Sun MP C Compiler
Technical White Paper

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Sun MP C Compiler (Technical White Paper)

Vinod Grover
Douglas Walls

November 11, 1996

1. Introduction

Sun MP C is an extended ANSI C compiler that can compile code to run on SPARC shared memory multiprocessor machines. The compiled code, may run in parallel using the multiple processors on the system.

The MP C compiler is integrated with the Sun optimizing compilers. The Sun optimizing compilers already have capabilities to exploit instruction level parallelism. These capabilities are enhanced by the analysis performed by the MP compilers.

The MP C compiler also supports extensions that allow users to control parallelism explicitly.

The MP C compiler generates parallel code for those loops that it determines are safe to parallelize. Typically, these loops have iterations that are independent of each other. For such loops, it does not matter in what order the iterations are executed or if they are executed in parallel. Many, not all, vector loops fall into this category.

The MP C compiler is able to perform extensive automatic restructuring of user code. These automatic transformations expose higher degrees of loop level parallelization. These transformations include: loop interchange, loop fusion, loop distribution and software pipelining.

MP C provides explicit and automatic capabilities for parallelizing loops in ANSI C. This product is chiefly intended to be useful to users who write loop intensive codes that are common in certain numeric and scientific codes. Some vector codes fall in this category. These codes typically spend significant amount of time in loops that process data stored in arrays. The loops to which the parallelizing capability is applicable are further restricted to Fortran style DO loops. This means that the loop in question must have an index variable that changes with a uniform step.

Due to the potential of aliasing in C, it is especially hard to determine the safety of parallelization. To make the compiler's job easier, MP C offers pragmas and pointer qualifications to provide aliasing information to the compiler.

MP C does not offer any capabilities to automatically parallelize arbitrary regions of code or other C statements. Of course the user can use user-level threads to parallelize applications instead of MP C.

1.1. Caveats

Often dramatic speedups can be obtained automatically, by simply compiling the program with the appropriate compilation flags. Often these dramatic speedups will fail to scale up as the number of processors is increased. But more often, it would not be the case that the program will speedup right away. There may be cases when the program will even slowdown. In such cases tuning and measurement of the code would be needed. Based on this the user could use directives or pragmas, or even restructure the code.

For certain applications attempts at automatic parallelization, explicit use of pragmas, restructuring or tuning may be well worth the effort.

This document attempts to educate the user about what factors can benefit parallelization and what factors can hurt it. Hopefully the user will be able to decide for himself if the effort is worth it.

1.2. Example of Use.

The following example illustrates the use of MP C and how parallel execution can be controlled. In order to enable parallelization of the target program the -xautopar option can be used as follows:

```
%
% cc -fast -xO4 -xautopar example.c -o example
%
```

This results in an executable called example to be generated, which can be executed normally. If multiprocessor execution is desired then an environment variable needs to be set:

```
%
% setenv PARALLEL 2
%
```

This will enable the execution of the program on two threads. If the target machine has multiple processors, then this will allow the threads to map to independent processors.

```
%
% example
%
```

Running the program will now lead to creation of two threads that will execute the parallelized portions of the program.

2. Overview of Parallelization Concepts.

2.1. Data Dependence and Interference.

The MP C compiler performs analysis on loops in programs to determine if it is safe to execute different iterations of the loops in parallel. The purpose of this analysis is to determine if any two iterations of the loop could interfere with each other. Typically this happens if one iteration of a variable could read a variable while another iteration is writing the very same variable. Consider the following program fragment:

example 1.

```
for (i=1; i < 1000; i++) {
   sum = sum + a[i]; /* S1 */
}
```

In this example, any two successive iterations, i and i+1, will write and read the same variable sum. Therefore, in order for these two iterations to execute in parallel some form of locking on the variable would be required. Otherwise it is not safe to allow the two iterations to execute in parallel.

However, the use of locks imposes overhead that might slowdown the program. The MP C compiler will not ordinarily parallelize the above loop.

In the above example there is a data dependence between two iterations of the loop.

Consider another example.

example 2.

```
for (i=1; i < 1000; i++) {
   a[i] = 2 * a[i]; /* S1 */
}
```

In this case each iteration of the loop references a different array element. Therefore different iterations of the loop can be executed in any order. They may be executed in parallel without any locks because no two data elements of different iterations can possibly interfere.

The analysis performed by the compiler to determine if two different iterations of a loop could reference the same variable is called “data dependence analysis”.

Data dependences prevent loop parallelization if one of the references writes to the variable.

The dependence analysis performed by the compiler can have 3 outcomes:

1) There is a dependence: in this case it is not safe to execute the loop in parallel. Example 1 above illustrates this.
2) There is no dependence in this loop, the loop may safely be executed in parallel using an arbitrary number of processors. Example 2 above illustrates this case
3) The dependence cannot be determined: in this case it is not clear if there is a data dependence or not. So the compiler, for safety, assumes that there might be a dependence that prevents parallel execution of the loop and will not parallelize the loop.

example 3.

```
for (i=1; i < 1000; i++) {
    a[b[i]] = 2 * a[i];
}
```

In this example, whether or not two iterations of the above loop write to the same element of array ‘a’ depends on whether or not array b contains duplicate elements. Unless the compiler can determine this fact, it will assume that there is a dependence and will not parallelize the loop.

2.2. Parallel Execution Model

The parallel execution of loops is performed by Solaris threads. The thread starting the initial execution of the program is called the “master thread”. At program start-up the master thread creates multiple “slave” threads. At the end of the program all the slave threads are terminated. Slave thread creation is performed exactly once to minimize the overhead.

