You can use these directives to manually parallelize some loop nests, while leaving others to APO.This approach has the following positive and negative aspects:

- As a positive aspect, the OpenMP and Origin series directives are well defined and deterministic. If you use a directive, the specified loop is run in parallel. This assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.
- The negative side to this is that you must carefully analyze the code to determine that parallelism is safe. In particular, you may need to specify special attributes for some variables, such as `private` or `reduction`, or specify explicit synchronizations, such as a `barrier` or a `critical` section.

In addition to the OpenMP and Origin series directives, you can also use the APO−specific directives described in this section. These directives give APO more information about your code.

> **Note:** APO also recognizes the Silicon Graphics multiprocessing directives. These directives are outmoded, and you must include the `-mp` option on the `CC(1)` command line in order for the compiler to recognize them. The OpenMP directive set is the preferred directive set for multiprocessing.

The APO directives can affect certain optimizations, such as loop interchange, during the compiling process. To direct the compiler to disregard any of the preceding directives, specify the `-x`*`dirlist`* option.

The APO directives are as follows:

- `#pragma concurrent call`. This directive directs APO to ignore dependencies in function calls that would inhibit parallelization. For more information on this directive, see Section 15.4.1.
- `#pragma concurrent`. This directive asserts that APO should not let perceived dependencies between two references to the same array inhibit parallelizing. For more information on this directive, see Section 15.4.2.
- `#pragma serial`. This directive requests that the following loop be executed in serial mode. For more information on this directive, see Section 15.4.3.
- `#pragma prefer concurrent`. This directive parallelizes the following loop if it is safe. For more information on this directive, see Section 15.4.4.

- `#pragma permutation (`*`array_name`*`)`. Asserts that array *array_name* is a permutation array. For more information on this directive, see Section 15.4.5.
- `#pragma no concurrentize` and `#pragma concurrentize`. The `#pragma no concurrentize` directive inhibits either parallelization of all loops in a function or parallelization of all loops in a file. The `#pragma concurrentize` directive overrides the `#pragma no concurrentize` directive, and its effect varies with its placement. For more information on these directives, see Section 15.4.6.

  Note: The compiler honors the following APO directives even if the `-apo` option is not included on your command line:

  - `#pragma concurrent call`
  - `#pragma prefer concurrent`
  - `#pragma permutation (`*`array_name`*`)`

15.4.1 *#pragma concurrent call*

The `#pragma concurrent call` directive instructs APO to ignore the dependencies of function and function calls contained in the loop that follows the assertion. The directive applies to the loop that immediately follows it and to all loops nested inside that loop. Other points to be aware of are the following:

Note: The directive affects the compilation even when `-apo` is not specified.

APO ignores potential dependencies in function `fred()` when it analyzes the following loop:

```
#pragma concurrent call  
for(i=0; i<n; i++) {  
    fred();  
    ...  
}
```

To prevent incorrect parallelization, make sure the following conditions are met when using `#pragma concurrent call`:

- A function inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.
- A function inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.

Example. The following code shows an illegal use of the directive. Function `fred()` writes to variable `x`, which is also read from by `wilma()` during other iterations, and the directive instructs APO to ignore this dependence.

```
void fred(float *b, int i, float *t) {  
    *t = b[i];  
}  
void wilma(float *a, int i, float *t){  
    a[i] = *t;  
}  
  
    #pragma concurrent call  
    for(i=0; i<m; i++) {  
       fred(b, i, &x);  
       wilma(a, i, &x);  
    }
```

The following example shows how you can manually parallelize the preceding example safely by ‘localizing’ variable `x` with a declaration `float x;` at the top of the loop body.

```
#pragma concurrent call  
for (i=0, i<m, i++) {  
   float x;  
   fred(b, i, &x);  
   wilma(a,i, &x);  
```

```
}
```

15.4.2 `#pragma concurrent`

The `#pragma concurrent` directive instructs APO, when analyzing the loop immediately following this directive, to ignore all dependencies between two references to the same array. If there are real dependencies between array references, the `#pragma concurrent` directive can cause APO to generate incorrect code.

> **Note:** This directive affects the compilation even when `-apo` is not specified.

The following example shows correct use of this directive when `m > n`:

```
#pragma concurrent  
for(i=0; i<n; i++)  
    a[i] = a[i+m];
```

Be aware of the following points when using this directive:

- If multiple loops in a nest can be parallelized, `#pragma concurrent` causes APO to parallelize the loop immediately following the assertion.
- Applying this directive to an inner loop can cause the loop to be made outermost by APO's loop interchange operations.
- This directive does not affect how APO analyzes function calls. For more information on APO's interaction with function calls, see Section 15.4.1.
- This directive does not affect how APO analyzes dependencies between two potentially aliased pointers.
- The compiler may find some obvious real dependencies. If it does so, it ignores this directive.

