ORC-OpenMP: An OpenMP Compiler Based on ORC

Yongjian Chen, Jianjiang Li, Shengyuan Wang, and Dingxing Wang

Tsinghua University, Beijing 100084, P.R. China,
chenyj99@mails.tsinghua.edu.cn

Abstract. This paper introduces a translation and optimization framework for OpenMP, based on the classification of OpenMP translation types. And an open source OpenMP compiler, which implements this framework is also introduced as a high performance research platform for Linux/IA-64. Different from other open source OpenMP compilation system, this compiler has the following characteristics: First, it's integrated into the backend optimization compiler that mainly exploits Instruction Level Parallelism. This integral solution makes analyses and optimizations that require interactions between the instruction level and the thread level possible. Second, it's based on a unified model called translation type classification. This approach improves the code quality by reducing runtime overhead and code size.

1 Introduction

There is still not enough high performance OpenMP implementation with source code opened to public for further research usage. As far as we know, there are several other OpenMP compilers opened to public domain for research usage: OdinMP/CCp (translate C OpenMP programs to C programs with pThreads([2]), Omni OpenMP Compiler ([1]) and PCOMP (Portable Compiler for OpenMP, [3]). All these OpenMP compilers are implemented as source-to-source translators, and thus are loosely coupled with backend compilers. Although these source-to-source translation approaches gain the merit of portability, we find that they are not suitable as a research platform for OpenMP optimization, especially when we try to use such a platform to study thread level parallelism upon the traditional instruction level parallelism, which requires the ability to model the interactions between these two levels, since to the three compilers, the backend optimization compilers just appear as black boxes.

This paper introduces a new framework to translate and optimize OpenMP programs for shared memory systems, and based on this framework, an OpenMP implementation based on ORC (Open Research Compiler, [4]) for Linux/IA-64 (or simply called ORC-OpenMP), which is designed and implemented as a research platform on IA-64 based systems. Different from source-to-source translation strategies found in the three implementations mentioned above, our OpenMP compiler is implemented as a module inside ORC, integrated with other optimization modules of ORC. While the source-to-source approaches are more portable, implementing OpenMP as an integrated part of compilers has more opportunities to share information with other optimization modules in the compiler, and are more flexible to translate code in more complex way. Results of OpenMP

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 414–423, 2004.

benchmark show that the implementation can fulfill the performance requirement as a research platform.

The rest of the paper is organized in the following way: the framework to translate and optimize OpenMP programs is discussed in section 2. The design and implementation issues of ORC-OpenMP are presented as section 3, and section 4 presents performance results about this implementation. Section 5 summaries the work and also gives plans about future work. Bibliography comes as section 6.

2 The Translation and Optimization Framework

2.1 Dynamic Directive Nesting, Binding, and Directive Classification

To be more versatile, dynamic directive nesting and binding are introduced into OpenMP standards. It means that the binding of execution context of a specific OpenMP construct can be delayed to runtime. According to this, a program constructs may be executed sequentially, by multithreads, or by nested multithreads. Code 1 illustrates the case of dynamic nesting and binding.

Code 1. dynamic directive binding, orphaned directives and nested parallelism

```
#pragma omp parallel /*1. normal parallel */
{
#pragma omp parallel /*2. nested parallel*/
  {
  }
  foo( ); /* call site 1 */
}

foo( )
{
#pragma omp for  /*3. orphaned for */
  {
  }
}
```

In Code 1, unique numbers identifies three OpenMP directives and two call sites of function foo are also numbered. Since foo is compiled in a separate compiling unit, directive 3 is not in the lexical extent of its call sites, so its binding to the parallel region must be delayed until the call occurs in execution. At call site 1, the parallel for declared in directive 3 is bound to parallel region declared in directive 1, and at call site 2, the parallel for should be bound to master thread according to the binding rule. While these kinds of semantics tend to be more flexible, they bring more complexity into the implementation.

According to the lexical containing type relative to parallel region constructs, directives can be divided into three categories: normal directives, directives in nested parallel region, and orphaned directives. In Figure 1, if we assume the parallel region declared in directive 1 is the outmost parallel region during execution, then all the directives directly nested in its lexical extent (not include those in nested parallel regions) are called

normal directives in this paper. The directive 2 declares a parallel region inside another parallel region, and all it's containing directives are shortly called nested directives. The directive 3 is not in the lexical extent of any parallel region constructs, and it's binding to parallel regions is dynamic. Such kind of directives is called orphaned directives in the OpenMP specifications.