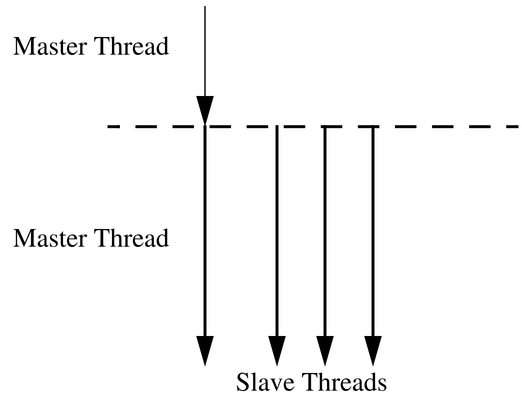

FIGURE 1. Master and Slave Threads

After start-up, master thread starts the execution of the program while slave threads wait idly. When the master thread encounters a parallel loop, different iterations of the loop are distributed among the slave and master threads which start the execution of the loop. After each thread finishes execution of its chunk it synchronizes with the remaining threads. This synchronization point is called a “barrier”. The master thread cannot continue executing the remainder of the program until all the threads have finished their work and reached the barrier. The slave threads go into a wait state after the barrier waiting for more parallel work, and the master thread continues execution of the program.

In this process several overheads can occur: 1) the overhead of synchronization and work distribution, and 2) the overhead of barrier synchronization. In general there may be some parallel loops for which the amount of useful work performed is not enough to overcome the overhead. For such loops there may be appreciable slowdown.

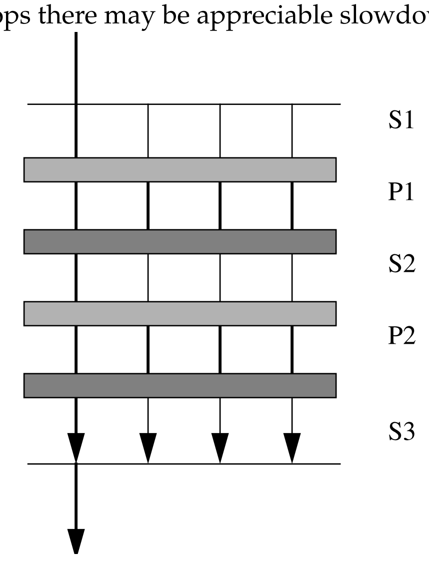

FIGURE 2. Parallel Execution of a Sample Program

2.3. Private Variables.

2.3.1. Private Scalars

There are some data dependences for which the compiler may still be able to parallelize a loop. Consider the following example.

example 4.

```
      for (i=1; i < 1000; i++) {
          t = 2 * a[i]; /* S1 */
          b[i] = t;     /* S2 */
    }
```

In this example, assuming that arrays a and b are non-overlapping arrays, there appears to be a data dependence in any two iterations due to the variable t. If we observe iterations 1 and 2 we will see the following statements being executed:

```
      t = 2*a[1];  /* 1 */
      b[1] = t;     /* 2 */
      t = 2*a[2];  /* 3 */
      b[2] = t;     /* 4 */
```

Because statements 1 and 3 modify the variable t, one cannot execute these two in parallel. But we know that for each iteration the value of t that is used is always computed in the same iteration, we can use a separate copy of t for each iteration. This eliminates the interference between different iterations due to such variables. In effect, we have made variable t as a private variable for each thread executing that iteration. This can be illustrated as follows:

example 5.

```
      for (i=1; i < 1000; i++) {
          pt[i] = 2 * a[i]; /* S1 */
         b[i] = pt[i];      /* S2 */
     }
```

This is essentially the same example as example 3, but each scalar variable reference 't' is now replaced by an array reference 'pt'. Each iteration now uses a different element of pt, and this results in eliminating any data dependencies between any two iterations. Of course one problem with this illustration is that this may lead to an extra large array. In practice, the compiler only allocates one copy of the variable for each thread that will participate in the execution of the loop. Each such variable is, in effect, private to the thread.

2.3.2. Private Arrays.

In previous section we described how privatization of a scalar variable can create opportunities for parallel execution of loops. The same can be done for array variable as the following example illustrates:

example 6.

```
      for (i=1; i < 1000; i++) {
         for (j=1; j < 1000; j++) {
           x[j] = 2 * a[i]; /* S1 */
           b[i][j] = x[j];   /* S2 */
       }
    }
```

In this example, different iterations of the outer loop modify same elements of array x, and thus the outer loop cannot be parallelized. However, if each thread executing the outer loop iterations had a private copy of the entire array x, then there would be no interference between any two iterations of the outer loop. This is illustrated as follows:

example 7.

```
for (i=1; i < 1000; i++) {
   for (j=1; j < 1000; j++) {
      px[i][j] = 2 * a[i]; /* S1 */
      b[i][j] = px[i][j]; /* S2 */
  }
}
```

As in the case of private scalars, it is not necessary to expand the array for all the iterations, but only up to the number of threads executing in the systems. This is done automatically by the compiler by allocating one copy of the original array in the private space of each thread.

2.4. Storeback

Privatization of variables can be very useful for improving the parallelism in the program. However, if the private variable is referenced outside the loop then the compiler needs to assure that it has the right value.

example 8.

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i]; /* S1 */
    b[i] = t;     /* S2 */
}
 x = t;       /* S3 */
```

In this example the value of t referenced in statement S3 is the final value of t computed by the loop. After the variable t has been privatized and the loop has finished executing, the right value of t needs to be stored back into the original variable. This is called “storeback”. This is done by copying the value of t on the final iteration back to the original location of variable t. In many cases the compiler can do this automatically. But there are situations where the last value cannot be computed so easily.

example 9.

```
for (i=1; i < 1000; i++) {
    if (c[i] > x[i] ) {/* C1 */
      t = 2 * a[i];     /* S1 */
      b[i] = t;          /* S2 */
   }
}
 x = t*t;              /* S3 */
```

For correct execution, the value of t in statement S3 is not, in general, the value of t on the final iteration of the loop. It is in fact the last iteration for which the condition C1 is true. Computing the final value of t is quite hard in the general cases. In cases like this the compiler will not parallelize the loop.

2.5. Reduction Variables

There are cases when there is a real dependence between iterations of a loop and the variables causing the dependence cannot simply be made private. This can arise, for example, when values are being accumulated from one iteration to the next.

example 10.

