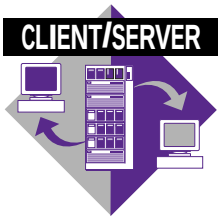
XL Fortran Compiler for IBM SMP Systems



By Dattatraya H. Kulkarni, Sudarsan Tandri, Lisa Martin, Nawal Copty, Raul Silvera, Xin-Min Tian, Xing Xue, and Julian Wang

This article describes salient features of the XL Fortran V5 compiler, which leverages parallelism in Symmetric Multiprocessor (SMP) systems for improving the performance of scientific applications. The article also includes guidelines for performance debugging and tuning for effective parallelization.

The XL Fortran V5 compiler accepts full Fortran 90 as well as several new language features that support parallelization and performance tuning on Symmetric Multiprocessors (SMP). Nested loops, the major source of parallelism in scientific applications, account for a substantial portion of execution time. The XL Fortran compiler can be used to automatically identify the parallel loops. In addition, programmers can guide the compiler by providing directives to indicate available parallelism. In order to enable portability and code migration, the XL Fortran compiler includes user directives from the OpenMP specification. OpenMP is supported, or is planned to be supported, by many vendors, including SGI, DEC®, KAI, and IBM.

The compiler uses a fork-join computation model, implemented using POSIX™ threads, for executing parallel loops. The fork-join model enables the implementation of several scheduling strategies to improve locality and load balance, and an effective parallel execution that is tolerant to load variance in multi-user environments.

In this article, we outline the architecture of the XL Fortran V5 compiler and briefly describe the functionality of the components. Second, we will describe automatic parallelization of nested loops in the compiler. We will illustrate examples of programmer support in the XL Fortran V5 compiler to express parallelism. We describe the methodology used to implement the fork-join computation model for executing parallel loops on the SMP processors.

The XL Fortran compiler performs a seamless set of optimizations for competitive performance on both uniprocessor and multiprocessor systems. We believe that these optimizations are useful for the programmer to hand optimize the code as well. We also summarize the performance of the compiler on standard serial and parallel benchmarks.

SMP Architecture for Parallel Computation

A symmetric multiprocessor system has multiple processors that access a global shared memory. It typically has 2 to 16 processors, which share a single address space (shown in Figure 1). More importantly, each processor in an SMP requires the same number of cycles to access a data item from the global shared memory in the absence of contention.

SMP systems are attractive platforms for parallel computation because they provide the shared memory computation model that is easier to program. Compared with a message passing computation model, they provide a simpler path to migrate serial applications onto parallel platforms. They can also be used for other general-purpose

computational needs, such as enterprise computing.

Because scientific applications tend to be computation and resource intensive, there is a tremendous advantage in porting these applications to parallel hardware. A majority of these applications were originally written for uniprocessor systems, which means they cannot exploit the power of SMPs in their current form. Fortunately, nested loops account for a substantial portion of execution time, and they can potentially be executed in parallel across multiple processors of an SMP. Ideally, a compiler should identify all the parallel loops and optimize the application for good overall performance.

Parallelism and the XL Fortran Compiler

The XL Fortran V5 compiler helps exploit the parallelism in IBM SMP hardware to improve the performance of scientific applications. It can automatically identify parallel loops and generate code to execute them across the processors of the SMP. However, scientific applications can have nested loops that cannot be automatically parallelized by the current compiler technology.

The XL Fortran compiler provides directives to help users guide the compiler in parallelization. The compiler also provides directives to identify critical sections of code and indicate policies for balancing the workload across SMP processors. Thus, the XL Fortran V5 compiler serves two purposes:

- It automatically parallelizes nested loops for faster porting of serial applications onto SMP systems.
- It has user directives that guide parallelization for careful performance tuning of the applications. These user directives conform to the directives provided by most vendors.

The XL Fortran V5 compiler also supports explicit multithreaded programming with a Fortran 90 interface to the AIX `pthread` library. The compiler supports native Fortran constructs, such as the `CHARACTER` data type and array language constructs, which can be used with the `pthread` interface.

Another new feature introduced in XL Fortran V5 is asynchronous I/O, which is needed for speed and efficiency in large scientific applications. The performance benefits result from the ability to overlap the input and output of large chunks of data with the execution of other statements.

