J Supercomput (2015) 71:808–823
DOI 10.1007/s11227-014-1326-3

Automatic scoping of task clauses for the OpenMP tasking model

Chun-Kun Wang · Peng-Sheng Chen

Published online: 2 December 2014

Abstract OpenMP provides an easy-to-learn and powerful programming environment for the development of parallel programs. We propose here an algorithm for the automatic correction of the OpenMP tasking model. Assuming a compiler or programmer has identified task regions in the source programs, the proposed algorithm will automatically generate correct task clauses and synchronization. The proposed algorithm is implemented here based on the ROSE compiler infrastructure; 14 benchmark programs are tested, each of which has had all clauses in the task directives removed for the evaluation. The results of this experimental evaluation show that the proposed technique can successfully generate correct clauses for the tested benchmark programs. The proposed technique can simplify the parallelizing of programs using the OpenMP tasking model, making parallel programming more effective and productive.

Keywords OpenMP · Tasking model · Parallelization · Validation

1 Introduction

Shared-memory multi-core/multi-processor architectures are becoming increasingly mainstream in modern computer systems, and OpenMP is one of the most important

C.-K. Wang
Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi 621, Taiwan
e-mail: amos76530@gmail.com

P.-S. Chen (✉)
Department of Computer Science and Information Engineering, Advanced Institute of Manufacturing for High-tech Innovations, National Chung Cheng University, Chiayi 621, Taiwan
e-mail: pschen@cs.ccu.edu.tw

programming approaches for this architecture [22]. OpenMP supports a simple and flexible programming interface for the development of portable and scalable parallel applications. It consists of compiler directives, library routines, and environment variables for C, C++, and Fortran programs. It also provides programmers with several parallel patterns for writing parallel programs, such as parallel, work-sharing, combined parallel work-sharing, tasking constructs, and synchronization constructs. OpenMP is designed to execute correctly a program in two ways: (1) parallel execution has OpenMP directives enabled and the OpenMP supported library is linked; and (2) sequential execution, during which OpenMP directives are ignored and the OpenMP stub library is linked. An OpenMP-compliant implementation is not required to check issues of data dependencies, data conflicts, race conditions, deadlocks, or improper uses of directives. The use of OpenMP in applications is intended to produce conforming programs. Therefore, even if OpenMP is easy to study and use, much time is often required for the manual step-by-step writing of a correct parallel program.

1.1 OpenMP tasking model

The OpenMP tasking model [6,10] was proposed to allow users to exploit the parallelism of irregular and dynamic program structures, such as unbounded loops, recursive algorithms, and producer–consumer patterns. Figure 1 shows the syntax of the OpenMP task construct. The supported clauses, which control the data-sharing attributes of the variables, are `shared`, `private`, `firstprivate`, and `default`. A task construct is composed of the code to be executed and its data environment. Users need to select task regions and insert proper task constructs to enclose the chosen task regions.

```
/*
The supported clauses which control data-sharing
attributes of variables:
 - if(scalar-expression)
 - final(scalar-expression)
 - untied
 - mergeable
 - shared
 - private
 - firstprivate
 - default(shared | none)
*/
#pragma omp task [clause[[,] clause] ···] newline
   structured-block
```

Fig. 1 Syntax of the OpenMP task construct

```
// Assume A[] and B[] are
// integer arrays.

int num = 0;


while (B[num] <= 1000) {

   if (B[num]%2 == 0)
        do_work1(A[num]);
   else
        do_work2(A[num]);

   num ++;

}
```

(a) Sequential code

```
// Assume A[] and B[] are integer
// arrays.

int num = 0;

#pragma omp parallel
#pragma omp single
while (B[num] <= 1000) {
   if (B[num]%2 == 0)
     #pragma omp task firstprivate(num)
       do_work1(A[num]);
   else
     #pragma omp task firstprivate(num)
       do_work2(A[num]);
   num ++;
}
```

(b) Parallel code by OpenMP tasking model

Fig. 2 Parallelization using the OpenMP tasking model

Figure 2a shows a sequential code fragment and Fig. 2b shows the corresponding parallel code using the OpenMP tasking model. The execution of the parallel code is described as follows. First, a thread encounters the `parallel` directive and then creates a team of threads based on the fork-join model. The `single` directive ensures that only one thread in the team can enter the `single` construct. The other threads in the team will become work threads which are possible candidates for the execution of the generated tasks. When a thread encounters a task construct, it packages the associated structured block and data environment into a task. The thread can immediately execute this task or defer its execution by putting the task into the task pool. The work threads wait until they find tasks in the task pool. Any work thread may pick up the task from the task pool and execute it. Accordingly, all while loop iterations can be quickly screened by one thread, and the parts of the loop body (i.e., the tasks) can be executed by the work threads at the same time.

