Chapter 2

Understanding Incomplete Optimization

This chapter discusses programming practices that prevent your code from being fully optimized by the Auto−Parallelizing Option. The MIPSpro APO cannot always parallelize programs effectively. Sometimes this is unavoidable, but at other times you can help it improve its effectiveness. There are three general categories of incomplete optimization:

- The MIPSpro APO does not detect that a loop is safe to parallelize, as described in "Failing to Parallelize Safe Loops".
- The MIPSpro APO parallelizes the wrong loop, usually in a nest, as covered in "Parallelizing the Wrong Loop".
- The MIPSpro APO unnecessarily parallelizes a loop, as discussed in "Incurring Unnecessary Parallelization Overhead".

Failing to Parallelize Safe Loops

A program's performance may be severely constrained by the Auto−Parallelizing Option's not recognizing that a loop is safe to parallelize. A loop is safe if there isn't a data dependence, such as a variable being assigned in one iteration of a loop and used in another. The MIPSpro APO analyzes every loop in a sequential program. If it cannot prove a loop is safe, it does not parallelize that loop.

The MIPSpro APO often does not parallelize loops containing any of the following constructs described in this section:

- "Function Calls in Loops"
- "GO TO Statements in Loops"
- "Problematic Array Subscripts"
- "Conditionally Assigned Temporary Nonlocal Variables in Loops"
- "Unanalyzable Pointer Usage in C and C++"

However, in many instances such loops can be automatically parallelized after minor changes. Reviewing your program's *.l* file, described in "About the .l File", can show you if any of these constructs are in your code.

Function Calls in Loops

By default, the Auto−Parallelizing Option does not parallelize a loop that contains a function call because the function in one iteration of the loop may modify or depend on data in other iterations. However, a couple of tools can help with this problem.

- Interprocedural analysis (IPA), specified by the **−IPA** command−line option, can provide the MIPSpro APO with enough information to parallelize some loops containing subroutine calls by inlining those calls. For more information on IPA, see "Interprocedural Analysis" and the *MIPSpro Compiling and Performance Tuning Guide.*
- As discussed in "C*$* ASSERT CONCURRENT CALL and #pragma concurrent call", you can

tell the MIPSpro APO to ignore the dependences of function calls when analyzing the specified loops by using the following:

- the assertion **C*$* ASSERT CONCURRENT CALL** for Auto−Parallelizing Fortran 77 and Auto−Parallelizing Fortran 90
- the **#pragma concurrent call** for Auto−Parallelizing C and Auto−Parallelizing C++

GO TO Statements in Loops

GO TO statements are unstructured control flows. The Auto−Parallelizing Option converts most unstructured control flows in loops into structured flows that can be parallelized. However, **GO TO** statements in loops can still cause two problems:

- Unstructured control flows the MIPSpro APO cannot restructure. 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.

Problematic Array Subscripts

There are three cases where array subscripts are too complicated to permit parallelization:

- The subscripts are indirect array references.
- The subscripts are unanalyzable.
- The subscripts rely on hidden knowledge.

Indirect Array References

The MIPSpro APO is not able to analyze indirect array references. Consider the following Fortran example:

```
DO I = 1, N
  A(IB(I)) = ...
END DO
```

This loop cannot be run safely in parallel if the indirect reference *IB(I)* is equal to the same value for different iterations of *I*. If every element of array *IB* is unique, the loop can safely be made parallel. To achieve parallelism in such cases, you can use either manual or automatic methods to achieve parallelism. For automatic parallelization, the **C*$* ASSERT PERMUTATION** assertion, discussed in "C*$* ASSERT PERMUTATION and #pragma permutation", is appropriate.

Unanalyzable Subscripts

The MIPSpro APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain four 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 case, the MIPSpro APO cannot analyze the division operator (/) in the array subscript and cannot reorder the loop:

```
DO I = 2, N, 2
  A(I/2) = ...
END DO
```

Hidden Knowledge

In the following example there may be hidden knowledge about the relationship between the variables *M* and *N*:

```
DO I = 1, N
  A(I) = A(I+M)
END DO
```

The loop can be run in parallel if *M* > *N*, because the array reference does not overlap. However, the MIPSpro APO does not know the value of the variables and therefore cannot make the loop parallel. Using the **C*$* ASSERT DO (CONCURRENT)** assertion, explained in "C*$* ASSERT DO (CONCURRENT) and #pragma concurrent", lets the MIPSpro APO parallelize this loop. You can also use manual parallelization.

Conditionally Assigned Temporary Nonlocal Variables in Loops

When parallelizing a loop, the Auto−Parallelizing Option often localizes (privatizes) temporary scalar and array variables by giving each processor its own non−shared copy of them. Consider the following example:

```
DO I = 1, N
  DO J = 1, N
    TMP(J) = ...
  END DO
  DO J = 1, N
    A(J,I) = A(J,I) + TMP(J)
  END DO
END DO
```

The array *TMP* is used for local scratch space. To successfully parallelize the outer (*I*) loop, the MIPSpro APO must give each processor a distinct, private *TMP* array. In this example, it is able to localize *TMP* and, thereby, to parallelize the loop.

