IA64 Oriented OpenMP Compiler: Design and Implementation of Fortran Front End

Hongshan Jiang, Suqin Zhang, and Jinlan Tian

Department of CS&T,
Tsinghua University,
Beijing, 100084, China

Abstract. This paper presents an OpenMP compiler Fortran Front End. It introduces principles and algorithms to deal with the implicit data parallelism, which is directed by WORKSHARE directive in OpenMP Fortran API V2.0. For implementation, automatic parallel computation division by compiler is achieved by front end's converting implicit data parallelism to explicit one in Very High WHIRL. In addition, this paper presents some optimization techniques to handle compiler-generated redundant synchronization and consistent DO loop. At the end, a performance experiment is given to prove the effectiveness of mentioned strategy.

1 Introduction

IA64 (Itanium Architecture)[1] is a 64-bit architecture published recently by the Intel Corporation. The main characteristics of this architecture include the following. The instruction level parallelism is enhanced by more structural components such as bigger register heap and instruction bundles that are able to accommodate up to three instructions. By using the branch registers, the conditional registers and the conditional execution instructions, branches have been reduced to the least. By allowing compilers to schedule load operation beforehand and enabling cache management of memory hierarchy, the storage latency is concealed. Finally, the special hardware enabled function returning and invoking mechanism supports those modular codes generated by compilers.

OpenMP is a shared-memory parallel programming language initiated by DEC, IBM, Intel and SGI in Oct. 1997. OpenMP inherits a number of features from ANSI X3H5 standard and Pthreads of IEEE. Its most prominent features[2,3] are its support of incremental parallelism and its suitability for developing coarse-grain parallelism in the thread level. Because of its easy learning and usage together with the wide support from several famous parallel machine venders, it has become the standard of shared-memory parallel programming language.

Using IA64 processors to construct SMP and adopting OpenMP as the parallel programming environment has many advantages due to its effective utilization of IA64's powerful instruction level parallelism capability in conjunction with OpenMP's advantage in thread level parallelism. ORC (Open Research Compiler) is a public compiler research infrastructure aimed for the IA64 architecture. Based on it, we have developed an OpenMP compiler to support OpenMP Fortran API V2.0. This paper mainly discusses the design and implementation of the Fortran front end of this OpenMP compiler.

2 Design and Implementation

The front end of ORC includes a C/C++ front end and a Fortran front end. It performs lexical analysis, syntax analysis and semantic analysis on source code and finally generates its corresponding Very High WHIRL intermediate presentation. The Fortran front end descends from Cray F90 front end. Its architecture is shown in figure 1.

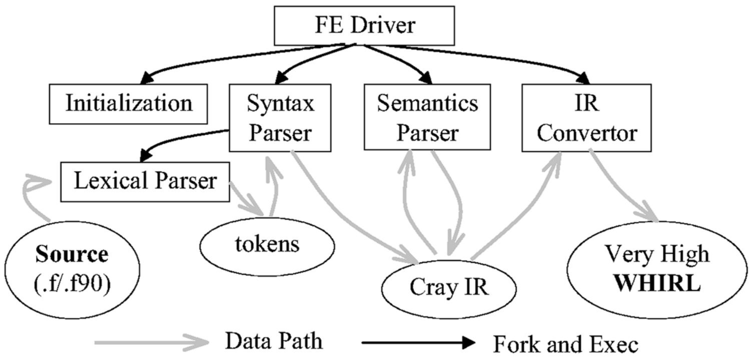

Fig. 1. Control flow and data flow of Fortran FE.

OpenMP is not an independent programming language. It can be considered as a parallel extension to current sequential high-level language such as Fortran or C/C++. The extensions can be compiling directives in source code, functions in programming libraries or OS environment variables. Each front-end processes corresponding version of OpenMP separately.

Current version of Fortran front end can only handle OpenMP Fortran API V1.0 directives. Based on it, we added the processing mechanism for new features in OpenMP Fortran API V2.0, such as the support for NUM_THREADS, COPYPRIVATE clause and WORKSHARE[3] directive etc. Moreover, some semantic processing needs modification, such as the processing for REDUCTION clause. The processing of OpenMP directive spreads over every phase of the front end from lexical analysis to semantic analysis. After being translated to VH WHIRL, OpenMP directives are changed to pragma statements or WHIRL regions with pragma labels.

