OpenMP 指导语句全局嵌套类型的静态分析及应用*

陈永健+, 舒继武, 李建江, 王鼎兴

(清华大学 计算机科学与技术系 高性能计算研究所,北京 100084)

Static Analysis of OpenMP Directive Nesting Types and Its Application

CHEN Yong-Jian+, SHU Ji-Wu, LI Jian-Jiang, WANG Ding-Xing

(Institute of High Performance Computation, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

+ Corresponding author: Phn: +86-10-82611515 ext 1950, Fax: +86-10-82861400, E-mail: chenyj99@mails.tsinghua.edu.cn

Received 2003-12-18; Accepted 2004-07-21

Chen YJ, Shu JW, Li JJ, Wang DX. Static analysis of OpenMP directive nesting types and its application. *Journal of Software*, 2005,16(2):194−204. http://www.jos.org.cn/1000-9825/16/194.htm

Abstract: Because of the rules of dynamic directive nesting and binding, some of the thread context in OpenMP programs can only be totally determined at runtime. However, by compiling time static analysis, nesting type can be partly determined and this information can be passed to other compiling phases to guide later translation and optimizations. Since the binding and nesting may span the procedure boundaries through calls, local and global analyses are not enough. It is the interprocedural analysis that provides the most required ability. By integrating information into traditional interprocedural analysis, the nesting type information of procedures is propagated along call graphs. And later translation and optimization phases can bind this global information with local information inside the procedure to determine the nesting types at compiling time. The results demonstrate that in typical science and engineering workload the nesting type is highly determinable at compiling time, and the application of this information may achieve less runtime overhead and the reduced code size.

Key words: OpenMP; compiler; interprocedural analysis; global nesting type; OpenMP translation

摘 要: 由于指导语句动态嵌套与绑定规则的存在,OpenMP 程序中线程的一些上下文只能在运行时刻才能完全确定.然而,通过编译时刻的静态分析可以部分确定指导语句的嵌套类型,这些信息可以用于指导后续的编译与优化.由于函数调用的存在,嵌套与绑定常常会跨越过程边界,除了通常的局部和全局分析之外,还需要过

* Supported by the National Natural Science Foundation of China under Grant No.69933020 (国家自然科学基金)

CHEN Yong-Jian was born in 1977. He is a Ph.D. candidate at the Institute of High Performance Computation, Tsinghua University. His current research interests include compilation and optimization for high performance computation and high performance computer architecture. **SHU Ji-Wu** was born in 1969. He is an associate professor at the Institute of High Performance Computation, Tsinghua University. His research areas are large scale science and engineering computing, parallel and distributed processing and software, cluster communication, parallel simulation and SAN. **LI Jian-Jiang** was born in 1972. He is a Ph.D. candidate at the Institute of High Performance Computation, Tsinghua University. His current research interests include optimization and performance tuning tools for high performance computation, parallel algorithm for science and engineering computation. **WANG Ding-Xing** was born in 1937. He is a professor and doctoral supervisor at the Institute of High Performance Computation, Tsinghua University. His current research areas are parallel and distributed computer system.

程间分析的支持.通过在通常的过程间分析的基础上附加信息,可以使得嵌套类型信息在过程调用图中进行传播.将这些全局信息与过程内的局部信息结合起来,就可以在编译时刻确定语句的嵌套类型.结果表明,编译时刻的嵌套类型分析可以有效地确定通常的科学与工程计算程序中指导语句的嵌套类型,基于嵌套类型的翻译与优化可以同时减少运行时开销和目标代码长度.

关键词: OpenMP;编译;过程间分析;全局嵌套类型;OpenMP 翻译

中图法分类号: TP314 **文献标识码**: A

1 Introduction

In OpenMP, directives, especially parallel regions, may be lexically nested in other directives according to some specific rules. This is called nesting rules. Because of nesting and procedure calls, some directives have semantic relationship with other directives. This is called directive binding. Inside a procedure, the binding happens according to static scoping rules. In order to ease the parallelization process of sequential programs, the OpenMP standards[1] introduce dynamic binding and dynamic nesting rules to enable the incremental development feature and to let the programmers keep their focus on a local procedure other than the global structure of the whole program. Mainly, dynamic binding and nesting rules specify the necessary behavior of directives when calls occur and directives are involved in the callee and the call sites of the callers. The binding semantics are implemented through the cooperation of OpenMP compilers and OpenMP runtime systems.