A task may be temporarily suspended when a thread encounters a task scheduling point. A task scheduling point can be explicitly set by the `barrier`, `taskyield`, and `taskwait` directives, or it can be implicitly decided by some other directives. If a task is always executed by the same thread upon its resumption from the suspended, the task is a tied task. Otherwise, it is an untied task. OpenMP provides `untied` clause to specify these properties. A task will be treated as a tied task if neither clause is specified.

OpenMP also provides the `taskwait` construct to synchronize the execution of tasks and to preserve dependence relationships among tasks. Task synchronization can suspend an encountered task until all child tasks of the current task are completed.

Although the OpenMP tasking model provides useful strategies to parallelize irregular program structures, programmers are required to use proper clauses to describe the data-sharing attributes of the affected variables to obtain correct execution results, even if the task regions have been proper selected. Figure 3a is an example of the improper use of the OpenMP task directives. If the directive is ignored, the program

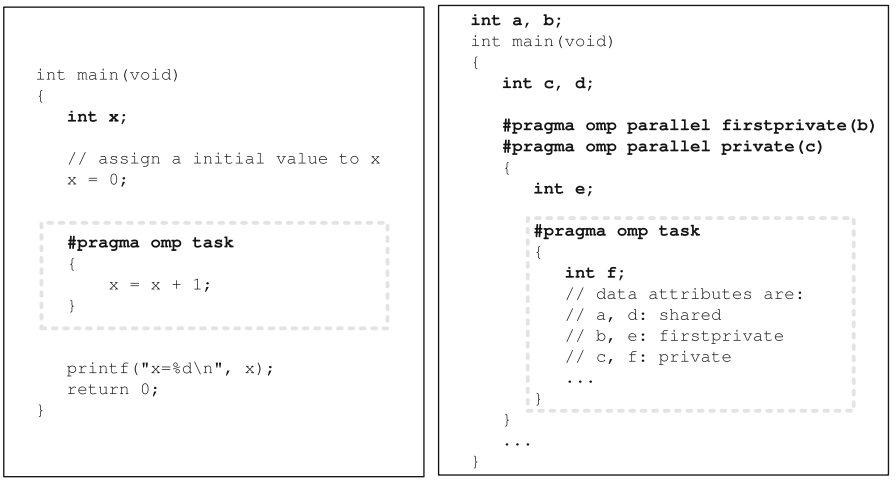

Fig. 3 Examples of using the OpenMP tasking model

is sequentially executed, and we obtain the result: `x` = 1. If the directive is enabled, the program is parallel executed, and we obtain the result: `x` = 0. In this case the reason for the incorrect result is the misunderstanding of the task data scoping. The `task` directive in this program does not describe the attribute of the variable `x`. When the program is executed in parallel by enabling the directive, the value of variable `x` will not be preserved out of the task region because the default data scoping rule `firstprivate` is applied.

Figure 3b shows another example of OpenMP applying implicit rules when data clauses are not provided in an OpenMP directive. The implicit rules are that a global variable will be viewed as a `shared` variable, a variable defined outside an OpenMP task construct will be viewed as a `firstprivate` variable in the task construct, and a variable defined outside an OpenMP parallel construct will be viewed as a `shared` variable in the parallel construct.

Accordingly, the data attributes of `a`, `b`, `c`, `d`, `e`, and `f` in the task region are `shared`, `firstprivate`, `private`, `shared`, `firstprivate`, and `private`, respectively. Implicit rules can make a program difficult to understand and debug. Although OpenMP explicitly defines rules for handling situations in which no data scoping clauses are provided, it is error prone and programmers may have difficulties dealing with the interactions of directives and data scoping, particularly for large and complicated programs.

In this paper, we propose an algorithm for the automatic scoping of task clauses in the OpenMP tasking model. Assuming a compiler or programmer has identified the task regions in the source program, the proposed algorithm will automatically generate correct task clauses and synchronization. The proposed algorithm is implemented based on the ROSE compiler infrastructure [16,25]. The Barcelona OpenMP Task Suite (BOTS) [11], one program from the website [28], and two hand-coded

programs—which have had all clauses removed from the task directives—are used as benchmark programs for testing. The experimental evaluation shows that the proposed technique can successfully correct the tested benchmark programs.

1.2 Contributions

This paper makes the following contributions:

1. **Automatic correction algorithm for OpenMP tasking model.** An algorithm is presented that automatically generates correct task clauses for the OpenMP tasking model. The algorithm also inserts proper task synchronization to preserve data dependence relationships.
2. **Experimental results.** The proposed algorithm was implemented based on the ROSE compiler infrastructure. The previously defined benchmarks are used for the evaluation.
   The experimental results show that our approach can correctly generate clauses and synchronization.

The remainder of this paper is organized as follows. Section 2 describes the proposed algorithm in detail. Section 3 reports the experimental results of an evaluation of the proposed approach, and Sect. 4 discusses related work on automatic parallelization using OpenMP. Finally, the conclusions are presented in Sect. 5.