```
 for (i=1; i < 1000; i++) {
       sum += a[i]*b[i];   /* S1 */
}
```

In this example the loop computes the vector product of two arrays into a common variable called sum. This loop cannot be parallelized in a simple manner. The compiler can take advantage of the associative nature of the computation in statement S1 and allocate a private variable called psum[i] for each thread. Each copy of the variable psum[i] is initialized to 0. Each thread computes the its own partial sum in its own copy of the variable psum[i]. Before crossing the barrier, all the partial sums are added onto the original variable sum. In this example the variable sum is called a reduction variable because it computes a sum-reduction. However, one danger of promoting scalar variables to reduction variables is that the manner in which roundoffs are accumulated can change the final value of sum.

Compiler will do this transformation only if the user specifically gives permission for it.

3. Speedups

If the portion of program where significant portion of time is spent does not get parallelized, then no speedup will occur. This is basically a consequence of Amdahl's Law. For example, if a loop that accounts for only 5% of execution of a program is parallelized, then the overall speedup will be limited to 5%. However, in practice, depending on the size of the workload and parallel execution overheads there may not be any speedups.

So as a general rule, larger the fraction of program execution that is parallelized, larger is the likelihood of a speedup.

Each parallel loop incurs a small overhead during start-up and shutdown. The start overhead includes the cost of work distribution, and the shutdown overhead includes the cost of the barrier synchronization. If the total amount of work performed by the loop is not big enough then no speedup will occur. In fact the loop might even slow down. So if a large amount of program execution is accounted by a large number of short parallel loops, then the whole program may slow down instead of speeding up.

The compiler performs several loop transformations that try to increase the granularity of the loops. Some of these transformations are: loop interchange, and loop fusion.

So in general, if the amount of parallelism in a program is small or is fragmented among small parallel regions then speedup will be less.

Often scaling up a problem size improves the fraction of parallelism in a program. For example, consider a problem that consists of two parts: a quadratic part that is sequential, and a cubic part that is parallelizable. For this problem the parallel part of the workload grows faster than the sequential part. So at some point the problem will speedup nicely, unless it runs into resource limitations.

Many users would have to do some tuning, experimentation with directives, problem sizes and program restructuring in order to achieve benefits from parallel C.

3.1. Amdahl's Law

Fixed problem-size speedup is generally governed by Amdahl's law. Amdahl's Law simply says that the amount of parallel speedup in a given problem is limited by the sequential portion of the problem.

$$\frac{1}{S} = F + \frac{(1-F)}{P} \qquad \text{(EQ 1)}$$

The above equation describes the speedup of a problem where F is the fraction of time spent in sequential region, and the remaining fraction of the time is spent uniformly among P processors. As one can see that if the second term of the equation drops to zero the total speedup is bounded by the first term, which remains fixed.

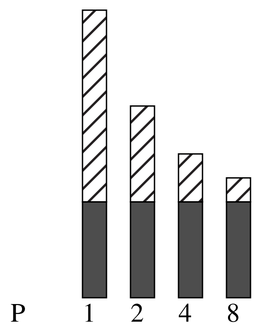

FIGURE 3. Fixed Problem Speedups

The above diagram illustrates this concept diagrammatically. The darkly shaded potion represents the sequential part of the program, and remains constant for 1, 2, 4, and 8 processors, while the lightly shaded portion represents the parallel portion of the program that can be divided uniformly among arbitrary number of processors.

In reality, however, there may be overheads incurred due to communication and distribution of work to multiple processors. These overheads may or may not be fixed for arbitrary number of processors used.

Figure 4. Illustrates the ideal speedups for a program continuing 0%, 2%, 5%, and 10% sequential portions. Here no overhead is assumed.

3.1.1. Overheads

Once the overheads are incorporated in the model the speedup curves change dramatically. Just for the purposes of illustration we assume overheads to consist of two parts: a fixed part which is independent of the number of processors, and a non-fixed part that grows quadratically with the number of the processors used. i.e.:

$$\frac{1}{S} = \frac{1}{F + \left(1 - \frac{F}{P}\right) + K_1 + K_2 P^2} \quad \text{(EQ 2)}$$

In this equation K1 and K2 are some fixed factors. Under these assumptions the speedup curve is shown in Figure 5.

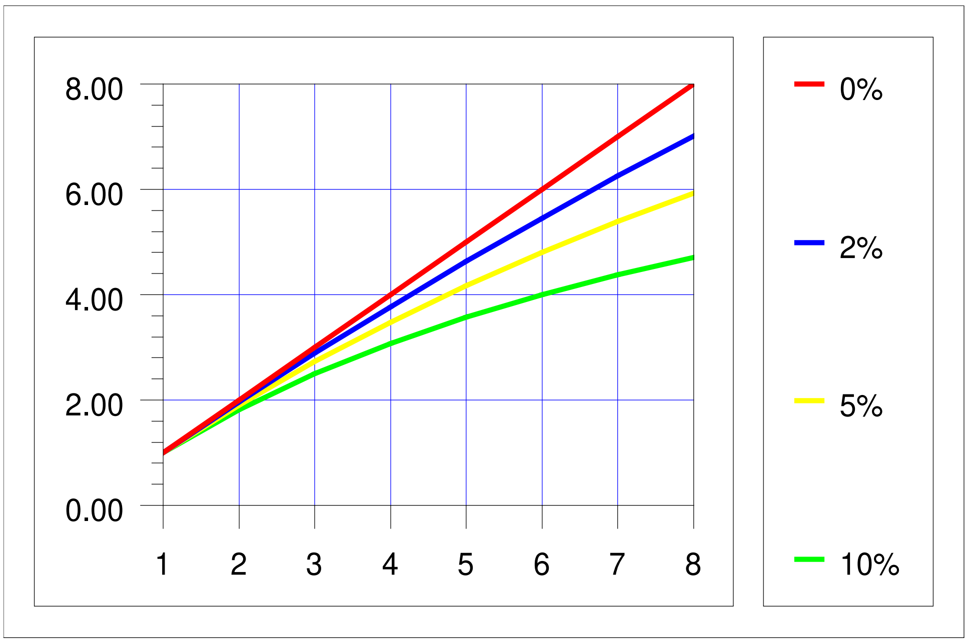

FIGURE 4. Amdahl’s Law Speedup Curve

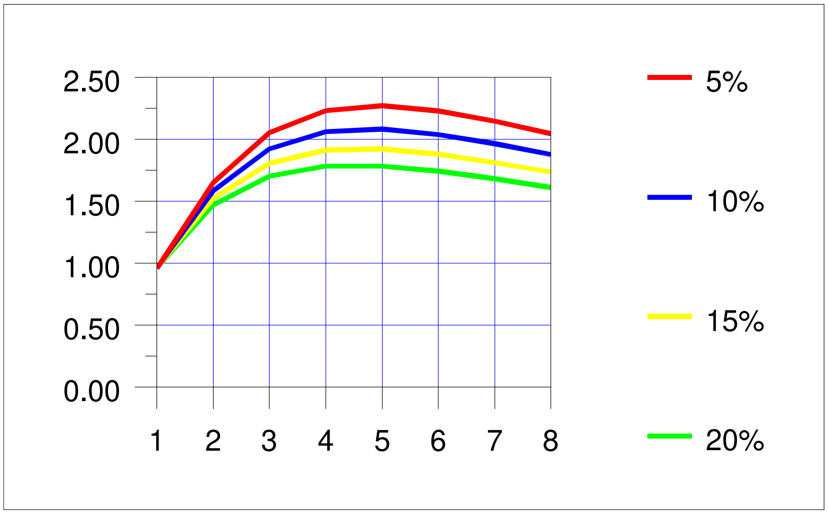

FIGURE 5. Speedup Curve with Overheads

It is interesting to note that in this case the speedups peak out. After a certain point adding more processors is detrimental to performance.

3.2. Gustafson's Law

Amdahl's Law can be misleading for predicting parallel speedups in real problems. The fraction of time spent in sequential sections of the program sometimes depends on the problem size. That is by scaling the problem size one may improve the chances of speedup. This can best be illustrated by an example.