The following simple example illustrates the case of dynamic nesting and binding (see Code 1). In the example, parallel region directive 2 is (lexically) nested and bound to directive 1, and directive 3 in function foo() is dynamically nested and bound to directive 1, through the call to foo at call site 1.

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

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

Code 1 Dynamic nesting and dynamic binding

The directive structures in OpenMP are executed by threads. According to different contexts, these threads may be the initial master thread in a sequential execution mode, or in the top level thread team, or in a nested thread team. These are called the execution mode of code blocks in this paper. Generally, different execution modes stand for different execution contexts, and maintain different amount of internal data structures, which represent the context. So making distinction between these execution modes may achieve better performance. This is the motivation of this paper. Since execution mode closely relates with nesting type, analysis to infer the nesting type is quite helpful to improve target code quality, both for OpenMP translation and OpenMP optimization.

In this paper, we show that a proper static analysis phase may be quite able to discover this binding and nesting type information of OpenMP constructs. Since the problem mainly exists because of the existence of calls, analysis that can handle calls and across procedure boundaries is critical to extract such kind of information. In this paper, an analysis is introduced based on the traditional interprocedural control flow analysis. When the call graph is built up when the interprocedural analysis is enabled, additional information collected by a local nesting type analysis is tagged to call nodes and the information is propagated among call graph to establish a nesting type call graph.

Nesting type information for multiple calls of a specified procedure may then be combined together to determine the procedure nesting type. Later phases can combine this global information with local information to determine the directive nesting type accurately.

The information obtained by static analysis can be used to guide later translation and optimizations. There are several cases where this information can be useful. First of all, by making such distinction between different binding types, more efficient translation can be done, since less stuff code is required to handle extra contexts. Secondly, with the extra information, more restrict use cases are assumed, and thus more aggressive optimizations are possible. Moreover, by summarizing the binding type information of a specific procedure, we can present the programmer a view of how consistent the procedure is during the parallelization process, and thus can further adjust the arrangement of OpenMP directives in the program.

This paper is organized as follows. The idea of binding type classification and the algorithms to determine the binding type at compiling time is introduced in Section 2. In Section 3, applications of this analysis are discussed. Section 4 comes with comments about related works. In Section 5, the summary is presented.

2 Determining Nesting Types by Compiling time Static Analysis

The problem is to determine how some interested directives, if not all, are nested in the scope of the whole program. In fact, in most of the cases we are only interested in whether these directives are nested in some parallel regions. For programs that comprise of many procedures distributed in many files, the task can be viewed as two subtasks. First we should determine the global context for all the directives in a procedure, i.e., determine how a procedure (i.e., all calls to the procedure), or every call to this procedure is nested. Then we can use this global context to determine the nesting type of individual directives in this procedure. Before we can discuss the algorithms solving this problem, a refinement of these free-used terms is required.

2.1 Nesting types of directives, calls and procedures

For each OpenMP directive in a procedure, it can be lexically nested in some other directives. Since every parallel region may create a new group of threads, the enclosing parallel region declaration directive is of most interests. For a given directive, we define the lexical nesting level of this directive as the number of levels of its enclosing parallel directives. In fact, this definition can be extended to other none-OpenMP constructs, since it may also make sense to some post-translation optimizations. But since in this paper, mainly the directive translation and optimizations at directive level are considered, and other constructs can be well related to some directives, we only consider directive lexical nesting level here. According to the lexical nesting level, directives can be categorized into three classes.

Definition 1. For a directive with lexical nesting level n, if n=0, the directive is lexically orphaned; if n=1, the directive is lexically normal; if n>1, this directive is said to be lexically nested.

These lexical properties are also called local nesting types of directives, since they represent how directives are nested in the scope of the procedure they are in. Since we are considering the global nesting properties of the directives, this definition must be extended to the scope of the whole program. To simplify the definition, we can selectively inline related procedures, so that all the nesting parallel directives are now in our scope, and the global nesting type now turns out to be the local nesting type in the new inlined procedure.

Definition 2. The global nesting level of a directive is the local nesting level of the corresponding directive in an ideally all-inlined program.