2 Algorithm

This section describes the proposed algorithm. An OpenMP program can be correctly executed under two situations: parallel execution, during which OpenMP directives are enabled and the OpenMP supported library is linked, and sequential execution, during which OpenMP directives are ignored and the OpenMP stub library is linked. Assuming an OpenMP program uses the OpenMP tasking model, the proposed algorithm aims to generate a corresponding OpenMP program with proper clauses, whose parallel execution gives the same result as its sequential execution. The proposed algorithm does not consider the problem of the association of numeric operations affecting the numeric results during parallel executions.

Two conditions must be satisfied for parallel execution to give the same result as sequential execution: the scope of each variable value in the sequential program must be properly reflected in its parallel version, and data dependence relationships [24,29] in the sequential program must be preserved in the parallel version. These two conditions underpin the development of the proposed algorithm.

For the `task` directive, OpenMP provides three clauses (`shared`, `firstprivate`, and `private`) for the explicit control of the data-sharing attributes of the variables. Figure 4a shows a code skeleton of the `task` directive, which contains three different attributes of the variables using these clauses. Figure 4b correspondingly shows the scopes of the variable values for the example in Fig. 4a. The variable `a` has the attribute `shared`; therefore, its value before entering the task region is visible inside the task, and its value remains visible after exiting the task region. The

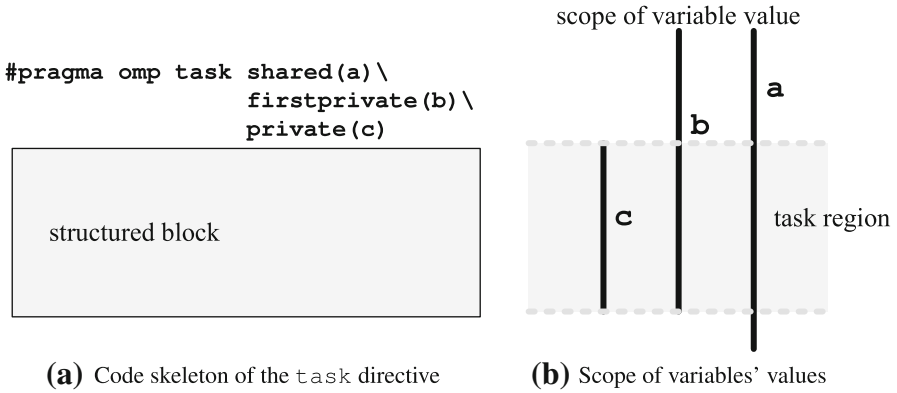

Fig. 4 Code skeleton of the `task` directive and the scope of its variables' values

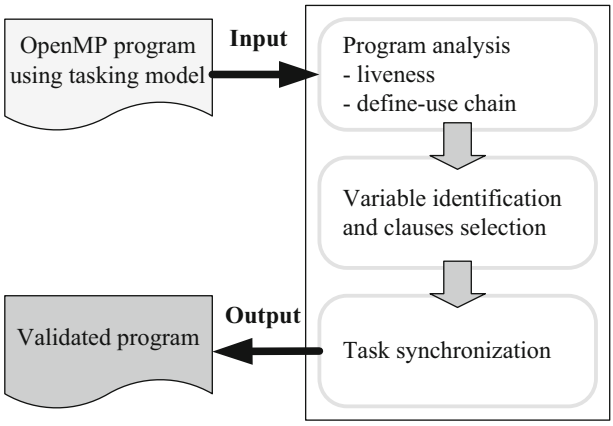

Fig. 5 Algorithm overview

variable `b` has the attribute `firstprivate`; its value before entering the task region is visible inside the task and after exiting the task region, its value is the value that it had before entering the task region. The values of variable `c` inside and outside the task region are unrelated because it has the attribute `private`.

Accordingly, if the attributes of the variables can be analyzed, and the variables can be properly classified based on their attributes, we can know their data-sharing attributes in the OpenMP execution model, and the issue of the scope of the variable value can be overcome. The preservation of data dependence can be achieved by analyzing data dependence relationships and inserting a suitable task synchronization directive.

Figure 5 shows the proposed algorithm. An OpenMP program using a tasking model is the input program. First, classic compiler techniques are used to analyze the program. During the analysis all OpenMP-related directives are ignored and information about which code fragments are tasks as identified by the OpenMP directives is retrieved and fed into the algorithm. Liveness analysis [1,2,20] is used to iden-

tify the data-sharing attributes of the variables for clause selection. The define-use chain [1,2,20] is used to clarify data dependence relationships for task synchronization. Then, according to the analytic results, OpenMP clauses are selected, and task synchronization directives are inserted. Finally, the algorithm will output a validated OpenMP program corresponding to the input program.