The handling of WORKSHARE directive, which occupies a large amount of workload for the compiler implementation, is difficult for Fortran front end. WORKSHARE directive is a means for OpenMP Fortran API V2.0 to support implicit data parallelism in Fortran90 i.e. it does not specify the parallel model explicitly. For instance, in the array assignment operation of Fortran90, assignment of different elements of arrays can be executed in parallel, but how these assignment operations are dispatched to different processors is not specified distinctly. The implicit data parallelism can be regarded as a simple way for programmers to specify parallel behaviors so as to free programmers from some trivial issues such as computation division of parallelism. The only thing a programmer should to do is to supply some necessary simple information, the remainder is taken care of by the compiler.

Our solution for implicit data parallelism is to transform WORKSHARE region to WHIRL region with WORKWHARE pragma by front end, and then translate implicit data parallelism in WORKSHARE region to explicit form. This involves the division of computation. In the next section, we will elaborate computation division principles, algorithms and implementations. Furthermore, we will introduce some optimization techniques for the special features of OpenMP. The IA64 related optimization is placed in the back end.

3 Process of Implicit Data Parallelism

A basic block that can be executed in data parallelism is specified by WORKSHARE directive in OpenMP. The data parallelism manner of such basic block is not specified explicitly but handled by the compiler. Syntax of WORKSHARE directive is shown in figure 2.

```
!$OMP WORKSHARE
   block
!$OMP END WORKSHARE [NOWAIT]
```

Fig. 2. Syntax of WORKSHARE directive.

In the figure, statements in the block are the array assignment of Fortran90. WORKSHARE directive indicates that work of each statement in *block* is shared i.e. they are divided into separate units of work and are executed by a group of threads in parallel. If it is guaranteed that each unit will be executed exactly once, the units of work may be assigned to threads of the PARALLEL region[3] in any manner.

3.1 Computation Division Principle

Array operations of Fortran90 can be classified into four categories: elemental operations, query operations, reduction operations and complex operations. According to the semantics of WORKSHARE directive, we introduce the following computation division principle of array operations in the WORKSHARE region.

1. In the case of elemental operations, the evaluation for each element of the array is counted as one unit of computation.
2. In the case of the reduction operation, current work is divided into several units with each unit being executed in one thread. Each unit is to evaluate the local reductive result, and then an additional work unit is added to deduce local results to global reductive results. Necessary synchronization is added at the end of each local reduction.

3. In the case of complex array function operations, the computation division depends on semantics. The principle is to assign irrelevant operations to separate threads. For operations that have strong internal data dependency such as PACK[4] and UNPACK[4], the whole operation forms a single unit of work.
4. If none of the above rules applied, that operation is treated as a single unit of work.

3.2 Computation Division Algorithm

According to semantic requirement, statements in the WORKSHARE region are array assignments, scalar assignments, FORALL statements or WHERE statements[4]. Actually, the last two are conditional assignments, so the main issue of parallel compiling algorithm is the assignment division and the expression division.

Assignment division belongs to the elemental computation division, which can be processed according to the aforementioned principles. Solution of expression division is discussed as follows. The evaluation of expression can be treated as the post-order traversal of an expression binary tree. Every internal node of the tree corresponds to a single array operation while every leaf node corresponds to an array or scalar. Therefore, expression can be mapped to a series of single evaluation statements by post-order traversing an expression tree. For every single evaluation statement, its right value is an array expression that has no more than one operator or function, while its left value is a temporary variable of an array or a scalar. We divide every single evaluation statements into units of work by the rule of aforementioned principles, combine consistent work units, eliminate redundant temporary variables, and thus generate statement series that have been divided into work units. Consistency of work units herein means to compute the elements in the same position of two arrays of the same shape.

For instance, suppose the number of current threads is 2, ***A*** and ***B*** are one-dimensional arrays of length *N*. According to statement WHERE(***B***<0)***B***= ***A***-MIN(***A***), its computation division is illustrated in Figure 3.