Similarly, the global nesting type can be defined. Different from local nesting type, we add a fourth type called global undetermined, since in practical the expansion of the program is not always possible because the existence of loops in call graphs. Because call statements are now in our consideration, the idea of nesting level and nesting type are extended beyond directives to all statements.

Definition 3. For a statement (or block of statements) with global nesting level N, if N=0, the statement is

globally orphaned; if N=1, the statement is globally normal; if N>1, the statement is globally nested. In cases that the value of global nesting level cannot be determined, the statement is said to be of a global nesting type of globally undetermined.

Like the definition of local nesting types, these global nesting properties of statements are called global nesting types of statements. For calls, the global nesting types form the global nesting-calling context of all statements contained in the procedure body. Further more, since a procedure may be called at many call sites, the global nesting types of calls to a given procedure can be combined into a single type. This is called the global nesting type of procedures in this paper.

For every instance of call to a given procedure, the local nesting types of the directives lexically contained in are the same, so if we can determine the nesting context of a given call or all the calls to a given procedure, we can determine the global nesting types of all contained directives.

2.2 Basic idea of nesting type inference

The call relationship of procedures can be represented by a static call graph at compiling time. By combining the local nesting type information of calls along a specific path in the static call graph, we can determine the global nesting types of a call that ends this path. This is the basic idea that drives the algorithms in this paper.

Now we take the OpenMP directives into consideration. Figure 1 illustrates the basic idea of the nesting type analysis. The local nesting information in subroutine *a* is used to set up the nesting context for calls to subroutines *b* and *c*. These nesting contexts are then used to determine the global nesting types of directives in *b* and *c*.

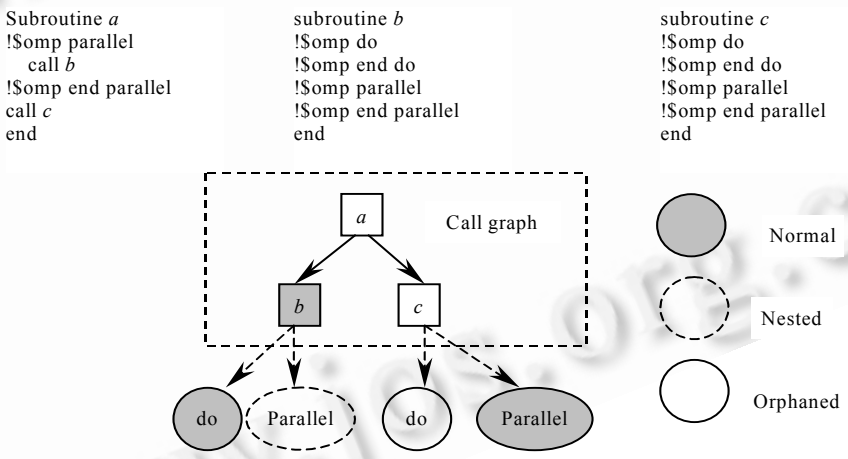

Fig.1 Inferring nesting types

The framework of the nesting type analysis is given in Fig.2. The nesting type analysis algorithm comprises a local nesting type analysis, an interprocedural nesting type analysis, and a combination phase. The local nesting type analysis algorithm is the basic algorithm underlying all three parts: the interprocedural part uses it to step forward along call graphs, and the combination phase uses it to infer the nesting types for directives in leaf functions.