2.1 Variable identification and clause selection

The variables that are visible just before their entry to the task region are candidates for liveness analysis. According to the data-sharing attributes in OpenMP, the liveness of variables can be classified into four categories: $\mathbf{live_{inner}}$, $\mathbf{live_{in}}$, $\mathbf{live_{out}}$, and $\mathbf{live_{out\_mod}}$. If a variable lives only within a task, it is classified as $\mathbf{live_{inner}}$. A variable is $\mathbf{live_{in}}$ if it is not used after exiting a task. Variables that are live after exiting a task can be further classified. If the variable is modified within a task, it is a $\mathbf{live_{out\_mod}}$ variable. Otherwise, it is a $\mathbf{live_{out}}$ variable. Figure 6 outlines the different categories. The variable `x` is $\mathbf{live_{out\_mod}}$. The variable `w` is not modified within the task; therefore, it is $\mathbf{live_{out}}$. The variable `y` is $\mathbf{live_{in}}$, and `z` is a $\mathbf{live_{inner}}$ variable.

This classification is properly mapped to the data-sharing clauses of the `task` directive. According to the classification, it is easier to identify systematically the proper data-sharing clauses for each candidate variable. Assuming $\mathbf{V}(t)$ represents the set of all candidate variables for a task t, the data-sharing clauses can be determined by Eq. 1.

A $\mathbf{live_{out}}$ variable can be set to either `shared` or `firstprivate`. Using `firstprivate` has fewer overheads of data synchronization, which is important for the performance. Using `shared` may minimize space and data copying which may be important for large-sized data and embedded systems. Users can select proper clause according to their requirements. In the implementation, a $\mathbf{live_{out}}$ variable is set to `firstprivate`.

In Fig. 6, the variable `x` is set to `shared`, and `w` and `y` are set to `firstprivate`. The variable `z` is described by the clause `private`.

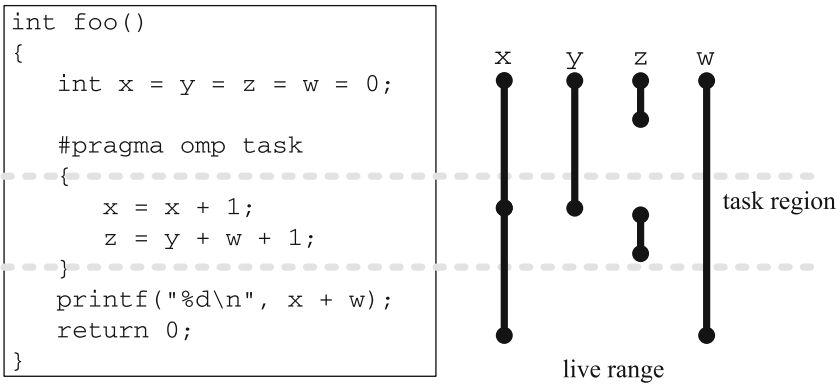

Fig. 6 Variable classification

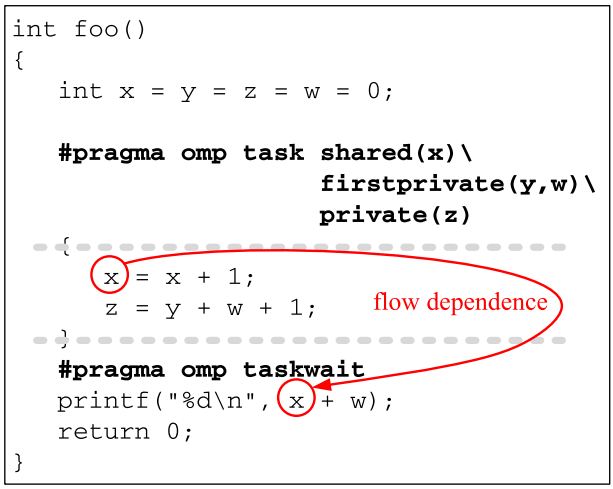

Fig. 7 Example of task synchronization

$$\forall V_i \in \mathbf{V}(\mathrm{t}), \text{the clause of } V_i = \begin{cases} \texttt{shared} & \text{if } V_i \in \mathbf{live}_{\mathbf{out\_mod}}, \\ \texttt{firstprivate} & \text{if } V_i \in (\mathbf{live}_{\mathbf{in}} \cup \mathbf{live}_{\mathbf{out}}), \\ \texttt{private} & \text{otherwise.} \end{cases} \quad (1)$$

2.2 Task synchronization

The `shared` attribute indicates that a variable has data dependence relationships with some places outside a task. To preserve data dependence relationships, the task that modifies a shared variable has to be completed before it is used outside the task. OpenMP provides the task synchronization directive `taskwait`, which suspends an encountered task until its direct child tasks have been completed. In the proposed algorithm, the input program is first analyzed and its define-use chains are constructed to clarify the data dependence relationships. Task synchronization directives are then inserted into locations prior to the use of the program. Due to task synchronization being a barrier to some task executions, postponing synchronization could increase the parallelism of a program and avoid a parallel execution degenerating to a sequential execution. Therefore, task synchronization is delayed until the first dependent variable is encountered. Consider the example in Fig. 6. The variable `x` is re-defined in the task and used outside the task. According to the define-use chain, the first dependent statement is identified, and a task synchronization directive is inserted immediately before it. Figure 7 shows the corresponding example code.