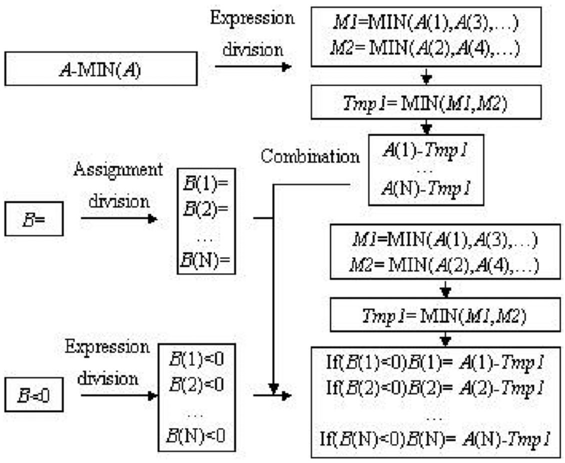

Fig. 3. Example of computation division.

3.3 Implementation

Our implementation is to convert implicit parallelism to explicit one in VH WHIRL. For each array operation in Fortran90, we give its corresponding explicit form of operation. For instance, if A is a two dimensional array of shape N×M, B=TRANSPOSE(A) can be translated to the explicit parallel form as in Figure 4.

```
!$OMP DO PRIVATE(I,J)
  DO I=1,M
    DO J=1,N
      B(I,J) = A(J,I)
    END DO
  END DO
```

Fig. 4. Explicit form of parallelism of B=TRANSPOSE(A).

After a series of processing, this explicit form will be compiled to parallel codes by the back end. The particular way of computation division is decided by the number of threads at the time and environment variables such as OMP_SCHEDULE.

4 Compilation Optimization

The compilation optimizations are mainly focused on the optimization in VH WHIRL for features of OpenMP directive. Synchronization exists implicitly in many OpenMP directives and may leads to additional cost. How to reduce the synchronization cost to minimum is the main problem that the compilation optimization should care about. Moreover, most parallel computations of OpenMP program exist in loop statements. Therefore, the optimization of loop statement of OpenMP is also a problem needs consideration. The following are the optimization techniques we adopt.

1) Removal of Redundant Synchronization

Synchronization is implied at the end of DO loop directive of OpenMP. If successive loop directives are not data dependent, their redundant synchronization can be removed in order to reduce the waiting time and improve the parallel efficiency. As Figure 5 illustrates, A and B are two successive tasks which have no data dependency. These two tasks are executed by 2 threads in parallel. A synchronization operation occurs between them. Because of the workload imbalance, after thread 1 finishes A2 at time t1, it will wait till thread 0 finishes A1. At time t2, these 2 threads start to execute B. This will bring unnecessary extra execution time. We can see that after the removal of redundant synchronization between A and B, unnecessary waiting time is saved and parallel efficiency is improved.

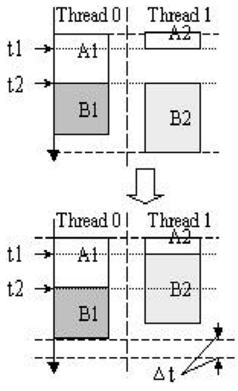

Fig. 5. Case of removal of redundant synchronization.

The time saved by the removal of redundant synchronization equals to the minimum waiting time of all threads, i.e. $T_{\text{saved}} = \text{MIN}(T_{\text{waiting}}1, T_{\text{waiting}}2, \ldots, T_{\text{waiting}}n)$. In fact synchronization always brings additional cost, so $T_{\text{waiting}}1$, $T_{\text{waiting}}2$, …, $T_{\text{waing}}n > 0$, thus $T_{\text{saved}} > 0$.

2) Combination of consistent loop

For successive loop statements, if their lower bound, upper bound and step length are all identical, and no data dependency occurs between loop bodies of different subscript, their loop bodies can be combined into one loop body. The execution order of these statements must be kept same as before. This can reduce iteration times, thus enhance the efficiency. Such cases usually exist in statement series converted from implicit parallel statements in Fortran90.

3) Optimization for Orphan Directives[5]

An orphan directive is an OpenMP directive that does not appear in the lexical extent of a PARALLEL construct, but may lie in its dynamic extent. If this directive does not lie in the dynamic extent of a PARALLEL construct, executing the sequential codes could gain higher performance. Therefore, for each orphan directive, whether to execute its parallel codes or to execute its sequential codes is dynamically determined by its context. Our solution is to add a statement, i.e. IF(OMP_IN_PARALLEL()), to tell if the directive is in parallel context or in sequential context. If in parallel context, the parallel version is executed, otherwise the sequential version is executed.