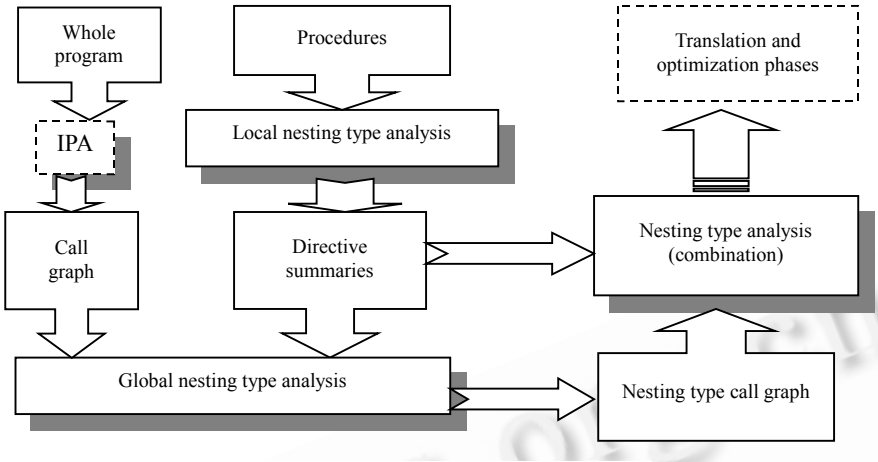

Fig.2 The framework of nesting type analysis

2.3 Local nesting type analysis

The local nesting type analysis summarizes the directives' nesting information for individual procedures. Since many OpenMP directives are block-scoped and can be nested, in the IR(intermediate representation) of compiler, these directives are also represented as regions, and these regions can be further nested. The whole procedure is organized as a tree with the procedure entry as the root, and the local nesting type analysis works in a top-down and hierarchical manner beginning with the root node. The algorithm is illustrated using pseudo-code 2. Function transNT() translates a local nesting level into a nesting type, and function isParallel() judges whether a statement is a parallel region directive.

```
procedure GetBlockNestingTypes
  rootNode: in, the Root of an IR tree
  currentNestingLevel: in, integer
  NTMap: out, statements × NestingType
begin
  newLevel: integer = currentNestingLevel
  tempNT: Nestingtype
  if rootNode is a leaf node then
    tempNT = transNT(currentNestingLevel)
    NTMap[rootNode]= tempNT
  else
     if isParallel(rootNode) then
       newLevel = newLevel + 1
     endif
     for each child p of rootNode
        GetBlockNestingTypes(p,newLevel,NTMap)
     endfor
  endif
end

procedure LocalNTAnalysis
  rootNode: in, Root of procedure's IR tree
  NTMap: out, statement × NestingType
begin
  getBlockNestingType(0, newLevel, NTMap)
end
```

Code 2 Local nesting type analysis algorithm

2.4 Interprocedural nesting type analysis

With the local nesting type analysis, the interprocedural nesting type analysis works on the static call graph to infer the global nesting type for interested directives in the whole program. A static call graph can typically be derived after an interprocedural analysis phase in the compiler.

The Interprocedural nesting type analysis algorithm walks through the static call graph top down in a

wavefront manner, starting from the main entrance of the whole program. When stepping forward along a specific path, the algorithm combines the local nesting type information with the global context so far established. The combination process is described by a state transition model depicted in Fig.3. In Fig.3, local nesting types of directives in current node (represents a procedure) are represented as states, and the global nesting type of the node (representing a call or a procedure) is treated as input. The target state is the global nesting type of these directives.