2.3 Tied and untied tasks

OpenMP regards tasks as tied tasks by default. A tied task is always executed by the same thread. However, improvements of performance and the avoidance of dead-

lock may be achievable by making a task untied. Any thread can execute or resume an untied task. OpenMP provides a directive `threadprivate` to replicate variables in which each thread has its own copy. In addition, the execution models of the directive `critical` and the built-in function `omp_get_thread_num()` have direct relationships with the threads themselves. Because an untied task can migrate between threads at any task scheduling point, the use of these thread-centric constructs complicates the behavior of the program and can cause unexpected results. Therefore, the proposed algorithm does not consider the use of these thread-centric constructs. If a program does not use thread-centric constructs, the proposed algorithm can correctly work, regardless of whether tasks are tied or untied.

Algorithm 1 lists the proposed algorithm. Assume *class_dataflow_analysis*() performs liveness analysis and constructs the define-use chain for the input program. For a task T, a candidate variable of T is a variable that is visible at the point just before its entry to the task region. First, the program will be analyzed by classic data-flow analyses. Then, each task is traversed to clarify the relationships between the task itself and outside the task. For each task, proper data-sharing clauses are computed for its candidate variables by their liveness properties. Task synchronization is also inserted, if necessary, according to the dependency relationships obtained from the define-use chain.

Algorithm 1: Algorithm for correcting OpenMP tasking model

```
Input: An OpenMP program using the tasking model
Output: A validated OpenMP program corresponding to the input program
begin
    /*Do classic program analyses: liveness analysis and define-use chain.   */
    classic_dataflow_analysis();
    foreach task T do
        /*Variable identification and clause selection                        */
        foreach candidate variable V of T do
            switch liveness class of V do
                case live_out_mod
                    V_clause ⟵ shared;
                    break;
                case live_out
                    V_clause ⟵ firstprivate;
                    break;
                case live_in
                    V_clause ⟵ firstprivate;
                    break;
                otherwise
                    V_clause ⟵ private;
        /*Task synchronization: using the dependency relationships obtained from the define-use
          chain                                                               */
        Insert task synchronization directives;
end
```

3 Experiment

3.1 Experimental environment

The proposed algorithm was implemented based on the ROSE compiler infrastructure (version 0.95.5a), which was developed at the Lawrence Livermore National Laboratory for building source-to-source program transformation and analysis tools. The liveness, reaching-definition, and alias analyses provided by ROSE are leveraged in our implementation. Table 1 lists the configuration of the experimental environment. Experiments were executed on an Intel 2.67 GHz Corei7 920 processor coupled to 12 GB of RAM and running Ubuntu 10.04. The tested benchmark programs were the two hand-coded programs, one program from [28], and the BOTS 1.1 benchmarks, which each had all clauses in each task directive and all task synchronization directives `taskwait` removed. Table 2 lists the tested benchmark programs and their characteristics. Each benchmark program was fed into our implementation, and then a new OpenMP program was generated to give the same results as the original program, which all directives ignored. The generated OpenMP program was compiled using GCC 4.4.3 with the "–O2" option; and it was then executed.

Table 1 Configuration of the experimental environment

	Item	Value
Hardware	CPU	Intel Core i7 running at 2.67 GHz
	Cache	L1: 64 kB, L2: 1 MB, L3: 8 MB
	Memory	12 GB
Software	Operating system	Ubuntu Linux (kernel version: 2.6.32)
	Native C compiler	GCC 4.4.3 with "–O2" option

Table 2 Benchmark programs

Program	Description	Nested tasks	Source
Pi	Compute the value of π using integrals	Yes	Hand coding
Knight	Find a closed knight's tour on a chessboard	Yes	Hand coding
Gram–Schmidt	Gram–Schmidt algorithm	No	Website [28]
Alignment	Align sequences of proteins	No	BOTS
Fib	Compute Fibonacci numbers	Yes	BOTS
FFT	Compute a Fast Fourier transformation	Yes	BOTS
Floorplan	Compute the optimal placement of cells in a floorplan	Yes	BOTS
Health	Simulate a country health system	Yes	BOTS
N Queens	Find solutions of the N Queens problem	Yes	BOTS
Sort	Uses a mixture of sorting algorithms to sort a vector	Yes	BOTS
Sparselu	Compute the LU factorization of a sparse matrix	No	BOTS
Strassen	Compute a matrix multiply with Strassen's method	Yes	BOTS
UTS	Compute the number of nodes in an unbalanced tree	No	BOTS

We evaluated the proposed algorithm with the Oracle studio compiler (Oracle Solaris Studio 12.3 [23]) which provides an autoscoping feature to automatically determine data sharing attributes. The input to the Oracle studio compiler is the benchmark programs which all clauses in each task directive were replaced with the clause `default(__auto)` to enable the autoscoping feature. Due to that the Oracle studio compiler does not support automatic generating `taskwait`, all the task synchronization directives `taskwait` were also preserved in the inputs.