In order to produce better code, we extend the classification according to runtime binding relationship instead of the lexical containing relationship of constructs. The directives of only one level of parallelism are called normal directives, just as stated above, and nested directives are those inside multi-level parallel regions. Orphaned directives can be normal directives, nested directives, or serialized parallel directives according to the binding situation at runtime.

While such classification is not necessary for all OpenMP implementations, it helps to generate better code by eliminating redundant code and further enable other optimizations. In this paper, these attributes of OpenMP constructs are called translation types.

2.2 Translation Type Classification and the Framework

Since at compile time, the static analysis may not be able to determine all the dynamic bindings, the translation type of OpenMP directives can not always be determined as these three types. In the compiler, one more type is added called default directives type, to represent those whose translation types are not explicitly determined. The default directives type is the default setting for all directives. So the translation type can be one the following four:

1. default directives
2. normal directives
3. nested directives
4. serialized directives

We call the process of determining the translation type of directives "translation type disambiguation". Most of the case, such a process must cross the boundary of procedures or subroutines, thus must be done with the help of InterProcedural Analysis.

The framework for OpenMP translation and optimization is presented in Figure 1.

In this framework, frontends is used to translate OpenMP constructs in different languages into a common IR form, and later phases all work on this common IR. Two phases are necessary: pre-translation and optimization and OpenMP translation phase, while two others are optional: translation type disambiguation and post optimization phase. Translation type disambiguation has been introduced above. In this phase, translation types will be derived across procedure boundaries, and necessary data structures will be set up for later usage. Pre-translation and optimization works at the OpenMP directives level (but in the form of IR), and it may transform one OpenMP constructs into equivalent constructs, or do some optimizations such as redundant directives elimination, or merging immediately adjacent parallel regions. OpenMP translation phase is the main translation phase. It translates OpenMP constructs into low level constructs. Essentially, this phase translates OpenMP programs into normal multithreaded programs. An optional post optimization phase works on the multithreaded program level, doing optimizations such as deadlock and race condition detection.

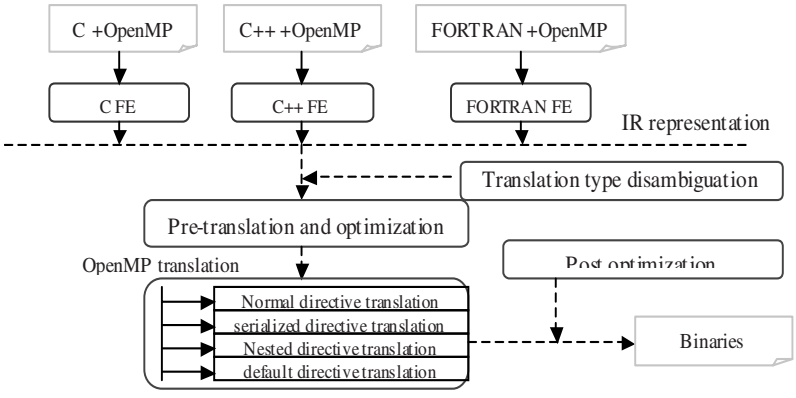

Fig. 1. The framework for OpenMP translation and optimization.

Normal directives, nested directives, serialized directives and default directives are treated differently for better performance.

2.3 Default Directives Translation

Default directives can be any of the other three translation types. So stuff code must be inserted to dynamically determine the region type and choose right code segment. An example translation is depicted in code 2. This translation implements the nested parallel regions as single-threaded groups.

Code 2. Default directives translation

```
#pragma omp for
{
/* for body */
}

(a) OpenMP constructs

if( _is_parallel()) {
  /* normal parallel do translation*/
}
else {
  /* serialized parallel do translation*/
}

(b) translated code
```

Since the OpenMP constructs are structural program constructs, the processing process of OpenMP constructs is also in a top-down manner. That means, when the compiler translates an OpenMP construct, all the OpenMP constructs inside this construct will also be translated.

2.4 Normal Directives, Serialized Directives, and Nested Directives Translation

Normal directives need not any stuff code to check the nesting type at runtime, and the directives are translated in the normal way. Further more, some common variables, such as thread id and number of threads, can be set up in a direct way at the beginning of parallel region.

Serialized directives can be stripped off from the programs, and the result code executes in sequential mode. For those directives with multiple translation types, two version of the procedure can also be generated.