15.4.3 `#pragma serial`

The `#pragma serial` instructs APO not to parallelize the loop following the assertion; the loop is executed in serial mode. APO can, however, parallelize another loop in the same nest. The parallelized loop can be either inside or outside the designated sequential loop.

Example. The following code fragment contains a directive that requests that loop `j` be run serially:

```
for(i=0; i<m; i++) {  
    #pragma serial  
    for(j=0; j<n; j++)  
        a[i][j] = b[i][j];  
    ...  
}
```

The directive applies only to the loop that immediately follows it. For example, APO still tries to parallelize loop `i`. This directive is useful in cases like this when the value of `n` is known to be very small.

15.4.4 `#pragma prefer concurrent`

The `#pragma prefer concurrent` directive instructs APO to parallelize the loop immediately following the directive if it is safe to do so.

Example. The following code fragment encourages APO to run loop `i` in parallel:

```
#pragma prefer concurrent  
for(i=0; i<m; i++) {  
    for(j=0; j<n; j++)  
        a[i][j] = b[i][j];  
    ...  
}
```

When dealing with nested loops, APO follows these guidelines:

- If the loop specified by the `#pragma prefer concurrent` directive is safe to parallelize, APO parallelizes the specified loop even if other loops in the nest are safe.
- If the specified loop is not safe to parallelize, APO parallelizes a different loop that is safe.
- If this directive is applied to an inner loop, APO can interchange the loop and make the specified loop the outermost loop.
- If this directive is applied to more than one loop in a nest, APO parallelizes one of the specified loops.

15.4.5 *#pragma permutation*

When placed inside a function, the `#pragma permutation (`*`array_name`*`)` directive informs APO that *array_name* is a *permutation* array. A permutation array is one in which every element of the array has a distinct value.

The directive does not require the permutation array to be *dense*. That is, within the array, every `b[i]` must have a distinct value, but there can be gaps between the values, such as `b[1]` = 1, `b[2]` = 4, `b[3]` = 9, and so on.

Note: This directive affects compilation even when `-apo` is not specified.

Example. In the following code fragment, array `b` is declared to be a permutation array for both loops in `sub1()`:

```
void sub1(int n) {  
    int i;  
    extern int a[], b[];  
    for(i=0; i<n; i++) {  
         a[b[i]] = i;  
    }  
    #pragma permutation (b)  
    for(i=0; i<n; i++) {  
        a[b[i]] = i;  
    }  
}
```

Note the following points about this directive:

- As shown in the example, you can use this directive to parallelize loops that use arrays for indirect addressing. Without this directive, APO cannot determine that the array elements used as indexes are distinct.
- `#pragma permutation (`*`array_name`*`)` affects every loop in a function, even those that appear before it.

15.4.6 *#pragma no concurrentize, #pragma concurrentize*

The `#pragma no concurrentize` directive inhibits parallelization. Its effect depends on its placement.

- When placed inside functions, this directive inhibits parallelization. In the following example, no loops inside `sub1()` are parallelized:

```
void sub1() {  
#pragma no concurrentize  
    ...  
}
```

- When placed outside of a function, `#pragma no concurrentize` prevents the parallelization of all functions in the file, even those that appear ahead of it in the file. Loops inside functions `sub2()` and `sub3()` are not parallelized in the following example:

```
void sub2() {  
    ...  
}  
#pragma no concurrentize  
```

```
void sub3() {  
    ...  
}
```

The `#pragma concurrentize` directive, when placed inside a function, overrides a `#pragma no concurrentize` directive that is placed outside of it. Thus, this directive allows you to selectively parallelize functions in a file that has been made sequential with a `#pragma no concurrentize` directive.

15.5 Troubleshooting Incomplete Optimizations

Some loops cannot be safely parallelized and others are written in ways that inhibit APO's efficiency. The following subsections describe the steps you can take to make APO more effective. The sections that follow, and the topics they discuss, are as follows:

- Section 15.5.1, describes constructs that inhibit parallelization.
- Section 15.5.2, describes constructs that reduce performance of parallelized code.

15.5.1 Constructs That Inhibit Parallelization

A program's performance can be severely constrained if APO cannot recognize that a loop is safe to parallelize. APO analyzes every loop in a program. If a loop does not appear safe, it does not parallelize that loop. The following sections describe constructs that can inhibit parallelization:

- Section 15.5.1.1, describes basic data dependencies.
- Section 15.5.1.2, describes function calls.
- Section 15.5.1.3, describes `goto` statements.
- Section 15.5.1.4, describes problematic array subscripts.
- Section 15.5.1.5, describes conditionally assigned local variables.

In many instances, loops containing the previous constructs can be parallelized after minor changes. Reviewing the information generated in program *`file`*`.list`, described in Section 15.2.1, can show you if any of these constructs are in your code.