Symmetric Multiprocessor Architecture

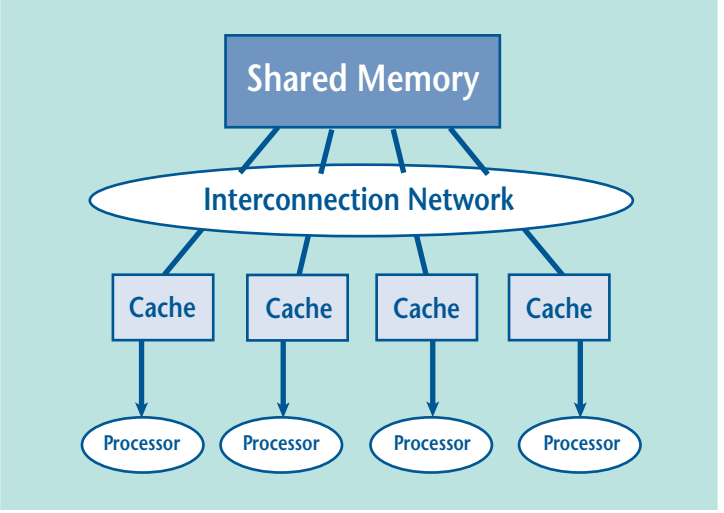


Figure 1. Symmetric multiprocessor architecture

The Compiler Architecture

The architecture of the XL Fortran V5 compiler is as shown in Figure 2. The internals of the XL Fortran compiler have four main component groups. The first group is the Fortran 90 frontend, which takes as input the user's source program, which may be annotated with directives for SMP parallelization. The frontend produces a high-level intermediate representation of the code for use by subsequent phases. In addition, the frontend ensures that the source conforms syntactically and semantically to the Fortran language definition and emits error messages for invalid source statements.

The language supported by the frontend includes full Fortran 90, several industry extensions, as well as selected Fortran 95 features. FORTRAN 77 is a subset of the Fortran

90 language definition, so the XL Fortran compiler supports full FORTRAN 77 as well.

The second group contains the scalarizer for converting the array language statements of Fortran 90 into FORTRAN 77 loops, as well as locality optimizer, serial and SMP optimizer, parallelizer, and finally, the outliner.

The third group contains components that support the actions of the components in the second group. The components in this group analyze the data flow and dependence in the program, and implement loop and data transformations required by the components in the second group.

The final group consists of the optimizing backend, which produces object code from the internal representation of the parallelized input program, and the SMP runtime.

Scalarizer

The scalarizer transforms the high-level intermediate representation of Fortran 90 array language statements into scalar `DO` loops. Through the use of data dependency information, the scalarizer eliminates temporaries, where possible, to generate efficient code for array language statements. The scalarizer also generates inline code for most Fortran 90 array intrinsics. Inlining of Fortran 90 intrinsics eliminates explicit calls, extra array temporaries, and array copying, which helps Fortran 90 programs to achieve performance comparable to FORTRAN 77 programs.

Invoking the scalarizer before parallelization is an important optimization. This is because the compiler can exploit parallelism in nested loops generated by the scalarizer as well as nested loops coded explicitly by the user.

Automatic Parallelization of Nested Loops

The locality optimizer, other optimizers for serial and parallel execution, and the parallelizer are within the high-order transformation framework of the compiler. The optimizations are robust in that they systematically combine transformations for improved serial execution with transformations for exploiting maximum parallelism.