```
/*
 * initialize the arrays
 */
for (i=0; i < n; i++) {
  for (j=0; j < n; j++) {
     a[i][j] = 0.0;
     b[i][j] = ...
     c[i][j] = ...
  }
}
/*
 * matrix multiply
 */
```

```
for (i=0; i < n; i++) {
   for(j=0; j < n; j++) {
      for (k=0; k < n; k++) {
         a[i][j] = b[i][k]*c[k][j];
      }
   }
}
```

Assuming ideal overhead of zero, and that the only the second loop nest is executed in parallel, it is easy to see that for small problem sizes (i.e. small values of n), the sequential and parallel parts of the program are not so far from each other. But as n grows larger the time spent in the parallel part of the program grows faster than the time spent in the sequential part. For this problem, it is beneficial to increase the number of processors as the problem size increases.

4. Load Balance and Loop Scheduling.

The process of distributing iterations of a parallel loop to multiple threads is called "loop scheduling". In order to maximize the speedup it is important that the work be distributed evenly among the threads while not imposing too much overhead. MP C offers several types of scheduling for different situations.

4.1. Static or Chunk Scheduling.

When the work performed by different iterations of a loop is the same then it is beneficial to divide the work evenly among the different threads on the system. This approach is known as "static" scheduling.

example 11.

```
   for (i=1; i < 1000; i++) {
         sum += a[i]*b[i];   /* S1 */
  }
```

Under static or chunk scheduling, each thread will get the same number of iterations. If there were 4 threads, then in the above example, each thread will get 250 iterations. Provided there are no interruptions and each thread progresses at the same rate, all the threads will complete at the same time.

4.2. Self Scheduling

When the work performed by each iteration varies, static scheduling will in general not achieve good load balance. In static scheduling, each thread grabs the same chunk of iterations. All threads, except the master thread, upon completion of its chunk waits to participate in the next parallel loop execution. The master thread continues execution of the program.

In self scheduling, each thread grabs a different a small chunk of iteration and after completion of its assigned chunk it tries to acquire more chunks from the same loop.

4.3. Guided Self Scheduling

In guided self scheduling (GSS), each thread gets successively smaller number of chunks. In cases where the size of each iteration varies, GSS can help balance the load.

5. Loop Transformations

MP C compiler performs several loop restructuring transformations to help improve the parallelization of loop in programs. Some of these transformations can also improve the single processor execution of loops as well. The transformations performed by the compiler are described below.

5.1. Loop Distribution

Often loops contain a few statements that cannot be executed in parallel and many statements that can be executed in parallel. Loop Distribution attempts to remove the sequential statements into a separate loop and gather the parallelizable statements into a different loop. This is illustrated in the following example:

example <n>

```
        for (i=0; i < n; i++) {
            x[i] = y[i] + z[i]*w[i];                    /* S1 */
            a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;    /* S2 */
            y[i] = z[i] - x[i];                            /* S3 */
    }
```

Assuming that arrays x, y, w, a, and z do not overlap, statements S1 and S3 can be parallelized but statement S2 cannot be. But if the loop were split or distributed into two different loops:

example <n>