Nested directives can be treated in two ways. First is to implement a pseudo-nested region, i.e., we treat the nested directives just like serialized directives and strip them away, and the nested parallelism is implemented as a one-thread group. While this is a simplified implementation, it may satisfy most practical requirements raised in SMP case. In the second case, nested parallelism must be supported. In this case, special stack must be maintained, to provide correct thread environment, such as thread group number and thread id. And further more, special thread task structures may be helpful to reduce the call stack and thus reduce the overhead of implementing multi-level thread creation.

There is still optimization space for this translation process and better translation may exist for some special cases. For example, when both the thread number of a parallel region (by using NUM_THREADS clause of PARALLEL directive) and the trip count of the do loop are known at runtime, for the static schedule type, since the loop iterations assignment can be determined totally at compile time, the prologue schedule code can be eliminated.

3 ORC-OpenMP

3.1 Introduction to ORC

ORC has an five-level IR (Intermediate Representation) named WHIRL ([5]) for its BE (BackEnd) processing, and various optimizations are carried out on different levels of WHIRL. In general, WHIRL is a tree like structure, and each WHIRL tree represents one function/subroutine in the programs. Generally, OpenMP regions are also represented as region type nodes in the WHIRL tree, and OpenMP directives are represented as pragma nodes attached to the region node. OpenMP directives without region scope are represented as single pragma nodes, just like normal statement nodes.

3.2 Framework of the OpenMP Module

The logical processing flow of OpenMP processing module in ORC is given in Fig. 2.

Four main components for OpenMP processing are indicated in the framework.

Two frontends process the OpenMP syntax in FORTRAN and C programs and translate OpenMP directives into proper WHIRL structures. The FORTRAN frontend accept FORTRAN90 syntax, and after a FORTRAN90 expansion module, the FORTRAN90 vector operations are translated into proper loops. After this point, except for special language divergences, the later processing modules don't bother with the language details anymore.

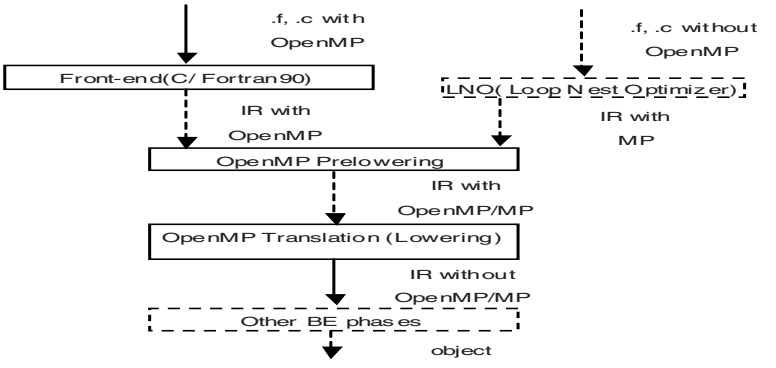

Fig. 2. OpenMP logical processing flow inside ORC

Two translation modules, OpenMP prelowering and OpenMP lowering, translate program constructs marked with OpenMP/MP pragmas to ordinary constructs using RTL API calls on IR level.

3.3 OpenMP Prelowering

OpenMP prelowering module is a pre-processing pass for OpenMP. In this module, OpenMP related program constructs are normalized and some constructs, such as SECTIONS, are translated into others for the convenience of the later OpenMP lowering module. The most important normalization operation is for loops declared to be parallel loops. The form of these loops is carefully adjusted to the form assumed by later lowering process.

In the pre-pass, OpenMP SECTIONS construct is translated into equivalent OpenMP DO construct. But in practice, more efficient program constructs such as branch-table will be used for translation.

3.4 OpenMP Lowering

The final translation of OpenMP constructs into RTL(RunTime Library) constructs is done in OpenMP lowering module. Translation (or lowering) of OpenMP constructs falls into two ways: direct one-to-one translation and WHIRL tree restructuring.

Only several OpenMP directives can be directly translated into corresponding RTL calls. An example is BARRIER. A BARRIER WHIRL node (represent a BARRIER directives in the program) lexically inside parallel regions can be replaced with one single WHIRL node with the type function call represents a call to RTL function __omp_barrier. ATOMIC and FLUSH directives with explicit arguments can also be translated in this way.

Other OpenMP region based constructs often need to restructure the corresponding WHIRL tree rather than simple replacement. This kind of constructs include PARALLEL region, DO loops, SINGLE and MASTER section, and CRITICAL section.