Fortran V5 Compiler Architecture

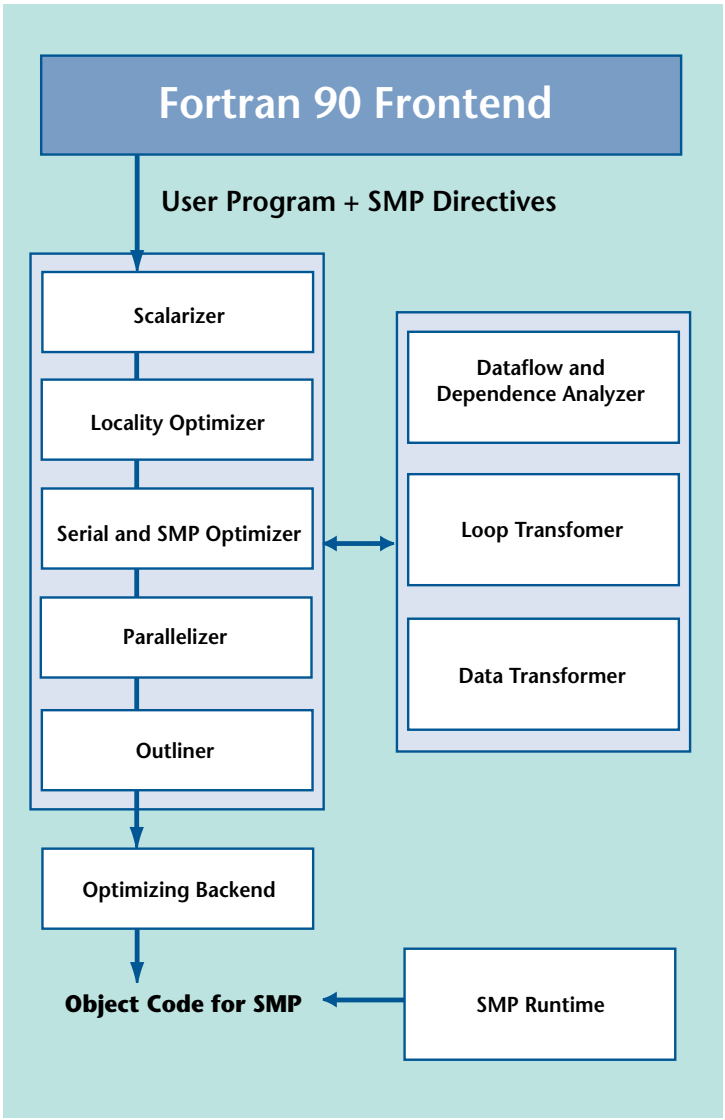


Figure 2. XL Fortran V5 compiler architecture

The locality and other optimizers use loop reordering and array padding transformations in order to improve the overall program performance. For example, nested loops are reordered so that the arrays are accessed with the smallest stride first to improve cache locality. Other optimizations performed within the compiler include loop tiling, loop unrolling, replacement of array references by references to temporary scalars, and elimination of conditionals. Most of these optimizations use cost models

that are parameterized by the hardware characteristics of the target system.

The parallelizer uses loop reordering transformations to automatically parallelize loops at the outermost levels. This minimizes parallelization overheads and ensures larger computation granularity on each of the processors of the SMP system.

Outlining

Outlining is a program transformation technique that is the converse of subroutine inlining. Whereas inlining a subroutine inserts the code in the subroutine at the call site, outlining converts a region of code into a subroutine and replaces the region of code by a call to the subroutine.

The outliner in the XL Fortran compiler converts parallel loops identified either by the compiler or specified by the user into subroutines. These subroutines implement a fork-join model for executing parallel loops with the help of calls to the SMP runtime library.

Outlining is a key ingredient in allowing scheduling schemes that make the performance of applications tolerant to load variations in multi-user SMP environments.

Augmenter

The main tasks of the augmenter include:

- Implementing XL Fortran's parameter passing conventions
- Creating runtime descriptors and implementing operations that require knowledge of their contents, including pointer assignment, `ALLOCATE`, and `DEALLOCATE`
- Inlining and generating calls to a variety of XL Fortran intrinsic functions

Automatic Parallelization

Nested loops in Fortran programs have long been recognized as an important source of parallelism. A majority of these loops have control structures and data access patterns that have made formal frameworks for automatic transformation and parallelization feasible. Over the past several years, the technology for automatically parallelizing nested loops has matured. The XL Fortran V5 compiler incorporates a substantial amount of this technology for parallelizing Fortran 90 programs on IBM SMP systems.