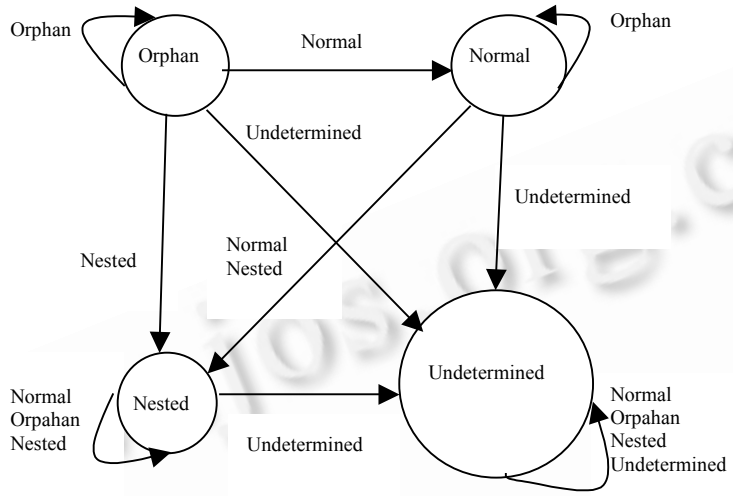

Fig.3 State transition model for nesting type combination

The global nesting type analysis algorithm is illustrated in pseudo-code (see code 3).

```
procedure walkCGOneStep
node: in, Node of call graph
NTMap: in, Map of local nesting type
callNTMap: out, Calls × NestingTypes
procNTMap: out, Procedures × NestingTypes
begin
 currentNT: NestingTypes
 if all incoming edge's NT available then
    procNTMap[node] = combineNT(node)
 endif
 else
    procNTMap[node] = undetermined
 endif
    currentNT = procNTMap[node]
 for each edge p from node
  callNTMap[p] = STM_NT(currentNT,NTMap[p])
 endfor
end
```

```
procedure walkCG
entryNode: in, Entry of call graph
NTMap: in, Map of local nesting type
callNTMap: out, Calls × NestingTypes
procNTMap: out, Procedures × NestingTypes
begin
 walkCGOneStep(entryNode, NTMap,
                    callNTMap, procNTMap)
 while there are nodes unannotated do
  if there are some ready nodes then
   select p from the ready nodes
   WalkCGOneStep(p, NTMap, callNTMAP, procNTMap)
  else
   select q from the unannotated nodes
   WalkCGOneStep(q, NTMap, callNTMAP, procNTMap)
  endif
 enddo
end
```

Code 3 Interprocedural nesting type analysis algorithm

A few functions are used in this algorithm. STM_NT() stands for the state transition model mentioned in Fig.3, and it simply uses the STM to determine a new global nesting type from the context and the local nesting type. Function combineNT() is used to combine all the global nesting types of calls to a specific procedure into the procedure's global nesting types. The rules are just simple: if all incoming edges of a node have the same nesting

type, the node inherits it, or the nesting type of this procedure is set to be undetermined.

In procedure WalkCG, nesting type information is propagated in a wavefront way by first selecting ready nodes that are nodes with all incoming edges processed. For all recursive calls, the global nesting types are generally undeterminable, and thus they will have some undetermined incoming edges. By a late selection strategy, the loop in Call Graph is handled in a simple way.

The combination phase is just the same as what the function walkCGOneStep() does. But the range of the considered statements is all interested statements and statement regions rather than calls and OpenMP directives.

Further improvements can be made on the base of this baseline algorithm. A major one is using the condensed call graph other than the detailed call graph. In this case, the work of combineNT() and related iterations can be removed away from the algorithm. By splitting nodes in the call graph, procedure cloning can be implemented to support multi-versioning of the OpenMP translation, and this will be discussed in next section.

2.5 The determinability of directive nesting type

For OpenMP programs written in FORTRAN, the nesting types are highly determinable by utilizing nesting type analysis. Figure 4 gives the result on seven benchmarks out of NPB3.0 OpenMP benchmark[2], which represents a large class of science and engineering applications. For each benchmark, the total OpenMP directive number, the determinable directive number, and the determinable directive number without IPA are given. It shows that the compiling time nesting type analysis can determine the nesting types of all OpenMP directives since the computation structures in these benchmarks are highly regular. The figure also shows that the ability heavily depends on the IPA, since without IPA, only a small amount of direct nesting types can be determined in two of the seven benchmarks (CG and EP).

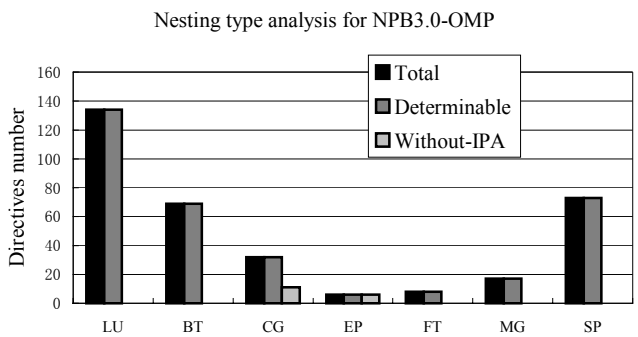

Fig.4 Static determinability of directive nesting types

3 Applications of Global Nesting Type Information

3.1 Nesting type information guided OpenMP translation

The nesting type information obtained by nesting type analysis can be used to determine the translation types in OpenMP translation. Like nesting types, translation types are just a classification of directives in OpenMP translation and optimization modules to produce more efficient codes, e.g., corresponding to the four global nesting types, there can be also four translation types: sequential, top-level, nested and normal translation. They represent different translation and optimization strategies for the same type of directives under different contexts. The relationship between nesting types and these translation types is depicted in Table 1.