Nested directives are implemented in a simple way. Whenever the compiler ensure it encounter a nested parallel region, it simply strip off all the OpenMP directives contained

in the region except for a few synchronization directives. Thus the nested parallelism is always expressed as a single thread group.

The lowering pass is implemented based on the translation type classification framework, to eliminate the stuff codes required to handle dynamic nesting semantics. Compiler analysis is designed to gather information from the OpenMP programs or even profiling data.

A simple analysis is implemented in the IPA (InterProcedural Analysis) module. Every parallel region in the program is identified and the information is attached to the call graph generated by the main IPA analysis phase. A typical form of these annotated call graphs may look like Figure 3:

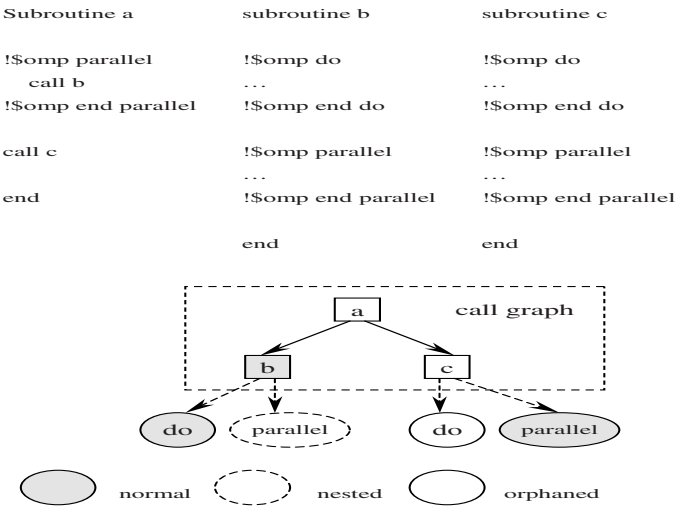

Fig. 3. Annotated call graph

For a given specific subroutine, if we know that it's a local call just like those decorated with qualifier static in C, or it will be called only in current module, then the annotated call graph can be used to extract information for further code simplification.

For example in Code 1, if we can determine from the call graph that all the calls to subroutine _foo_ are inside some parallel regions, then we can use this additional information to infer the directive attributes in subroutine _foo_. In this way, original Orphaned directives may be treated as normal directives and original normal directives will be treated as nested directives and some stuff code can be strip off because now the compiler get enough information about the context.

3.5 Data Environment Handling

Data environment constructs are not always translated into real code. More often, they specify where the specified variables are allocated and thus affect the way they are accessed. But for directives like REDUCTION, FIRSTPRIVATE, LASTPRIVATE and

THREADPRIVATE related constructs, stuff codes for initialization and finalization are also necessary.

For THREADPRIVATE variables, the official suggested implementation allocate such variables in special thread local storage sections .tdata or .tbss, and the real storage allocation and binding is done by the loader rather by compiler. In out implementation, we choose to let the program manage the allocation, binding and access, and the complexity is left to the compiler alone. The THREADPRIVATE variables are thus dynamically allocated in the heap storage.

4 Performance

4.1 Experiment Platform and Benchmarks

As a research platform, the implementation should also have satisfactory performance. NPB3.0 OpenMP FORTRAN benchmark ([6]) is used to test the FORTRAN performance. Another benchmark NPB2.3 OpenMP C version ([7]) developed as part of the Omni project is used to test the C OpenMP compilation functionality and performance. These benchmarks represent typical kinds of floating point workloads in science and engineering computation. The underlying system is a Linux SMP box with 4 itanium2 900MHz CPUs, 4GB main memory ("tiger 4" or "Bandera"). Benchmarks are compiled using Omni OpenMP compiler (1.4a) and our OpenMP compiler separately. Beyond the option to turn on OpenMP processing, for both compilers, only -O3 is specified as compiler option, and more aggressive optimizations, such as profiling and IPA are not turned on.

4.2 Performance Data

The experiment result of the NPB3.0 FORTRAN OpenMP benchmark suite is presented in figure 4. In this figure, benchmarks are presented in a form such BT.w. BT is the benchmark name, and w represents the problem size. The benchmark score is given in a metric defined as Mega-Operations per second, and the higher, the better. orf90 is our FORTRAN OpenMP compiler, and omf77 is the FORTRAN OpenMP compiler of Omni OpenMP compiler. Among the seven benchmarks, BT, SP and LU are application benchmarks, while CG, FT, MG and EP are kernel benchmarks.