The loop and data transformer, locality optimizer, serial and SMP optimizer, and parallelizer interact with each other extensively. These components are collectively called the transformer in the XL Fortran V5 compiler. Input to the transformer is an internal representation of the scalarized Fortran 90 program. The transformer optimizes the input program with a perfectly nested loop at a time, with the help of the data flow and dependence analyzer. Optimizations in the transformer have five main purposes:

- Identify parallel loops automatically
- Derive transformations that improve cache locality
- Perform optimizations that target certain serial and SMP-specific optimizations, such as replacing invariant array references by scalars and recognizing scalar and array reductions
- Check viability and legality of parallelism due to assertion directives on the loop such as the `INDEPENDENT` directive (see "Assertion Directives" below)
- Retain parallelism on the loop due to prescriptive directives, such as `PARALLEL DO`

The basis for the loop transformations is the *unimodular transformation* framework that enables loop interchange, skew, reversal, and permutations to be applied uniformly. As a first step, the transformer distributes the loops to maximize opportunities for parallelization. The loop fusion that follows locality optimization and parallelization fuses the loops together, in case it is beneficial and legal to do so. Loop fusion increases the granularity of computation in each iteration

and may improve cache performance and instruction-level parallelism.

Locality optimization is central to the transformer, where nested loops are transformed so that arrays are accessed with the smallest stride first. A loop nest may be tiled, if necessary, for improving cache performance. Locality optimization also identifies a group of innermost loops, called the *locality group*, where all the array accesses are contained in the cache. The later optimizations and parallelization attempt to preserve the locality group for continued locality of access.

The transformer uses unimodular transformations to expose available parallelism in outermost loops. While doing this, loops belonging to locality groups are parallelized only if the transformer cannot parallelize other loops.

An important aspect of the parallelization technique in the transformer is the use of information about the loop provided by the programmer. For a loop with the `INDEPENDENT` directive, the loop is parallelized unless the dependence analysis finds precise information that the loop iterations are dependent.

For a loop with the `CNCALL` directive, the loop is parallelized unless there is a dependence due to references other than the arguments to the subroutine and the call itself. For a loop with the `PERMUTATION` directive, the dependence analyzer assumes that there is no dependence due to array references indexed by the permutation variable. In some instances, two array references indexed by a permutation variable may refer to the same memory location—the loop is not parallelized in such cases.

The transformer assumes that all loops with the `PARALLEL DO` directive are indeed parallel. In fact, the transformer may prevent certain optimizations and reordering transformations to preserve the semantics of the `PARALLEL DO` directive.

The transformer also unrolls both inner and outer loops for improved instruction schedules. In order to eliminate repeated index computations, scalars replace invariant accesses to arrays.

Support for Programmer-Driven Parallelization

XL Fortran V5 supports programmer-driven parallelization of applications. This support is useful in tuning the application performance with the automatic parallelization in the compiler. The support enables programmers to express both data and task parallelism. They can improve data parallelism by providing hints to the compiler on parallelism in nested loops. They can implement task parallelism by creating independent threads of control, which may have data or task parallelism.

User Directives for Data Parallelism

XL Fortran V5 provides two classes of SMP directives: *assertion directives* and *prescriptive directives*. The assertion directives are *hints* to the compiler, whereas the prescriptive directives are *directions* to the compiler.

Assertion directives provide the compiler with additional information about loop characteristics. The automatic parallelizer can use this information to determine whether a given loop should be parallelized. Prescriptive directives can help in forcing the compiler to parallelize or to mark sections of code as critical. Thus, prescriptive directives provide users with more control over which sections of code are to be parallelized or which shared resources are to be protected by simultaneous access. The assertion and prescriptive directives supported in the XL Fortran V5 compiler are shown in Figure 3.

SMP directives are Fortran comment lines that begin with a special character sequence called a *trigger*. The compiler

Programmer Directives in XL Fortran

Assertion Directives	INDEPENDENT CNCALL PERMUTATION ASSERT
Prescriptive Directives	PARALLEL DO PARALLEL SECTIONS CRITICAL

Figure 3. Programmer directives in XL Fortran

```
      x = 0
!SMP$ INDEPENDENT, REDUCTION(X), NEW(I)    ! The iterations of the loop
      do i = 1,m                           ! are independent, where x is
        x = x + i**2                       ! a reduction variable.
      end do                               ! i is private to each iteration.
```

Figure 4. INDEPENDENT directive

```
!SMP$ ASSERT (ITERCNT(100), NODEPS)  ! The loop has approx. 100 iterations
      do i = 1,n,step                ! and there are no dependencies between
                                     ! iterations.
        a(i) = a(i-m) * parm
      end do
```

Figure 5. ASSERT directive

```
!SMP$ CNCALL
      do i = 1,n
        call no_side_fx(a(i),b)
      enddo
```

Figure 6. Concurrently called procedures

recognizes several default triggers when SMP compilation is requested (for example, `SMP$` and `$OMP`).