Table 1 Map from nesting type to translation type

Nesting type	Translation type
Globally orphaned	Sequential translation
Globally normal	Top level translation
Globally nested	Nested translation
Globally undetermined	Normal translation

A simple example can explain why this translation type classification gets benefits (see Code 4). A piece of OpenMP code skeleton is given in (a), and the four versions of code skeleton after translation according to different translation types are presented in (b)–(e).

```
#pragma omp parallel
{
   some_code();
}
  (a)OpenMP block

/* all directives stripped off,
     executed in sequential mode */
   some_code_after_translation();
  (b)sequential translation

/* executed only by master */
omp_fork(..., thread_func)
thread_func(...)
{
   some_code_after_translation();
}
  (c)translation for top-level group

/* executed by group masters */
omp_fork2(..., thread_func)
thread_func(...)
{
   some_code_after_translation();
}
  (d)nested translation

/* must handle undetermined
     contexts */
if (in_parallel() == 0) {
     omp_fork(...)
}
else {
     omp_fork2(...)
}
  (e)normal translation
```

Code 4 Translation skeletons for different contexts

The benefits of translating directives according to their runtime context mainly come from the elimination of unnecessary code, as shown above in the cases of (b)–(d). In addition, by disambiguating between the top-level thread group and the nested thread groups and translating the code into different sets of runtime calls, the runtime library can visit internal data structures more efficiently. After all, the average OpenMP programs spend most of their execution time in the normal case of only one group of threads. Figure 5 shows the reduction in code size when the OpenMP translation module is presented with the nesting type information for NPB3.0-OMP. The effect is represented as the reduction percentage of the assemble file size, not the target binary file size.

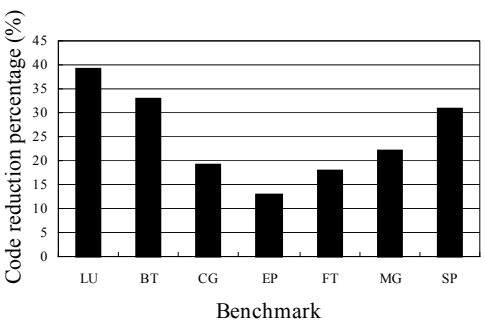

Fig.5 The code size reduction percentage when utilizing nesting type information

It is hard to measure the improvement in performance for the utilization of nesting type analysis, since besides factors inside the OpenMP compiler, a corresponding runtime library should also be built up to carry such kind of comparison. But in code 4, we have shown that by utilizating such kind of information, some stuff code can be eliminated. Moreover, this information may enable further optimizations that are otherwise impossible.

3.2 Nesting type information for the guided OpenMP translation

The dynamic binding and nesting rules enable the programmer to view the parallelization process as a local affair and thus to parallelize the program in an incremental manner. However, in an integrated development environment (IDE), information about a procedure's calling contexts may help to answer questions such as whether this procedure is properly arranged to perform best for all its call sites. A simple tool may simply present the results of the nesting type analysis by highlighting those globally undetermined directives as candidates that may be produced by improper parallelization actions.

More important, since automatical parallelization tools which typically process programs procedure by procedure often introduce calling contexts, the information provided by nesting type analysis is thus more useful for later optimizations.

3.3 Nesting type information for the guided OpenMP translation

In the above discussion, when the calling contexts of the same procedure are not consistent, the global nesting type analysis will set the global nesting type of this procedure to be globally undetermined. However, by node splitting in the call graph, the global nesting type can be explicitly determined. This means the procedure in the original program to be replicated, or the translated procedure to be cloned.

A tradeoff should be made between the code size and the gain from the improved translation strategy. Since the benefit of the latter factor is not easy to model, a set of heuristics may be developed to make decisions such as whether to clone a specific procedure.