```
        /* L1: parallel loop */
         for (i=0; i < n; i++) {
            x[i] = y[i] + z[i]*w[i];             /* S1 */
            y[i] = z[i] - x[i];                     /* S3 */
         }
        /* L2: sequential loop */
         for (i=0; i < n; i++) {
            a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;   /* S2 */
        }
```

After this transformation, loop L1 above does not contain any statements that prevent the parallelization of the loop and may be executed in parallel. The second loop L2, however, still has a non-parallelizable statement from the original loop.

Loop distribution is not always profitable or safe to perform. Compiler performs analysis to determine the safety and profitability of distribution.

5.2. Loop Fusion

When the granularity or the work performed by a loop is small, one may not get big enough performance gain. This is because the parallel loop start-up is too high compared to the loop workload. Loop fusion can be used to combine several loops into a single parallel loop, and thus increase the granularity of the loop. This is easily and safely done when loops with identical trip counts are adjacent to each other.

example <n>

```
        /* L1: short parallel loop */
         for (i=0; i < 100; i++) {
           a[i] = a[i] + b[i]; /* S1 */
        }
        /* L2: another short parallel loop */
         for (i=0; i < 100; i++) {
```

```
        b[i] = a[i] * d[i]; /* S2 */
    }
```

In this example two short parallel loops are next to each other, and can be safely combined as follows:

```
      /* L3: a larger parallel loop */
       for (i=0; i < 100; i++) {
         a[i] = a[i] + b[i]; /* S1 */
         b[i] = a[i] * d[i]; /* S2 */
    }
```

This allows the parallel loop execution overhead to be cut in half. Loop fusion can also help in other ways. For example if the same data is referenced in two loops, then combining them can improve the locality of reference.

Loop fusion is not always safe to perform. If loop fusion creates a data dependence that did not exist before then the fusion may result in incorrect execution.

example <n>

```
       /* L1: short parallel loop */
       for (i=0; i < 100; i++) {
         a[i] = a[i] + b[i]; /* S1 */
    }
      /* L2: a short loop with data dependence */
       for (i=0; i < 100; i++) {
         a[i+1] = a[i] * d[i]; /* S2 */
    }
```

If loop fusion were to be done in the above example, then a data dependence would be created from statement S2 to S1. In effect the value of a[i] in the right hand side of statement S1 would be that computed in statement S2. Which is not what would happen if the loops were not fused.

The compiler performs safety and profitability analysis to determine if loop fusion should be done. Often it can fuse an arbitrary number of loops. Increasing the granularity in this manner can sometimes push it far enough up for the loop to be profitable for parallelization.

5.3. Loop Interchange

It is generally more profitable to parallelize the outermost loop in a nest of loops, since the overheads incurred are small. However, it is not always safe to parallelize the outermost loops due to dependences that might be carried by such loops. This is illustrated in the following example:

example <n>

```
for (i=0; i <n; i++) {
for (j=0; j <n; j++) {
a[j][i+1] = 2.0*a[j][i-1];
}
}
```

In this example, the loop with the index variable i cannot be parallelized, because of a dependency between two successive iterations of the loop. The two loops could be interchanged and the parallel loop (i.e. the j-loop) will now become the outer loop.

```
for (j=0; j<n; j++) {
for (i=0; i<n; i++) {
a[j][i+1] = 2.0*a[j][i-1];
}
}
```

The resulting loop will incur an overhead of parallel work distribution only once, while previously the overhead would have been incurred "n" times. The MP C compiler, as usual, performs safety and profitability analysis to determine whether to perform loop interchange.

6. Aliasing and Parallelization.

ANSI C aliasing can often prevent loops from getting parallelized. Aliasing occurs when two references might refer to the same memory location. Consider the following example:

```
example <n>
    void copy(float a[], float b[], int n) {
        int i;
        for (i=0; i < n; i++) {
           a[i] = b[i];   /* S1 */
      }
  }
```

Since variables a and b are parameters, it is possible that a and b may be pointing to overlapping regions of memory. e.g if copy were called as follows:

example:

```
copy (x[10], x[11], 20);
```

In the called routine, two successive iterations of the copy loop may be reading and writing the same element of the array x. However, if the routine copy were called as follows:

example:

```
copy (x[10], x[40], 20);
```

then no possibility of overlap exists in any of the 20 iterations of the loop that will be executed. In general, it is not possible for the compiler to analyze this situation correctly without knowing how the routine is called.

MP C provides a keyword extension to ANSI C that permits the programmer to convey this kind of aliasing information. See description of Restrict Pointers below.

6.1. Array and Pointer References.

Part of the aliasing problem is caused by the nature of the C language in defining array referencing and definition via pointer arithmetic.

In order for the compiler to effectively parallelize loops, either automatically or explicitly via pragmas, all data that is laid out as an array must be referenced using C array reference syntax and not pointers. If pointer syntax is used, the compiler cannot determine the relationship of the data between different iterations of a loop. Thus it will be conservative and not parallelize the loop.

6.2. Restricted Pointers

In order for a compiler to effectively perform parallel execution of a loop, it needs to determine if certain lvalues designate distinct regions of storage. Aliases are lvalues whose regions of storage are not distinct. Determining if two pointers to objects are aliases is a difficult and time consuming process because it could require analysis of the entire program.

Consider function vsq() below:

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

The compiler can parallelize the execution of the different iterations of the loops if it knows that pointers a and b access different objects. If there is an overlap in objects accessed through pointers a and b then it would be unsafe for the compiler to execute the loops in parallel. At compile time the compiler does not know if the objects accessed by a and b overlap by simply analyzing the function vsq(); the compiler may need to analyze the whole program to get this information.

Restricted pointers are used to specify pointers which designate distinct objects so that the compiler can perform pointer alias analysis. To support restricted pointers, the keyword "_Restrict" is recognized by our ANSI C compiler as an extension. Below is an example of declaring function parameters of vsq() as restricted pointers:

```
void vsq(int n, double * _Restrict a, double * _Restrict b)
```

Pointers a and b are declared as restricted pointers, so the compiler knows that the regions of storage pointed to by a and b are distinct. With this alias information, the compiler is able to parallelize the loop.

_Restrict is a type-qualifier, like volatile, and it shall only qualify pointer types. _Restrict is recognized as a keyword only for compilations modes -Xa (default) and -Xt. For these two compilation modes, the compiler defines the macro __RESTRICT to allow users to easily write portable code with restricted pointers. For example, the following code should work on our compiler (all compilation modes) as well as on other compilers which do not support restricted pointers:

```
#ifdef __RESTRICT
  #define restrict _Restrict
#else
  #define restrict
#endif
 void vsq(int n, double * restrict a, double * restrict b)
 {
        int i;
        for (i=0; i<n; i++)
           b[i] = a[i] * a[i];
}
```

Should restricted pointers become part of the ANSI C Standard, it is most likely that "restrict" will be the keyword. Users may want to write code with restricted pointers using:

```
#define restrict _Restrict
```

as in vsq() because this way there will be minimal changes should "restrict" become a keyword in the ANSI C Standard. We chose _Restrict as the keyword because it is in the implementor's name space, so there is no conflict with identifiers in the user's name space.

There are situations in which a user may not want to change the source code. One can specify pointer valued function parameters to be treated as restricted pointers with the command line option:

```
-xrestrict=[func1,...,funcn]
```

If a function list is specified, then pointer parameters in the specified functions are treated as restricted; otherwise, all pointer parameters in the entire C file are treated as restricted. For example, -xrestrict=vsq, would qualify the pointers a and b given in the first example of the function vsq() above with the keyword _Restrict.

It is critical that _Restrict be used correctly. If pointers qualified as restricted pointers point to objects which are not distinct, loops may be incorrectly parallelized resulting in undefined behavior. For example, assume that pointers a and b of function vsq() point to objects which overlap, such that b[i] and a[i+1] are the same object. If a and b are not declared as restricted pointers the loops will be executed serially. If a and b are incorrectly qualified as restricted pointers the compiler may parallelize the execution of the loops, which is not safe, because b[i+1] should only be computed after b[i] had been computed.

7. Explicit Parallelization and Pragmas

As mentioned earlier, often there isn’t sufficient information available to the compiler to make a good decision on the legality or profitability of parallelization. MP C supports pragmas that allow the programmer to effectively parallelize loops that otherwise would be too hard or impossible for the compiler to handle.

7.1. Serial Pragmas

There are two serial pragmas, both apply to “for” loops:

1) #pragma MP serial_loop
2) #pragma MP serial_loop_nested

7.1.1. Serial loop pragma

```
#pragma MP serial_loop
```

The “serial_loop” pragma indicates to the compiler that the next “for” loop is not to be implicitly/automatically parallelized.

7.1.2. serial_loop_nested Pragma

```
#pragma MP serial_loop_nested
```

The “serial_loop_nested” pragma indicates to the compiler that the next “for” loop and any “for” loops nested within the scope of this “for” loop are not to be implicitly/automatically parallelized. The scope of the serial_loop_nested pragma does not extend beyond the scope of the loop to which it applies.

7.2. Parallel Pragmas

There is one parallel pragma:

```
#pragma MP taskloop
```

The MP taskloop pragma can, optionally, take one or more of the following arguments.

1) maxcpus (number_of_processors)
2) private (list_of_private_variables)
3) shared (list_of_shared_variables)
4) readonly (list_of_readonly_variables)
5) storeback (list_of_storeback_variables)

6) savelast

7) reduction (list_of_reduction_variables)

8) schedtype (scheduling_type)

Only one options can be specified per MP taskloop pragma, however the pragmas are cumulative and apply to the next for loop encountered within the current block in the source code:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop shared(a,b)
#pragma MP taskloop storeback(x)
```

These options may appear multiple times prior to the for loop to which they apply. In case of conflicting options, the compiler will issue a warning message.

7.3. #pragma MP taskloop [options]

A MP taskloop pragma applies to the next “for” loop within the current block.

A MP taskloop pragma suggests to the compiler that, unless otherwise disallowed (see Ineligibility issues), the specified for loop should be parallelized.

7.3.1. Nesting of “for” loops

There is no nesting of parallelized for loops by the MP C.

7.3.2. Ineligibility issues

For loops with irregular control flow and unknown loop iteration increment are not eligible for parallelization. For example, for loops containing setjmp, longjmp, exit, abort, return, goto, labels, and break should not be considered as candidates for parallelization.

Inter-iteration dependencies:

Of particular importance is to note that for loops with inter-iteration dependencies are not ineligible for explicit parallelization. This means that if a MP taskloop pragma is specified for such a loop the compiler will simply honor it, unless the for loop is disqualified (see Ineligibility issues). It is the user’s responsibility to make sure that such explicit parallelization will not lead to incorrect results.

7.3.3. Interaction between serial pragmas and parallel pragmas

If both the “serial_loop” or “serial_loop_nested” and “taskloop” pragmas are specified for a for loop the last one specified will prevail.

Consider the following example:

```
#pragma MP serial_loop_nested
    for (i=0; i<100; i++) {
   # pragma MP taskloop
     for (j=0; j<1000; j++) {
             ...
    }
   }
```

The i loop will not be parallelized but the j loop can be.

7.4. #pragma MP taskloop maxcpus(number_of_processors)

Specifies the number of processors to be used for this loop, if possible.

The value of maxcpus must be a positive integer. If maxcpus equals 1, then the specified loop will be executed in serial. (Note setting maxcpus to be 1 is equivalent to specifying the "doserial" directive.) The smaller of the values of maxcpus or the interpreted value of the PARALLEL environment variable will be used. When the environment variable PARALLEL is not specified, it is interpreted as having the value 1.