Comment lines beginning with such triggers are processed as SMP directives. By default (when SMP compilation is not requested), the triggers are not recognized and the lines with directives are treated as comments. This is important because it allows users to add directives to a program and compile it for either serial or parallel execution.

Assertion Directives

The following describes several assertion directives.

INDEPENDENT: This directive is adopted from High Performance Fortran. The `INDEPENDENT` directive asserts that the iterations of the `DO` loop that follows can be executed in parallel. It supports two clauses:

- `NEW` clause with a list of variables specifies that new objects should be created for variables in the list
- `REDUCTION` clause that lists variables appearing in reduction operations

Figure 4 shows an `INDEPENDENT` directive.

ASSERT: The `ASSERT` directive is an extension of the `-qassert` compiler option. It can indicate to the compiler that no data dependencies exist between iterations of a loop. It can also approximate the iteration count for a loop with runtime bounds, which can help the compiler determine whether parallelizing a given loop would be beneficial. Figure 5 shows an example of the `ASSERT` directive.

CNCALL: This directive asserts that procedures invoked within the loop can be called concurrently by separate iterations of the loop. Figure 6 shows an example.

```
!SMP$ PERMUTATION(index)
      do i = 1,n
         a(index(i)) = a(index(i)) + b      ! There are no data dependencies
                                            ! between iterations of this loop
                                            ! if index has no repeated values.
      enddo
```

Figure 7. PERMUTATION directive

```
!SMP$ PARALLEL DO PRIVATE(i), SHARED(a), REDUCTION(s), &
!SMP$              IF(n > 100), SCHEDULE(GUIDED)

      do i = 1,n                              ! This loop will be executed in
        a(i) = i*2                            ! parallel if there are more than
        s = s + a(i)                          ! 100 iterations.
      enddo
```

Figure 8. PARALLEL DO prescriptive directive

PERMUTATION: Data dependence analysis is difficult and often not possible when array subscript expressions are complex. Applications with irregular computations typically have array subscript expressions that involve references to other arrays. The PERMUTATION directive helps the compiler analyze nested loops that have such indirect array references by specifying arrays whose elements have unique values. It is difficult for the compiler to determine that a loop is independent if array element references within it use other arrays in subscript expressions. The compiler uses this directive to identify parallelism within loops, which have complex subscript expressions referencing the arrays specified in the PERMUTATION directive. Figure 7 shows an example.

The primary prescriptive directives are as follows: PARALLEL DO, CRITICAL, and PARALLEL SECTIONS.

PARALLEL DO: This directive can direct the compiler to parallelize a loop. It allows the programmer to indicate which variables should be private to the independent iterations of the loop and which variables should be shared. The directive also provides a REDUCTION clause, which is similar to the REDUCTION clause of the INDEPENDENT directive. The IF clause specifies a runtime condition under which a loop should be parallelized. The SCHEDULE clause indicates which chunking algorithm should be used to partition the loop. Figure 8 shows an example.

CRITICAL/END CRITICAL: CRITICAL sections are blocks of code that should be executed by a single thread at a time. These directives should protect access to shared resources such as shared variables. Each CRITICAL section is associated with a named lock variable; however, critical sections that do not specify explicit lock names share the same global default lock.

If a lock name is specified for more than one critical section, the compiler will allow only one thread to execute any of these sections at a time. See Figure 9 for an example.