The results from the generated OpenMP programs, Oracle studio compiler, and the original OpenMP programs are compared to assess the correctness of the generated OpenMP programs. The BOTS benchmarks were verified using their own self-verification methods.

3.2 Results

Table 3 presents the experimental results: the number of tasks, whether containing `taskwait`, and comparison results are listed. The ROSE compiler cannot parse the Alignment, Sparselu, and UTS programs, the modified benchmark programs which preserve tasking skeletons were used as the input for these three benchmark programs, the proposed algorithm can correctly generate the corresponding OpenMP programs. For all the other benchmark programs, the proposed algorithm was able to automatically generate correct OpenMP programs.

Consider the results from the Oracle studio compiler. For the Pi, Knight, Gram–Schmidt, and Fib programs, the Oracle studio compiler cannot generate correct data-sharing attributes in task directives. For the Health, N Queens, Sparselu, and UTS programs, it can successfully identify correct data-sharing attributes. For the remain-

Table 3 Evaluation results for the tested benchmarks

Benchmark	# of tasks	Containing `taskwait`	Comparison result	
			Proposed algorithm	Oracle studio compiler
Pi	1	Y	Correct	Error
Knight	1	Y	Correct	Error
Gram–Schmidt	1	Y	Correct	Error
Alignment	1	N	Parsing error	Fail
Fib	2	Y	Correct	Error
FFT	41	Y	Correct	Fail
Floorplan	1	Y	Correct	Fail
Health	2	Y	Correct	Correct
N Queens	1	Y	Correct	Correct
Sort	9	Y	Correct	Fail
Sparselu	4	N	Parsing error	Correct
Strassen	8	Y	Correct	Fail
UTS	2	Y	Parsing error	Correct

Table 4 Clause comparison

Benchmark	Original version			Proposed algorithm			Oracle studio compiler		
	S	*P*	*F*	*S*	P	*F*	S	P	*F*
Pi	1	0	2	1	0	2	**0**	**1**	**2**
Knight	1	0	1	1	0	1	**0**	**1**	**1**
Gram–Schmidt	4	2	1	2	5	0	**0**	**2**	**5**
Alignment	8	4	5	**4**	**4**	**9**	**8**	**4**	**5**
Fib	2	0	2	2	0	2	**0**	**2**	**2**
FFT	7	236	0	19	0	224	**2**	**0**	**241**
Floorplan	9	3	5	7	3	6	**8**	**1**	**6**
Health	2	0	1	1	1	1	1	0	2
N Queens	0	0	6	0	0	6	0	0	6
Sort	0	0	34	2	0	32	**0**	**0**	**34**
Sparselu	3	0	7	**3**	**0**	**7**	0	0	10
Strassen	4	0	45	17	0	32	**0**	**0**	**49**
UTS	3	0	3	**2**	**0**	**4**	2	0	4

ing programs, the compiler internally showed that autoscoping for some variables was not successful.

Table 4 compares the clauses of the original programs, and those generated by the proposed algorithm and the Oracle studio compiler for the task directives. The predetermined data-sharing attributes are not counted in the table. For the proposed algorithm, the gray cells indicate that the benchmark programs were modified in order to be handled in the ROSE compiler. For the Oracle studio compiler, the bold numbers represent that autoscoping is incorrect or failed. The original FFT and Strassen programs have 7 and 4 `shared` clauses, but the generated programs have 19 and 17 `shared` clauses, respectively. This is due to benchmark programs having several array and pointer variables, which lead to conservative assumptions underlying the analytic results. Although the execution results are correct, the increase of shared variables might hinder the parallelism of the programs. The generated versions of the Alignment, Floorplan, and Health programs have fewer shared variables than the original programs. For the Gram–Schmidt benchmark, the proposed algorithm identified more private variables than the original program, which can reduce the overheads of building task runtime environments. These cases show that the proposed algorithm can generate correct programs while also enhancing performance.

In our implementation, traditional analyses were used to handle array and pointer data types. The analytic results are correct, but very conservative. However, proper handling array is an important issue for automatic scoping. The size and behavior of array in OpenMP programs significantly affects the performance and memory footprints. Some reasonable cost functions should be developed to help compilers to select proper data-sharing attributes. In addition, accurate interprocedural program analyses

can help to aggressively identify data-sharing attributes of variables and minimize `taskwait` to meet the program issues (e.g., performance, code size).