The experiment result of the NPB2.3 C OpenMP benchmark suite is given in figure 5. 6 benchmarks are presented. The name convention is like Figure 4. orcc is our C OpenMP compiler, and omcc is Omni's C OpenMP compiler.

The benchmark scores simply demonstrate that our compiler's performance is much better than Omni's. And such a performance is what we need when we intend to use it as a research platform.

5 Summary

Indeed, this compiler is developed in hope to bridge the gap of traditional parallel compiler that exploits loop level parallelism and task level parallelism and traditional optimization compiler that mainly exploits instruction level parallelism. Possible research

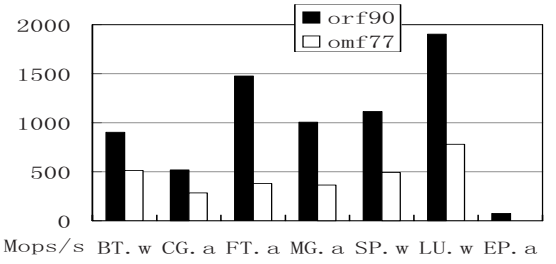

Fig. 4. OpenMP FORTRAN performance

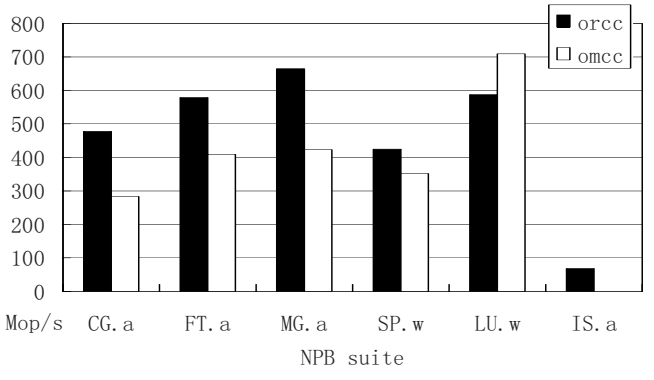

Fig. 5. OpenMP C performance. For LU.w, omcc gets better performance, but the compiled code produces wrong results and cannot pass the validity test

topics includes the interaction between thread level parallelism and instruction level parallelism, and auto-multithreading. Indeed, we are trying to use OpenMP as a tool to express thread level parallelism, and use other stuff multithreading techniques, such as helper thread to exploit the parallelism of applications at thread level, besides the efforts of exploiting instruction level parallelism. The requirement of exploiting parallelism at multiple levels demands for a unified cost model, and more direct communication between different modules. This is the motivation why we implement such an OpenMP compiler inside backend compiler, other than a standby one.

One important design decision in the design and implementation of this OpenMP compiler is the idea of translation type classification. This is in fact a static analysis for runtime context. By determine some of the contexts at compile time, the compiler can produce better code. Although the related optimizations are still not well explored, experiments show that the improvements in run time and code size is impressive.

In this paper, important design issues and tradeoffs for a special OpenMP implementation are presented, in hoping that it may help more OpenMP compilers for research use to be designed. As far as we know, this is the first open source OpenMP compiler on IA64 base platforms, and we also use this compiler as a research vehicle to study transitions from IA32 to IA64 in the HPC realm.

Acknowledgement. The work described in this paper is supported by Intel's university research funding, and partly supported by the Gelato project set up jointly by HP and Intel. We also want to thank the ORC group members for their work on ORC.

References

1. M. Sato, S. Satoh, K. Kusano and Y. Tanaka: Design of OpenMP Compiler for an SMP Cluster. In the 1st European Workshop on OpenMP (1999) 32–39
2. C. Brunschen, M. Brorsson: OdinMP/CCp-a portable implementation of OpenMP for C. Concurrency: Practice and Experience, Vol 12. (2000) 1193–1203
3. Seung Jai Min, Seon Wook Kim, M. Voss, Sang Ik Lee and R. Eigenmann.: Portable Compilers for OpenMP. In the Workshop on OpenMP Applications and Tools (2001) 11–19
4. Open Research Compiler: `http://ipf-orc.sourceforge.net`.
5. SGI Inc.: WHIRL Intermediate Language Specification. WHIRL Symbol Table Specification. (2000)
6. H. Jin, M. Frumkin, and J. Yan: The OpenMP implementation of NAS parallel benchmarks and its performance. NASA Ames Research Center Technical report, Report NAS-99-011. (1999)
7. RWCP: OpenMP C version of NPB2.3. `http://phase.etl.go.jp/Omni/benchmarks/NPB/index.html`.