PARALLEL SECTIONS/END PARALLEL SECTIONS: PARALLEL SECTIONS allow users to mark multiple independent sections of code for parallel execution. The PARALLEL SECTIONS construct supports clauses similar to PARALLEL DO; variables may be specified as PRIVATE, SHARED, and REDUCTION; the IF clause may be used to specify conditional parallelization. Note that although this construct is a convenient way to parallelize

```
!SMP$  PARALLEL DO  PRIVATE(I)
       do i = 1,n
          .
          .
          .
!SMP$  CRITICAL (lock_my_lib)              ! Calls to non-thread-safe
         call my_lib_routine(a,b)          ! library routines should
!SMP$  END CRITICAL (lock_my_lib)          ! be serialized
          .
          .
          .
       enddo
```

Figure 9. CRITICAL SECTION prescriptive directive

```
!SMP$  PARALLEL SECTIONS
!SMP$  SECTION
           call compute_sum(a)
!SMP$  SECTION
           call compute_min_element(a)
!SMP$  END PARALLEL SECTIONS
```

Figure 10. PARALLEL SECTIONS directive

independent blocks of code, it does not scale well since the number of parallel work items (that is, the number of `SECTION` clauses) is constant. See Figure 10.

Thread-safe Parallel I/O

The XL Fortran thread-safe I/O library, `libxlf90_r.a` , provides support for parallel execution of Fortran 90 I/O statements. It is essential to use this library in Fortran applications that contain I/O statements in a parallelized loop or in applications that create multiple threads and execute I/O statements from within different threads at the same time. In other words, such applications must be linked with this library to get the expected results.

Synchronization of I/O Operations

During parallel execution, multiple threads can perform I/O operations on the same file at the same time. If the threads are not synchronized, then the results of these I/O operations could be shuffled or merged so that the applications produce incorrect results or even crash. The XL Fortran thread-safe I/O library synchronizes I/O operations for parallel applications to ensure the integrity and correctness of each individual I/O operation. The synchronization, performed within the I/O library, is transparent to application programs. The synchronization for external files is performed on a per unit basis. When a thread is performing an I/O operation on one unit, other threads attempting to perform I/O operations on the same unit will have to wait until the first thread finishes its operation. For the purposes of I/O synchronization, all internal files are treated as though they were one single logical unit.

The XL Fortran thread-safe I/O library sets its internal locks to synchronize access to logical units. This should have no functional impact on the I/O operations performed by a Fortran program. Also, it will not impose additional restrictions to the operability of Fortran I/O statements other than what they already have, except for the use of I/O statements in a signal handler that is entered because of an asynchronous-generated signal.

Non-determinacy in Parallel I/O

The order in which parallel threads perform I/O operations is not predictable. The XL Fortran thread-safe I/O library does not control the ordering.

Parallel I/O can only be used in cases where each thread performs I/O on a predetermined record in *direct* access files, where the result of an application does not depend on the order in which records are written out or read in, or where each thread performs I/O on a different file. In these cases, results of the I/O operations are independent of the order in which threads execute.

For multiple threads to write or read the same *sequential* access file, however, the order of records written out or read in depends on the order in which the threads execute the I/O statement on them. Since this order is not predictable, the result of an application could be incorrect if it supposes records are sequentially related and cannot be written out or read in any order. For example, if the following loop:

```
do i = 1, 500
print *, i
enddo
```

is parallelized, the numbers printed out will no longer be in the sequential order from 1 to 500 as the result of a serial execution.

Outlining: Fork-Join Computation Model

Two basic approaches have been used for executing the work in different iterations of a parallel loop: the fork-join and the Single Program Multiple Data (SPMD) models. In the fork-join model of computation, a single thread—the main thread—sets up and coordinates parallel work, while all threads participate in performing the work. On the other hand, the SPMD computation model requires the compiler to set up the work for all the threads. In the fork-join model, threads are typically created at the start of work and destroyed once the work is completed. To optimize the thread creation overhead, threads are created once and they wait for work to be set up. The parallel work is divided among processors according to a loop scheduling strategy. Thread synchronization of the threads at the end of the parallel loop is implicitly accomplished by the main thread, which monitors the completion of work.

There are two reasons for this outlining. First, it creates a context for parallel execution. Secondly, it simplifies storage management, since each thread executing the subroutine will have a separate copy of the local variables automatically allocated on its stack, while non-local variables will be shared among the threads. The outlined subroutines are executed by all the participating threads with the help of a runtime library.