4 Related Work

There are several other OpenMP compilers designed in the last five years and available for research community. They are all source-to-source translators, stressing on portability and targeting various goals. OdinMP/CCp[3], which translates C-programs written in OpenMP to POSIX threads, was designed to provide a portable public domain implementation of OpenMP. The translation strategies adopted in OdinMP are rather direct and simple. Nearly no optimization is implemented. PCOMP, the portable compilers for OpenMP[4], implemented upon Polaris, a parallel compiler, is mainly used as a part of a unified parallel compilation systems using OpenMP as the intermediate language. Also, context information is not considered during the translation. NanosCompiler[5], derived from Parafrase-2, concentrates on the implementation of OpenMP extensions to exploit multiple levels of parallelism and automatic work generation. Its tasks are managed totally at runtime, not like the work in this paper. The Omni OpenMP compiler[6] and its descendant[7], as part of the RWCP project, were developed for SMP clusters. Optimizations for software implemented coherence scheme do exist in this compiler[8], but not for the language itself. The compiler, together with the runtime library, simply uses a unified thread model, and doesn't make differences between top level and nested level of thread groups as described in this paper. To summarize these implementations, directive classification and related analysis, like the work described in this paper, are not used in these systems. Moreover, although typically IPA is extensively studied and implemented for optimization compilers that exploit instruction level parallelism, the usage of it in OpenMP compilation is rather new.

Among the above compilers, the Omni OpenMP compiler gets the best performance that is comparable with some commercial compilers[9]. So we use this compiler to justify the implementation of the framework inside ORC2.0[10] by comparing the performances, again, on NPB3.0 OpenMP suite. The experiments are done on a 4-Itanium2-900MHz SMP box, with 4GB memory and running Linux. The performance result is depicted in Fig.6.

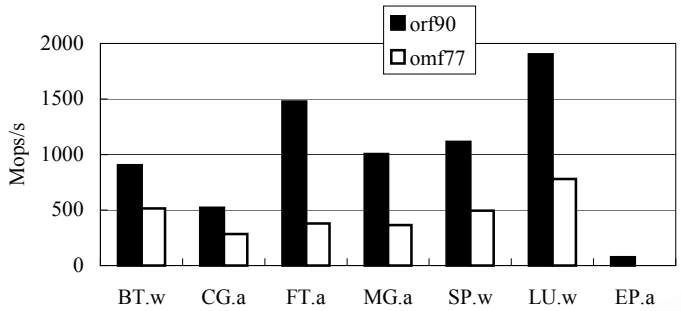

Fig.6 Performance comparison on NPB3.0 OpenMP FORTRAN suite. Mops/s means Mega-operations per second, the metric defined by NPB, and the higher, the better. orf90 is our OpenMP compiler based on ORC2.0[11], and omf77 is Omni OpenMP compiler(1.4a). The individual benchmark name is in a form like BT.w. BT is the application name, and w stands for a problem size. Both with –O3 enabled

5 Summary

In semantics, every parallel region can be viewed as a multithreaded region, and the nesting relationship between these multithreaded regions forms a partial order in the set of all multithreaded regions of a program. It's just part of the execution context of these multithreaded regions. Generally, for compilers that explicitly handle multithreaded programs, such kind of information is critical to perform multithread aware analysis and optimizations. This information can be explicitly expressed as a specific form of the intermediate representation (IR) in the compiler, just like Ref.[12], or as tag information to normal intermediate representation (non-multithread aware), like the work described in this paper.

Nesting type analysis mentioned in this paper tries to use compiling time analysis to collect static information to approximate the true execution context information that we are caring about. Local information is combined together to get the global view about a procedure, or calls to a procedure by an interprocedural analysis, which in fact relies on the common IPA, which does interprocedural control flow analysis to build up the static call graph. The ability to cross procedure boundary empowers the analysis to determine a large amount of directive nesting type in OpenMP programs. However, since the dynamic call graph can not always be constructed at compiling time, theoretically the nesting type analysis can only determine the nesting type of part of these directives.

Although the problem of determining nesting type is raised during OpenMP translation and optimization, it can be viewed as a kind of effort to do thread-aware analysis and optimizations. Indeed, this is just our ongoing work to build up a suitable framework for a thread-aware optimization compiler.

Acknowledgement The work described in this paper is supported by Intel's university research funding, and partly supported by the Gelato project initiated jointly by HP and Intel.

References:

[1] OpenMP Architecture Review Board. OpenMP FORTRAN Application Program Interface Version 2.0, November 2000. OpenMP C and C++ Application Program Interface version 2.0, March 2002. http://www.openmp.org

[2] Jin H, Frumkin M, Yan J. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report, Report NAS-99-011, NASA Ames Research Center, 1999.