15.5.1.1 Loops Containing Data Dependencies

Generally, a loop is safe if there are no data dependencies, such as a variable being assigned in one iteration of a loop and used in another. APO does not parallelize loops for which it detects data dependencies.

15.5.1.2 Loops Containing Function Calls

By default, APO does not parallelize a loop that contains a function call because the function in one iteration of the loop can modify or depend on data in other iterations.

You can, however, use interprocedural analysis (IPA) to provide the MIPSpro APO with enough information to parallelize some loops containing function calls. IPA is specified by the `-ipa` command line option. For more information on IPA, see `ipa(5)` and the *MIPSpro Compiling and Performance Tuning Guide*.

You can also direct APO to ignore function call dependencies when analyzing the specified loops by using the `#pragma concurrent call` directive described in Section 15.4.1.

15.5.1.3 Loops Containing `goto` Statements

`goto` statements are unstructured control flows. APO converts most unstructured control flows in loops into structured flows that can be parallelized. However, `goto` statements in loops can still cause the following problems:

- Unstructured control flows. APO is unable to restructure all types of flow control in loops. You must either restructure these control flows or manually parallelize the loops containing them.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

For improved performance, remove `goto` statements from loops to be considered candidates for parallelization.

15.5.1.4 Loops Containing Problematic Array Constructs

The following array constructs inhibit parallelization and should be removed whenever APO is used:

- Arrays with subscripts that are indirect array references. APO cannot analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference `b[i]` is equal to the same value for different iterations of `i`:

```
for(i=0; i<n; i++)  
    a[b[i]] = ...  
```

  If every element of array `b` is unique, the loop can safely be made parallel. To achieve automatic parallelism in such cases, use the `#pragma permutation(b)` directive, as discussed in Section 15.4.5.

- Arrays with unanalyzable subscripts. APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain the following elements:

– Literal constants (`1`, `2`, `3`, ...)

– Variables (`i`, `j`, `k`, ...)

– The product of a literal constant and a variable, such as `n*5` or `k*32`

– A sum or difference of any combination of the first three items, such as `n*21+k-251`

  In the following example, APO cannot analyze the division operator (`/`) in the array subscript and cannot reorder the loop:

```
for(i=0; i<n; i+=2)  
    a[i/2] = ...;
```

- Unknown information. In the following example there may be hidden knowledge about the relationship between variables `m` and `n`:

```
for(i=0; i<n; i++)  
    a[i] = a[i+m];
```

  The loop can be run in parallel if `m > n` because the array reference does not overlap. However, APO does not know the value of the variables and therefore cannot make the loop parallel. You can use the `#pragma concurrent` directive to have APO automatically parallelize this loop. For more information on this directive, see Section 15.4.2.

15.5.1.5 Loops Containing Local Variables

When parallelizing a loop, APO often localizes (privatizes) temporary scalar and array variables by giving each processor its own non−shared copy of them. In the following example, array `tmp` is used for local scratch space:

```
for(i=0; i<n; i++) {  
    for(j=0; j<n; j++)  
      tmp[j] = i+j;  
    for(j=0; j<n; j++)  
      a[i][j] = a[i][j] + tmp[j];  
}
```

To successfully parallelize the outer loop (`i`), APO must give each processor a distinct, private copy of array `tmp`. In this example, it is able to localize `tmp` and, thereby, to parallelize the loop.

APO cannot parallelize a loop when a conditionally assigned temporary variable might be used outside of the loop, as in the following example:

```
extern int t;  
for(i=0; i<n; i++) {  
    if(b[i]) {  
        t = ...;  
        a[i] += t;  
    }  
}  
s2();
```

If the loop were to be run in parallel, a problem would arise if the value of `t` were used inside function `s2()` because it is not known which processor's private copy of `t` should be used by `s2()`. If `t` were not conditionally assigned, the processor that executed iteration `i == n-1` would be used. Because `t` is conditionally assigned, APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable `t` is localized. If the value of `t` is not used outside the loop, replace `t` with a local variable. Unless `t` is a local variable, APO assumes that `s2()` might use it.

15.5.2 Constructs That Reduce Performance of Parallelized Code

APO parallelizes a loop by distributing its iterations among the available processors. Loop nesting, loops with low trip counts, and other program characteristics can affect the efficiency of APO. The following subsections describe the effect that these and other programming constructs can have on APO's ability to parallelize:

· Section 15.5.2.1, describes parallelizing nested loops.

· Section 15.5.2.2, describes parallelizing loops with small or indeterminate trip counts.

· Section 15.5.2.3, describes parallelizing loops that exhibit poor data locality.

15.5.2.1 Parallelizing Nested Loops

APO can parallelize only one loop in a loop nest. In these cases, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from that fact that more processors end up processing larger sections of the program. This saves synchronization and other overhead costs.