The MIPSpro APO runs into trouble when a conditionally assigned temporary variable might be used outside of the loop, as in the following example.

```
SUBROUTINE S1(A, B)
  COMMON T
  ...
  DO I = 1, N
```

```
    IF (B(I)) THEN
      T = ...
      A(I) = A(I) + T
    END IF
  END DO
  CALL S2()
END
```

If the loop were to be run in parallel, a problem would arise if the value of *T* were used inside subroutine *S2()*. Which processor's private copy of *T* should *S2()* use? If *T* were not conditionally assigned, the answer is the processor that executed iteration *N*. Because *T* is conditionally assigned, the MIPSpro 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, the MIPSpro APO must assume that *S2()* might use it.

Unanalyzable Pointer Usage in C and C++

The C and C++ languages have features that make them more difficult than Fortran to automatically parallelize. These features are often related to the use of pointers, such as implementing multidimensional arrays as pointer accesses and not as true arrays. The following practices involving pointers interfere with the Auto−Parallelizing Option's effectiveness:

- arbitrary pointer dereferences
- arrays of arrays
- loops bounded by pointer comparisons
- aliased parameter information

Arbitrary Pointer Dereferences

The MIPSpro APO does not analyze arbitrary pointer dereferences. It cannot parallelize a list traversal, for example. The only pointers it analyzes are array references and pointer dereferences that can be converted into array references.

Arrays of Arrays

Multidimensional arrays are sometimes implemented as arrays of arrays. Consider this example:

```
double **p;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    p[i][j] = ...
```

If *p* is a true multidimensional array, the outer loop can be run safely in parallel. However, if two of the array pointers, *p[2]* and *p[3]* for example, reference the same array, the loop must not be run in parallel. Although this duplicate reference is unlikely, the MIPSpro APO cannot prove it doesn't exist. You can avoid this problem by always using variable length arrays. To parallelize the code fragment above, rewrite it as follows:

```
double p[n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    p[i][j] = ...
```

Note: Although ANSI C did not allow variable−sized multidimensional arrays when this document was written, MIPSpro 7.2 C and C++ implemented this proposal. See the *C Language Reference Manual* for details.

Loops Bounded by Pointer Comparisons

The MIPSpro APO parallelizes only those loops for which the number of iterations can be exactly determined. In Fortran programs this is rarely a problem, but in C and C++, issues related to overflow and unsigned arithmetic can come to play. For example, pointers are unsigned data types and cannot have negative values. One consequence is that loops to be parallelized should not be bounded by pointer comparisons such as this:

```
int *pl, *pu;
...
for (int *p = pl; p < pu; p++)
```

In this case, it appears that the number of loop iterations is either the quantity *pu−p* or zero, whichever is greater. However, because they are unsigned types, *pu−p* must always be greater than zero, even if *p* > *pu*. Compiling this loop results in a *.l* file entry stating that the bound cannot be standardized. To avoid this result, restructure the loop to be of this form:

```
int lb, ub;
...
for (int i = lb; i < ub; i++)
```

Aliased Parameter Information

Perhaps the most frequent impediment to parallelizing C and C++ programs are aliases. Although Fortran guarantees that multiple parameters to a subroutine are not aliased to each other, C and C++ do not. Consider the following example:

```
void sub(double *a, double *b, n) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

This loop can be parallelized only if arrays *a* and *b* do not overlap. With the **__restrict** (two underscores) type qualifier keyword, available with the MIPSpro 7.2 C and C++ compilers, you can inform the MIPSpro APO that the arrays do not overlap. This keyword tells the compiler to assume that dereferencing the qualified pointer is the only way the program can access the memory pointed to by that pointer. Thus loads and stores through that pointer are not aliased with loads and stores through any other pointers.

The next example shows the **__restrict** type qualifier used to guarantee that *a* and *b* provide exclusive access to their arrays. This assurance permits the MIPSpro APO to proceed with the parallelization.

```
void sub(double * __restrict a, double __restrict *b, n) {
```

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

There are two more ways of dealing with aliased parameter information:

- The keyword **restrict** behaves similarly to **__restrict**, but might clash with other meanings of **restrict** in older programs. To enable the **restrict** keyword, you must specify **−LANG:restrict** in the command line. The type qualifier **__restrict** does not require that **−LANG:restrict** be specified.
- The **−OPT:alias=***restrict* compiler option, covered in "Miscellaneous Optimization Options", guarantees no aliasing by pointers. It is less desirable to use because it is global in scope and may have unforeseen effects.

Parallelizing the Wrong Loop

The Auto−Parallelizing Option parallelizes a loop by distributing its iterations among the available processors. When parallelizing nested loops, such as *I*, *J*, and *K* in the example below, the MIPSpro APO distributes only one of the loops:

```
DO I = 1, L
  ...
  DO J = 1, M
    ...
    DO K = 1, N
      ...
```

Because of this restriction, the effectiveness of the parallelization of the nest depends on the loop that the MIPSpro APO chooses. In fact, the loop the MIPSpro APO parallelizes may be an inferior choice for any of three reasons:

- It is an inner loop, as discussed in "Inner Loops".
- It has a small trip count, as covered in "Small Trip Counts".
- It exhibits poor data locality, as described in "Poor Data Locality".

The MIPSpro APO's heuristic methods are designed to avoid these problems. The next three sections show you how to increase the effectiveness of these methods.

Inner Loops

With nested loops, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from more processors processing larger sections of the program, saving synchronization and other overhead costs. Therefore, the Auto−Parallelizing Option tries to parallelize the outermost loop, after possibly interchanging loops to make a more promising one outermost. If the outermost loop attempt fails, the MIPSpro APO parallelizes an inner loop if possible.

Checking the *.l* file, described in "About the .l File", tells you which loop in a nest was parallelized. If the loop was not the outermost, the reason is probably one of those mentioned in "Failing to

Parallelize Safe Loops". Because of the potential for improved performance, it is probably advantageous for you to modify your code so that the outermost loop is the one parallelized.

Small Trip Counts

The *trip count* is the number of times a loop is executed. As discussed in "Incurring Unnecessary Parallelization Overhead," loops with small trip counts generally run faster when they are not parallelized. Consider how this fact affects this Fortran example:

```
DO I = 1, M
  DO J = 1, N
```

The Auto−Parallelizing Option may try to parallelize the*I* loop because it is outermost. If *M* is very small, it would be better to interchange the loops and make the *J* loop outermost before parallelization. Because the MIPSpro APO often cannot know that *M* is small, you can do one of the following:

- Use one of the *C*$* ASSERT* **DO PREFER** assertions to tell the MIPSpro APO that it is better to parallelize the *J* loop. They are discussed in
  - "C*$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent"
  - "C*$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial"
- Use the manual parallelization directives contained in the "Manual Parallelization References".

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 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*: If a memory location is referenced, there is a good chance 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.

To see the effect of locality on parallelization, consider Example 2−1 and Example 2−2 In each case assume that the loops are to be parallelized and that there are *p* processors.

Example 2−1 Distribution of Iterations

```
DO I = 1, N
  ...A(I)
END DO
DO I = N, 1, -1
  ...A(I)...
END DO
```

In the first loop of Example 2−1, 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: 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 example should run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2−2 Two Nests in Sequence

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = B(J,I) + ...
  END DO
END DO

DO I = 1, N
  DO J = 1, N
    B(I,J) = A(J,I) + ...
  END DO
END DO
```

In Example 2−2, the Auto−Parallelizing Option may choose to 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 the MIPSpro APO to do this with the *C*$* ASSERT* **DO PREFER** assertions described in

- "C*$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent"
- "C*$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial"

Incurring Unnecessary Parallelization Overhead

Parallelization of loops does not come without cost. There is overhead associated with distributing the iterations among the processors and synchronizing the results of the parallel computations. There can also be memory−related costs, such as the cache interference that occurs when different processors try to access the same data. One consequence of this overhead is that not all loops should be parallelized. As discussed in "Small Trip Counts", loops that have a small number of iterations run faster sequentially than in parallel. Two other cases of unnecessary overhead involve

- unknown trip counts
- nested parallelism

Unknown Trip Counts

If the trip count is not known (and sometimes even if it is), the Auto−Parallelizing Option parallelizes the loop conditionally. It generates code for both a parallel and a sequential version. By generating two versions, the MIPSpro APO can avoid running in parallel a loop that turns out to have a small trip count. The MIPSpro APO chooses 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.

Note: You can control this cost estimate by using the option **−LNO:parallel_overhead=***n*. The default value of *n* varies on different systems, but a typical value is several thousand machine cycles.

Having a sequential and parallel version of the loop incurs the run−time overhead of choosing between them. You can avoid this overhead by using the **prefer** pragmas or *C*$* ASSERT* **DO PREFER** assertions, discussed in

- "C*$* ASSERT DO PREFER (CONCURRENT) and #pragma prefer concurrent"
- "C*$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial"

These compiler directives ensure that the MIPSpro APO knows in advance whether or not to parallelize the loop.

Nested Parallelism

Nested parallelism is not supported by the Auto−Parallelizing Option. Thus, for every loop that could be parallelized, the MIPSpro APO must generate a test that determines if the loop is being called from within either another parallel loop or a parallel region. While this check is not very expensive, it can add overhead. The following example demonstrates nested parallelism:

```
SUBROUTINE CALLER
  DO I = 1, N
    CALL SUB
  END DO
  ...
END
SUBROUTINE SUB
  ...
  DO I = 1, N
    ...
  END DO
END
```

If the loop inside *CALLER()* is parallelized, the loop inside *SUB()* cannot be run in parallel when *CALLER()* calls *SUB()*. In this case, the test can be avoided. If *SUB()* is always called from *CALLER()*, you can use the **C*$* ASSERT DO (SERIAL)** or the **C*$* ASSERT DO PREFER (SERIAL)** assertion to force the sequential execution of the loop in *SUB()*. For more information on these compiler directives, see

- "C*$* ASSERT DO (SERIAL) and #pragma serial"
- "C*$* ASSERT DO PREFER (SERIAL) and #pragma prefer serial"