If more than one "maxcpus" pragma is specified for a for loop the last one specified will prevail.

7.5. Default scoping rules.

A variable used in a loop is classified as being either a "private", "shared", "reduction", or "readonly" variable. The variable will belong to only one of these classifications. A variable can only be classified as a reduction or readonly variable via an explicit pragma. See #pragma MP taskloop reduction and #pragma MP taskloop readonly. A variable can be classified as being either a "private or "shared" variable via an explicit pragma or through the following default scoping rules.

A private variable is one whose value is private to each processor processing some iterations of a for loop. In other words, the value assigned to a private variable in one iteration of a for loop is not propagated to other processors processing other iterations of that for loop. A shared variable, on the other hand, does not have this property. A shared variable is a variable whose current value is accessible by all processors processing iterations of a for loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop. Loops being explicitly parallelized through use of #pragma MP taskloop directives, that contain references to shared variables, must ensure that such sharing of values does not cause any correctness problems (i.e. race conditions). No synchronization is provided by the compiler on updates and accesses to shared variables in an explicitly parallelized loop.

In analyzing explicitly parallelized loops, the compiler uses the following "default scoping rules" to determine whether a variable is private or shared:

- If a variable is not explicitly classified via a pragma, the variable will default to being classified as a shared variable if it is declared as a pointer or array, and is only referenced using array syntax within the loop. Otherwise, it will be classified as a private variable.
- The loop index variable is always treated as a private variable and is always a storeback variable.

It is HIGHLY RECOMMENDED that all variables used in an explicitly parallelized for loop, be explicitly classified as one of shared, private, reduction or readonly, so to avoid the "default scoping rules."

Since the compiler does not perform any synchronization on accesses to shared variables, extreme care must be exercised before using a MP taskloop pragma for a loop that contains, for example, array references. If inter-iteration data dependencies exist in such an explicitly parallelized loop, then its parallel execution may give erroneous results. The compiler may be able to detect such a potential problem situation and issue a warning message, but in general it may not be able to do so. In any case the compiler will not disable the explicit parallelization of loops with potential shared variable problems.

7.6. Private Variables

```
#pragma MP taskloop private (list_of_private_variables)
```

Specifies all the variables that should be treated as private variables for this loop. All other variables used in the loop that are not explicitly specified as shared, readonly, or reduction variables, will be either shared or private as defined by the default scoping rules.

A private variable is one whose value is private to each processor processing some iterations of a loop. In other words, the value assigned to a private variable by one of the processors working on iterations of a loop is not propagated to other processors processing other iterations of that loop. A private variable has no initial value at the start of each iteration of a loop and must be set to a value within the iteration of a loop prior to its first use within that iteration. Execution of a program with a loop containing an explicitly declared private variable whose value is used prior to being set will result in undefined behavior.

7.7. Shared Variables

```
#pragma MP taskloop shared (list_of_shared_variables)
```

Specifies all the variables that should be treated as shared variables for this loop. All other variables used in the loop that are not explicitly specified as private, readonly, storeback or reduction variables, will be either shared or private as defined by the default scoping rules.

A shared variable is a variable whose current value is accessible by all processors processing iterations of a for loop. The value assigned to a shared variable by one processor working on iterations of a loop may be seen by other processors working on other iterations of the loop.

7.8. Read only variables

```
#pragma MP taskloop readonly (list_of_readonly_variables)
```

Read-only variables are a special class of shared variables that are not modified in any iteration of a loop. Specifying a variable as readonly indicates to the compiler that it may use a separate copy of that variables value for each processor processing iterations of the loop.

7.9. Storeback Variables

```
#pragma MP taskloop storeback (list_of_storeback_variables)
```

A storeback variable is one whose value is computed in a loop, and this computed value is then used after the termination of the loop. The last loop iteration values of storeback variables are available for use after the termination of the loop. Such a variable is a good candidate to be declared explicitly via this directive as a storeback variable when the variable is a private variable, whether by explicitly declaring the variable private or by the default scoping rules.

Note that the storeback operation for a storeback variable occurs at the last iteration of the explicitly parallelized loop, regardless of whether or not that iteration updates the value of the storeback variable. In other words the processor that processes the last iteration of a loop may not be the same processor that currently contains the last updated value for a storeback variable. Consider the following example:

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
 for (i=1; i < n; i++) {
     if (...) {
         x =...
```

```
        }
    }
      printf ("%d", x);
```

In the above example the value of the storeback variable x printed out via the printf() call may not be the same as that printed out by a serial version of the i loop, because in the explicitly parallelized case, the processor that processes the last iteration of the loop (when i==n), which performs the storeback operation for x may not be the same processor that currently contains the last updated value for x. The compiler will attempt to issue a warning message to alert the user of such potential problems.

In an explicitly parallelized loop, variables referenced as arrays are not treated as storeback variables. Hence it is important to include them in the list_of_storeback_variables if such storeback operation is desired (for example, if the variables referenced as arrays have been declared as private variables).

7.10. Savelast

This pragma specifies that all the private variables of a loop be treated as a storeback variables. The syntax of this pragma is as follows:

```
#pragma MP taskloop savelast
```

It is often convenient to use this form, rather than list out each private variable of a loop when declaring each variable as storeback variables.

7.11. Reduction Variables

```
#pragma MP taskloop reduction (list_of_reduction_variables)
```

All variables appearing in the reduction list will be treated as reduction variables for the loop. A reduction variable is one whose partial values can be individually computed by each of the processors processing iterations of the loop, and whose final value can be computed from all its partial values. The presence of a list of reduction variables can facilitate the compiler in identifying that the loop is a reduction loop, allowing generation of parallel reduction code for it.

Consider the following example:

```
#pragma MP taskloop reduction(x)
     for (i=0; i<n; i++) {
         x = x + a[i];
  }
```

the variable x is a (sum) reduction variable and the i loop is a(sum) reduction loop.

7.12. Scheduling Control

MP C compiler supports several pragmas, that can be used in conjunction with the taskloop pragma, to control the loop scheduling strategy for a given loop. The syntax for this pragma is as follows:

```
#pragma MP taskloop schedtype (scheduling_type)
```

This pragma can be used to specify the specific scheduling_type to be used to schedule the parallelized loop. Scheduling_type can be one of the following:

- static

In static scheduling all the iterations of the loop are uniformly distributed among all the participating processors

Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(static)
 for (i=0; i<1000; i++) {
   ...
}
```

In the above example, each of the four processors will process 250 iterations of the loop.