4 Related work

Some issues of automatic scoping in OpenMP are covered in works that automatically parallelize programs using OpenMP [9,12–14] and verify OpenMP programs. These works analyze sequential programs, automatically identify proper code fragments for concurrent execution, and generate the corresponding OpenMP programs. The parallelized code fragments and the generated OpenMP program are controlled by the algorithms. However, in the present study, users specify the parallelized code fragments, and the proposed algorithm generates the correct clauses of the data-sharing attributes.

Liao et al. [17] proposed an algorithm to parallelize high-level abstractions (e.g., STL and complex user-defined class types) in C++ programs. The semantics and behaviors of high-level abstractions were analyzed to enhance the accuracy of compiler analyses. The automatic exploiting parallelism of programs can be improved in the case of more applicable analytic results. This research focuses on the parallelism of array-based computation loops using the OpenMP `parallel for` and `task` constructs.

Müller et al. [21] studied the validation of an OpenMP 2.0 implementation. The validation methodology consisted of a number of routines to test the functionalities of the OpenMP constructs. For each OpenMP construct, the proposed subroutine would return 'true' while the construct executed as expected, and return 'false' otherwise.

The tool VivaMP [15] is a static code analyzer for the verification of OpenMP programs.

Previous studies have considered the automatic scoping of variables in OpenMP programs. Bik et al. [8] proposed an algorithm to find automatically parallel loops, identify the data-sharing attributes of variables, and generate the corresponding multithreaded codes, rather than using OpenMP programs. They leveraged liveness analysis and classic data-flow analyses to ensure the data-sharing attributes of the variables.

Lin et al. [18] proposed several rules for the automatic scoping of variables in parallel regions. Programmers can use the new clauses `AUTO(`*`list-of-variables`*`)` and `DEFAULT(AUTO)` to drive a compiler to determine automatically the data-sharing attributes of variables. Several scoping rules were also proposed for scalar and array variables. To select a proper rule, a compiler needs to analyze the data race conditions in the parallel regions and the data dependence relationships between the scoped variables. If a matching rule is not selected, the binding parallel region will be serially executed. The proposed approach focuses on the constructs of parallel regions, parallel work-sharing, and parallel sections; it was implemented on a Sun Studio 9 Fortran 95 compiler. The proposed approach does not support the OpenMP tasking model.

Voss et al. [27] implemented automatic scoping in the Polaris source-to-source compiler. Similarly to Lin et al. [18], they focused on the Fortran language and implemented the clause `DEFAULT(AUTO)`. A subset of the SPECOMP benchmark pro-

grams [5] was used for evaluation. The authors did not describe in detail the method of identifying the scopes of the variables.

The most closely related to the present work are Oracle Solaris Studio 12.3 [23] and the *Auto-scoping for OpenMP Tasks* proposed by Royuela et al. [26]. The former [23] supports the automatic scoping feature. The compiler can handle scoping for the construct of parallel, work-sharing, parallel sections, and tasking models. The rules from [18] are extended to handle scoping for the construct of parallel, work-sharing, parallel sections, and tasking models. The compiler needs to analyze the data races and read/write behaviors for the scoped variables. If auto-scoping fails, the compiler will give a warning and assign the related code fragments to be serially executed.

Royuela et al. [26] proposed an algorithm to determine automatically the data-sharing attributes of variables for the OpenMP tasking model. For an OpenMP task, the algorithm identifies the code regions that will be concurrently executed with the task. Then the liveness and user-definition analyses are used to analyze the relationships between the variables, the concurrent regions, and the task region. According to the relationships, several rules can be proposed to determine the data-sharing attributes of the variables. A specific graph, a parallel control flow graph, is constructed for the analyses in the proposed algorithm.

The main difference between these two previous works and the present study is that our algorithm does not need to analyze the regions that are executed concurrently with the analyzed task. Instead, a task synchronization directive `taskwait` is used to ensure the data dependence relationships between the concurrent regions and the analyzed task.

The two previous works may not decide the data-sharing attributes of some variables, but the proposed algorithm can identify the data-sharing attribute for each variable.

Moreover, we do not need to construct a parallel control flow graph for the analyses. For our algorithm, the classic control-flow graph and data-flow analyses are sufficient to recognize the data-sharing attributes of the variables.

The previous works may achieve better parallelism, but the present study is simpler and easier to implement. In addition, the data-sharing attributes of variables can be determined using the present study.

5 Conclusion

OpenMP provides an easy-to-learn and powerful programming environment for the development of parallel programs. An OpenMP program also has portability and can be executed in most shared-memory multi-core systems. An open source compiler (GCC) and several major commercial compilers (e.g., Intel C++ compiler, Microsoft Visual C++) support OpenMP.