Figure 11 illustrates outlining of an example parallel loop, which transforms the parallel loops into a subroutine.

Optimizing Applications for SMPs

The key to optimizing applications on SMPs is improving parallelism and cache locality by either compiler optimizations or careful coding by the programmer. In this section, we first outline selected optimizations performed by the XL Fortran V5 compiler.

> ***The key to optimizing applications on SMPs is improving parallelism and cache locality by either compiler optimizations or careful coding by the programmer.***

Programmers can use these optimizations as templates to hand-tune applications on SMPs. In the second subsection, we provide broad guidelines for performance debugging and performance tuning of applications while using the XL Fortran V5 compiler.

Compiler Optimizations

The XL Fortran V5 compiler performs several optimizations to enhance both serial and parallel application performance. These optimizations are aimed at improving locality of data access and parallelism in nested loops, improving locality and balance of workload among parallel threads, and reducing access to data shared among the parallel threads.

Before Outlining

```
program user_prog
integer a(100),   i

parallel do i = 1, 100
       a(i) = i
end do

print * ,   a
end
```

After Outlining

```
program user_prog
integer    a (100) , i
integer _parDoBounds (3), _parDoChunkCtl(4) , _parDoStep

_parDoBounds    * . . .
_parDoChunkCtl = . . .
_parDoStop        = . . .
call _xlsmpParDoSetup (_parDoBounds, _parDoChunkCtl,
                                        _outlined_sub_1, 1. _parDoStep)
print *,  a


contains

subroutine _outlined_sub_1 (_libControl, _parDoStep)
integer _libControl, _parDoStep
integer i_1

do while (_xlsmpParDoChunk (_libControl, _parDoFrom, _parDoTo)
      do i_1 = _parDoFrom, _parDoTo, _parDoStep
           a(i_1) = i_1
      end do
end do

return
end _outlined_sub_1

end
```

Figure 11. Parallel Loop Outline

The compiler transforms nested loops in an effort to parallelize loops at the outermost loop levels. This improves the granularity of computation performed by each thread and also reduces the overhead of executing parallel iterations. While parallelizing the loops, the compiler avoids the parallelization of nested loops that must be serial for better cache locality. The runtime system has scheduling schemes that preserve locality in the parallel loops and distribute computational load evenly among the parallel threads.

Idiom recognition has been vital in exploiting the performance of multiprocessor systems. The XL Fortran V5 compiler uses scalar and array reductions and privatization as an integral part of the loop transformation process. Whenever possible, the compiler attempts to replace array references by scalars to reduce index computation overhead as well as implement certain array reductions efficiently. The compiler minimizes access to shared data by identifying variables that can be private to parallel

Benchmark #	Serial Time (s)	Parallel Time (s)
101.tomcatv	1101	727
102.swim	1719	570
103.su2cor	716	420
104.hydro2d	1442	504
107.mgrid	907	329
110.applu	955	713

Figure 12. The preliminary performance of SPECFP95 benchmarks with the XL Fortran V5 compiler on a 4-processor J40 SMP system

threads. The compiler also has techniques to identify repeated calls to intrinsics and replace them by efficient routines that apply intrinsics to a vector of array elements.

Performance Debugging and Tuning

The XL Fortran V5 compiler can generate a report on the parallelization and locality enhancements performed by the compiler by invoking the `-qreport=smplist` command-line option. The generated report is a pseudo-Fortran listing of the transformed input program. In this report, parallel loops are explicitly identified with a `PARALLEL DO`.

The report contains the compiler's reasons for not parallelizing certain loops. These reasons are useful as pointers for identifying the directives that programmers can insert to help parallelize the loops. For example, the compiler cannot parallelize a loop with a subroutine call because the compiler cannot determine the side-effects of the call. In such cases, the programmer may be able to analyze and determine that the calls to the subroutine indeed do not have any side-effects that affect the execution order of the iterations. Therefore, the programmer may insert the `CNCALL` directive to help the compiler in parallelizing the loop. However, an imprecise dependence in the loop can still prevent the compiler from parallelizing the loop. If this occurs, the programmer can analyze the loop to determine if the loop iterations are independent and insert the `INDEPENDENT` directive. Note that the `INDEPENDENT` directive is only an assertion, so the programmer may need to use the `PARALLEL DO` directive to force the compiler to parallelize the loop.