Example 1. Consider the following simple loop nest:

```
for(i=0; i<n; i++)  
    for(j=0; j<m; j++)  
        for(k=0; k<l; k++)  
            ...
```

When parallelizing nested loops `i`, `j`, and `k`, APO parallelizes only one of the loops. Effective loop nest parallelization depends on the loop that APO chooses, but it is possible for APO to choose an inferior loop to be parallelized. APO may attempt to interchange loops to make a more promising one the outermost. If the outermost loop attempt fails, APO attempts to parallelize an inner loop.

Section 15.2.1, describes *`file`*`.list`. This output file contains information that tells you which loop in a nest was parallelized. Because of the potential for improved performance, it is useful for you to modify your code so that the outermost loop is the one parallelized.

For every loop that is parallelized, APO generates a test to determine whether the loop is being called from within either another parallel loop or from within a parallel region. In some cases, you can minimize the extra testing that APO must perform by inserting directives into your code to inhibit parallelization testing. The following example demonstrates this:

Example 2:

```
void sub(int i, int n) {  
   int j;  
   #pragma serial  
```

```
   for(j=0; j<n; j++) {  
       ...  
   }  
}  
void caller(int n) {  
    int i;  
    #pragma concurrent call  
    for(i=0; i<n; i++) {  
        sub(i, n);  
    }  
}
```

Assume that `sub()` is called only from within `caller()`. The loop in `caller()` is parallelized, so the loop in `sub()` can never be run in parallel. In this case, the test is avoided by using the `#pragma serial` directive, as shown, to force the sequential execution of the loop.

For more information on this compiler directive, see Section 15.4.3.

15.5.2.2 Parallelizing Loops With Small Or Indeterminate Trip Counts

The *trip count* is the number of times a loop is executed. Loops with large trip counts are the best candidates for parallelization. The following paragraphs show how to modify your program if your program contains loops with small trip counts or loops with indeterminate trip counts:

- Loops with small trip counts generally run faster when they are not parallelized. Consider the following loop nest:

  ```
  #pragma prefer serial  
  for(i=0; i<m; i++) {  
    for(j=0; j<n; j++) {  
      ...  
    }  
  }
  ```

  Without the directive, APO would attempt to parallelize loop `i` because it is outermost. If `m` is very small, it would be better to interchange the loops and make loop `j` outermost, so that it would be parallelized. If that is not possible, and if APO cannot determine that `m` is small, you can use a `#pragma prefer serial` directive, as shown, to indicate to APO that it is better to parallelize loop `j`.

- Loops with large trip counts run faster if they are unconditionally parallelized. Consider the following loop:

  ```
  #pragma prefer concurrent  
  for(j=0; j<n; j++)  
    ...
  ```

  Without the directive, if the trip count is not known (and sometimes even if it is), APO parallelizes the loop conditionally. It generates code for both a parallel and a sequential version of the loop, plus code to select the version to use, based on the trip count, the code inside the loop's body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that run−time environment.

  You can avoid the overhead of conditional parallelization by using the `#pragma prefer concurrent` directive, as shown, to indicate to APO that only the parallel version of the loop should be generated.

15.5.2.3 Parallelizing Loops With Poor Data Locality

Computer memory has a hierarchical organization. Higher up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is further down in the hierarchy. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor's cache, a program's performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*, which means that if a memory location is referenced, it is probable that it or a nearby location will be referenced in the near future.

Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

The following examples show the effect of locality on parallelization. Assume that the loops are to be parallelized and that there are *p* processors.

Example 1. Distribution of Iterations.

```
for(i=0; i<n; i++) {  
    ...a[i]...  
}  
for(i=n-1; i>=0; i--) {  
    ...a[i]...  
}
```

In the first loop, the first processor accesses the first `n/p` elements of `a`; the second processor accesses the next `n/p` elements; and so on. In the second loop, the distribution of iterations is reversed. That is, the first processor accesses the last `n/p` elements of `a`, and so on. Most elements are not in the cache of the processor needing them during the second loop. This code fragment would run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2. Two Nests in Sequence.

```
for(i=0; i<n; i++)  
    for(j=0; j<n; j++)  
      a[i][j] = b[j][i] + ...;  
  
for(i=0; i<n; i++)  
    for(j=0; j<n; j++)  
      b[i][j] = a[j][i] + ...;  
```

In example 2, APO may parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses the first `n/p` rows of `a` and the first `n/p` columns of `b`. In the second nest, the first processor accesses the first `n/p` columns of `a` and the first `N/p` rows of `B`. This example runs much more efficiently if you parallelize the `i` loop in one nest and the `j` loop in the other. You can instruct APO to do this by inserting a `#pragma prefer serial` directive just prior to the `i` loop that contains the `j` loop that you want to be parallelized.