[3] Brunschen C, Brorsson M. OdinMP/CCp-a portable implementation of OpenMP for C. Concurrency: Practice and Experience, 2000, 12(12):1193−1203.

[4] Seung JM, Seon WK, Voss M, Sang IL Eigenmann R. Portable compilers for OpenMP. In: Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001). Purdue University, West Lafayette, Indiana, 2001. 11−19.

[5] Ayguade E, Marc G, Labarta J. NanosCompiler: A research platform for OpenMP extensions. In: Proc. of the 1st European Workshop on OpenMP (EWOMP'99). Lund, Sweden, 1999. 27−31.

[6] Sato M, Satoh S, Kusano K, Tanaka Y. Design of OpenMP compiler for an SMP cluster. In: Proc. of the 1st European Workshop on OpenMP (EWOMP'99). Lund, Sweden, 1999. 32−39.

[7] Sato M, Harada H, Hasegawa A. Cluster-Enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. Scientific Programming, 2001,9(2-3):123−130.

[8] Satoh S, Kusano K, Sato M. Compiler optimization techniques for OpenMP programs. Scientific Programming, 2001,9(2-3): 131−142.

[9] Kusano K, Satoh S, Sato M. Performance evaluation of the omni OpenMP compiler. In: Valero M, *et al*. eds. Proc. of the Workshop on OpenMP: Experiences and Implementations (WOMPEI2000). Berlin: Springer Verlag, 2000. 403−414.

[10] Open Research Compiler. http://ipf-orc.sourceforge.net

[11] Chen YJ, Li JJ, Wang SY, Wang DX. ORC-OpenMP: An OpenMP compiler based on ORC. In: Voss M, ed. Proc. of the Int'l Conf. on Computational Science 2004. Berlin: Springer-Verlag, 2004. 414−423.

[12] Tian XM, Bik A, Girkar M, Grey P, Saito H, Su E. Intel OpenMP C++/FORTRAN compiler for Hyper-Threading technology: Implementation and performance. Intel Technology Journal, 2002,6(1):36−46.

****

第 1 届中国分类技术及应用研讨会(CSCA 2005)

征 文 通 知

2005 年 9 月 23−25 日 北京

CSCA 2005 由中国计算机学会人工智能与模式识别专业委员会主办，由北京交通大学承办。分类是知识处理的基本问题，本次会议旨在推动分类技术研究及相关应用的发展，促进相关科技单位和个人的科技合作和学术交流，以及探讨分类与数据分析技术的研究与应用所面临的挑战性问题及关键性研究课题。会议录用论文将由《计算机研究与发展》(正刊，增刊）正式出版，会议还将评选大会优秀论文和研究生优秀论文。我们诚征有关分类和数据分析领域的最新创新性成果，包括分类和数据分析的原理、方法、算法以及特定领域的实际应用等。

征稿范围（不局限于下述范围)：

分类技术基础理论：监督学习，半监督学习，聚类技术，PLS 路径建模和分类，集成分类技术，多标签分类和 Preference 学习，多事例分类，Multimode 聚类和降维，差异性和聚类结构，分类和聚类算法复杂性等

领域相关的分类和聚类技术：数据密集场景中的分类，文本分类和聚类，Web 页面分类和聚类，时间序列的分类和聚类， 图像与视频检索，计算机视觉中的分类，生物特征识别中的分类等

分类技术应用：银行、金融、保险、市场营销、经济分析，商务智能，知识工程，目标识别，生物信息学、生物统计学，医药和健康科学，信息安全等

投稿要求：(1) 论文应是未发表的研究成果，论文要求中文，采用 word 文件排版，论文请参照《计算机研究与发展》网页“作者须知”中的“最终修改稿要求”(http://crad.ict.ac.cn）书写，论文格式参考 2005 年第 1 期执行。(2) 会议论文采用网上提交方式，在提交论文的同时，必须提交一份投稿声明（从 http://crad.ict.ac.cn 网站下载)，作者逐一签字后邮寄或传真到大会会务组，对不提交投稿声明的论文，会议将不予受理。

重要日期：截稿日期 2005-04-25，录用通知日期 2005-05-25，论文提交日期 2005-06-10

来稿请寄：100044 北京交通大学计算机学院 联系人：田凤占

电话：010-51688451，传真：010-51840526，E-mail: fztian@center.njtu.edu.cn