The programmer's next task is to improve the performance of parallel loops. The efficiency of a parallel loop is good when the benefits of dividing the work among the processors is much higher than the overhead of parallel execution. In particular, inner loop parallelism incurs much overhead because of barrier synchronization at the end of every parallel iteration. For this reason, parallelism in the outer loop is preferable to parallelism in the inner loops. In general, it is essential to prevent the parallelization of loops with very little computation and enable the parallelization of loops with significant computation.

Good performance results from a balance between parallelization and locality optimization. The XL Fortran V5 compiler performs several optimizations to improve locality of data access. While specifying the `PARALLEL DO` directive for a loop, it is necessary to ensure that the parallel threads access data mostly from the local cache. Sometimes, it is useful to replace the `PARALLEL DO` directive with the `INDEPENDENT` directive so that the compiler has freedom to make the trade-off between locality and parallelism in the loop.

Locks protect the access to shared resources in `CRITICAL SECTIONS`. The overhead due to these locks can be high, especially when they occur inside a nested loop.

In such cases, it may be worthwhile to investigate the need for the `CRITICAL SECTION` or to serialize the loop nest.

Performance Results

Figure 12 shows the preliminary performance of SPECFP95 benchmarks using the XL Fortran V5 compiler. The serial time corresponds to the best serial execution time without any parallel overheads. The parallel time corresponds to the execution time on an IBM SMP with four PowerPC 604™ processors.

Conclusion

We described the XL Fortran V5 compiler, which helps the programmer exploit the parallelism in IBM SMP hardware. Five features of the compiler are noteworthy:

- The compiler accepts the full Fortran 90 language as well as selected Fortran 95 features.
- It implements both advanced serial optimizations and automatic parallelization of nested loops seamlessly.
- It supports industry-accepted user directives, which help the programmer in tuning parallel program performance.
- It supports thread-safe parallel I/O.
- It provides support for task parallelism through a Fortran 90 interface to the AIX `pthreads` library.

The XL Fortran V5 compiler provides an effective means to port serial scientific applications onto SMPs and to tune parallel applications on SMPs.

Dattatraya H. Kulkarni, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Kulkarni is a member of the compiler team at the SWS Toronto Laboratory focusing on high-level optimizations in the XL Fortran compiler. His areas of interest include programming languages, optimizing compilers, and environments. He holds a PhD in Computer Science from the University of Toronto.

Sudarsan Tandri, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Sudarsan is a member of the compiler development team at the SWS Toronto Laboratory. His main focus has been robust runtime support for parallelization on SMP systems. His areas of interest include compiler and emerging applications on multiprocessor systems. He holds a PhD in Computer Science from the University of Toronto.

Lisa Martin, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Ms. Martin is a member of the compiler development team at the SWS Toronto Laboratory. She is the architect of the XL Fortran compiler. She holds a BMath degree from the University of Waterloo.

Nawal Copty, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Ms. Copty is a member of the compiler development team at the SWS Toronto Laboratory. She is currently working on the outlining support for the XL Fortran compiler. Her interests include parallel languages, algorithms, and compilers, and issues in parallel and distributed computing. She has a PhD in Computer Science from Syracuse University in New York.

Raul Silvera, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Silvera is a member of the compiler development team at the SWS Toronto Laboratory. He has an MS in Computer Science from McGill University in Montreal.

Xin-Min Tian, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Tian is a member of the compiler development team at the SWS Toronto Laboratory. He has a PhD from the Tsinghua University in Beijing.

Xing Xue, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Xue is a member of the compiler development team at the SWS Toronto Laboratory. He specializes in the I/O runtime environment and the augmentor phase of the compiler. He has a PhD in Computer Science from Nanjing University in China.

Julian Wang, IBM Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1V7. Mr. Wang is a member of the compiler development team at the SWS Toronto Laboratory. His areas of interest include compiler development and the operating system. He has an MS in Computer Science from the University of Toronto.