- self [(chunk_size)]

  In self scheduling each participating processor processes a fixed number of iterations (called the "chunk size") until all the iterations of the loop have been processed. The optional chunk_size parameter specifies the "chunk size" to be used. Chunk_size must be a positive integer constant, or variable of integral type. If specified as a variable chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

  Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(self(120))
 for (i=0; i<1000; i++) {
   ...
}
```

  In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

  120, 120, 120, 120, 120, 120, 120, 120, 40.

- gss [(min_chunk_size)]

  In guided self scheduling each participating processor processes a variable number of iterations (called the "chunk size") until all the iterations of the loop have been processed. The optional min_chunk_size parameter specifies that each variable chunk size used must be at least min_chunk_size in size. Min_chunk_size must be a positive integer constant, or variable of integral type. If specified as a variable min_chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

  Example:

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
 for (i=0; i<1000; i++) {
   ...
}
```

In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 10, 10, 10.

- factoring [(min_chunk_size)]

  In factoring scheduling each participating processor process a variable number of iterations (called the "chunk size") until all the iterations of the loop have been processed. The optional min_chunk_size parameter specifies that each variable chunk size used must be at least min_chunk_size in size. Min_chunk_size must be a positive integer constant, or variable of integral type. If specified as a variable min_chunk_size must evaluate to a positive integer value at the beginning of the loop. If this optional parameter is not specified or its value is not positive, the compiler will select the chunk size to be used.

  Example:

```
#pragma MP taskloop maxcpus(4)
 #pragma MP taskloop schedtype(factoring(120))
  for (i=0; i<1000; i++) {
     ...
}
```

  In the above example, the number of iterations of the loop assigned to each participating processor, in order of work request, are:

  125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10.

7.13. Legal pragma combinations

8. Compiler Options

- -xautopar: Turns on automatic parallelization.

  This option is effective only with optimization levels 3 or 4. If this option is used at optimization level less than 3, the compiler will change the optimization to -xO3 and emit a warning.

- -xexplicitpar: Generate parallelized code based on specification of #pragma MP directives.
  This option is effective only with optimization levels 3 or 4. If this option is used at optimization level less than 3, the compiler will change the optimization to -xO3 and emit a warning.

- -xparallel: This option is a macro and is equivalent to "-xautopar and -xexplicitpar". This option is effective only with optimization levels 3 or 4. If this option is used at optimization level less than 3, the compiler will change the optimization to -xO3 and emit a warning.

- -xreduction: Turns on automatic reduction recognition in iropt.
  This option is valid only when -xautopar or -xparallel, or -xexplicitpar with -xdepend, is specified. A warning is generated when -xreduction is specified but one of the following option sets is not -xautopar or -xparallel, or -xexplicitpar with -xdepend.
- When reduction recognition is enabled, the compiler will parallelize reductions as dot products, maximum and minimum finding. These reductions will yield different roundoffs.
- -xloopinfo: This option causes the emission of informative messages from compiler. These messages indicate which loops were parallelized and which ones were not. A short reason for not parallelizing a loop is also given. A warning is generated when -xloopinfo is specified but one of the following option sets is not -xautopar or -xparallel, or -xexplicitpar with -xdepend.
- -xdepend: This option causes dependence based transformations to be done in compiler. These dependence based transformations include: loop interchange, loop fusion, scalar replacement, and eliminating dead array assignments.
  This option is effective only with optimization levels 3 or 4. If this option is used at optimization level less than 3, the compiler will change the optimization to -xO3 and emit a warning.
  When this option is also specified with -xexplicitpar, automatic parallelization is also enabled.
- -xvpara: This option prints out warning messages for loops that have #pragma MP directives specified when compiler thinks the loop may not be properly specified for parallelization. For example, when compiler detects data dependencies between loop iterations.
- -Zlp[=X]: This option causes the compiler to generate profile instrumentation for all loops in the program. This information is used by looptool.
  This option is effective only with -xdepend, explicitly or implicitly specified. If this option is used without -xdepend, explicitly or implicitly specified, the compiler will turn on -xdepend and emit a warning. If this option is used at optimization level less than 3, the compiler will change the optimization to -xO3 and emit a warning.
- -xrestrict=[func1,...,funcn]: This option causes pointer valued function parameters to be treated as restricted pointers.
  If a function list is specified with this option, then pointer parameters in the specified functions are treated as restricted; otherwise, all pointer parameters in the entire C file are treated as restricted.
  This command line option can be used on its own, though it is recommended that for proper effect optimizations be turned on. e.g.

  > “cc -xO3 -xrestrict= prog.c” will treat all pointer parameters in the file “prog.c” as restricted pointers.
  >
  > “cc -xO3 -xrestrict=agc prog.c” will treat all pointer parameters in the function “agc” in file “prog.c” as restricted pointers. Also defines “-D__RESTRICT” to when used with -Xa and -Xt (default).

9. Licensing

The MP C automatic and explicit parallelization features are currently licensed with the Sun WorkShop.