4) Parallelism of data independent Operations

Some operations inside WORKSHARE directive cannot be divided into parallel work units for their tightly coupling. Yet there is no data dependency among those operations. Hence it might be more efficient to execute them in parallel. Our solution is to put data independent operations into different SECTION blocks in one SECTIONS region[3] to execute them in parallel.

Removal of redundant synchronization has little influence on efficiency in balanced workload cases, meanwhile in imbalanced workload cases, efficiency improvement is proportional to the minimum waiting time of all threads. Combination of loops brings about 7% enhancements in efficiency depending on loop body sizes. Optimization for orphan directives can enhance efficiency in sequential cases only if the decision cost is less than that of parallel codes.

5 Test Result

On an Itanium2 parallel machine with four CPUs of 900MHz, 4G bytes memory, we tested our compiler which adopting the above compilation strategy. The tested programs use WORKSHARE directive and Fortran90 array operation statements. They are numeric computation programs such as array addition, multiplication and Jacobi Iteration Method. Testing data includes an integer array of 1 million elements. Table 1 shows the result of addition and multiplication of arrays: $\boldsymbol{D}=\boldsymbol{A}+\boldsymbol{B}*\boldsymbol{C}$, where $\boldsymbol{A},\boldsymbol{B},\boldsymbol{C},\boldsymbol{D}$ are integer arrays of 1 million elements. Table 2 shows the result of using Jacobi Iteration Method to solve the equation: $(\mathrm{d}^2/\mathrm{d}x^2)u+(\mathrm{d}^2/\mathrm{d}y^2)u - u = f$, where $u(x,y) = (1-x^2)(1-y^2)$ $(-1<x<1, -1<y<1)$, $f(x,y) = -2(1-y^2)-2(1-x^2)- (1-x^2)(1-y^2)$, with grid size $100 * 100$, iterative parameter 0.1, maximum tolerant error 10^{-7}, iterative number 61309.

Table 1. Addition and multiplication

Thread Number	Execution Time (s)	Speed-up	Eff. (%)
1	1.323	1	
2	0.673	1.97	98.5
3	0.462	2.86	95.3
4	0.414	3.20	80

Table 2. Jacobi Iteration Method

Thread Number	Execution Time (s)	Speed-up	Eff. (%)
1	190.02	1	
2	109.25	1.74	87
3	74.12	2.56	85.3
4	56.37	3.37	84.3

The results show that our parallel compilation strategy has gained considerable efficiency.

6 Conclusion

How to take advantage of new features in architecture development and to actively make progress in bridging the gap between architecture and programming interface has long been a hotspot of IT community. This paper introduces the design and implementation of Fortran front end of an IA64 oriented OpenMP compiler. It mainly discusses the processing of implicit data parallelism supported by OpenMP, and elaborates some optimization techniques related to OpenMP programming features. The proposed algorithm and methodology have general meaning to the design of friendly parallel programming interface and to the dispatching of many detail issues, like computation division of parallelism, to compiler for automatic processing so as to improve the programming efficiency. In addition, transformation between control dependency and data dependency has to be further studied to take advantage of new features of conditional instruction of IA64.

References

[1] Intel Cop., Intel®IA-64 Architecture Software Developer's Manual[OL]. http://segfault.net/~scut/cpu/ia64/, July 2000.

[2] CHEN Guoliang. Parallel Computing : Architecture, Algorithm and Programming[M]. Beijing:Higher Education Press, 1999. (In Chinese).

[3] The OpenMP ARB, OpenMP Fortran Application Program Interface Version 2.0[OL]. http://www.openmp.org/specs/, November 2000.

[4] Michael Metcalf, John Reid. Fortran 90 Explained[M]. London.:Oxford University Press,1990

[5] Matthias Müller. Some Simple OpenMP Optimization Techniques[A]. Rudolf Eigenmann, Michael J. Voss. OpenMP Shared Memory Parallel Programming[C]. Berlin:Springer, 2001. 31–39.