Compiler-based automatic parallelization plays an important role in enabling effective multi-core processing, and

the parallelization strategies [3,4,7,19] affect the design of algorithms. The use of OpenMP to achieve automatic parallelization has attracted much attention.

There are two issues that are central to automatic parallelization. The first is the assessment of which code fragments in the program should be parallelized. This is related to the parallelization strategy. In most cases, the problem can be solved by profiling the program execution to find its hotspots or critical paths. The second issue is the way in which a compiler automatically inserts the correct OpenMP directives and data-sharing clauses. This problem is closely connected to the program analysis. Accurate program analyses (e.g., alias and data dependence) allow a compiler to identify correctly the variable attributes and to understand program behaviors with less conservative assumptions.

In this paper, we study a possible solution for the second issue. We propose an algorithm for the automatic correction of the OpenMP tasking model. Assuming a compiler or programmers have identified task regions in the source programs, the proposed algorithm will automatically generate the correct task clauses and synchronization. The proposed algorithm was implemented based on the ROSE compiler infrastructure, with 14 benchmark programs—from which all clauses in the task directives had been removed—being tested. The experimental evaluation showed that the proposed technique can successfully generate correct clauses for the tested benchmark programs.

The proposed technique can reduce programmers' burden in parallelizing programs using the OpenMP tasking model and make parallel programming more effective and productive.

Acknowledgments This work was sponsored by the National Science Council of Taiwan under Grants NSC-100-2221-E-194-034-MY2 and 102-2221-E-194-031-MY3. The authors are grateful to the National Center for High-Performance Computing for computer time and facilities.

References

1. Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques, and tools. Addison Wesley
2. Allen R, Kennedy K (2001) Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann
3. Arabnia H, Smith JW (1993) A reconfigurable interconnection network for imaging operations and its implementation using a multi-stage switching box. In: International conference on high performance computing: New Horizons (Alberta, Canada, 1993), pp 349–357
4. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10(2):188–192
5. Aslot V, Domeika MJ, Eigenmann R, Gaertner G, Jones WB, Parady B (2001) Specomp: a new benchmark suite for measuring parallel computer performance. In: Proceedings of the international workshop on OpenMP applications and tools: OpenMP shared memory parallel programming (London, UK, UK, 2001), WOMPAT '01, Springer, pp 1–10
6. Ayguadé E, Copty N, Duran A, Hoeflinger J, Lin Y, Massaioli F, Teruel X, Unnikrishnan P, Zhang G (2009) The design of openmp tasks. IEEE Trans Parallel Distrib Syst 20(3):404–418
7. Bhandarkar SM, Arabnia HR (1995) The refine multiprocessor theoretical properties and algorithms. Parallel Comput 21(11):1783–1805
8. Bik A, Girkar M, Grey P, Tian X (2001) Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. Intel Technol J Q1:9
9. Bondhugula U, Baskaran M, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P (2008) Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: International conference on compiler construction (ETAPS CC)
10. Duran A, Corbalan J, Ayguade E (2008) Evaluation of openmp task scheduling strategies. In: OpenMP in a new era of parallelism (4th IWOMP'08), Eigenmann R, de Supinski BR eds vol 5004 of lecture notes in computer science (LNCS). Springer, New York, West Lafayette, IN, USA, pp 100–110

11. Duran A, Teruel X, Ferrer R, Martorell X, Ayguade E (2009) Barcelona openmp tasks suite: a set of benchmarks targeting the exploitation of task parallelism in openmp. In: Proceedings of 2009 international conference on parallel processing (38th ICPP'09) CD-ROM (Vienna, Austria, Sept. 2009), IEEE Computer Society
12. Intel. Automatic parallelization with intel compilers. http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/. [Online; Accessed 1 July 2011]
13. Jin H, Jost G, Yan J, Ayguade E, Gonzalez M, Martorell X (2003) Automatic multilevel parallelization using openmp. Sci Progr 11(2):177–190
14. Johnson S, Evans E, Jin H, Ierotheou C (2005) The parawise expert assistant - widening accessibility to efficient and scalable tool generated openmp code. In: Proceedings of the 5th international conference on OpenMP applications and tools: shared memory parallel programming with OpenMP (Berlin, 2005), WOMPAT'04, Springer, pp 67–82
15. Karpov A (2009) Peaceful coexistence of pc-lint and vivamp. http://www.viva64.com/en/b/0005/ [Online; Accessed 1 July 2011]
16. Liao C, Quinlan DJ, Panas T, de Supinski BR (2010) A ROSE-based openmp 3.0 research compiler supporting multiple runtime libraries. In: IWOMP, Sato M, Hanawa T, Müller MS, Chapman BM, de Supinski BR eds vol 6132 of lecture notes in computer science, Springer, pp 15–28
17. Liao C, Quinlan DJ, Willcock J, Panas T (2010) Semantic-aware automatic parallelization of modern applications using high-level abstractions. Int J Parallel Program 38(5–6):361–378
18. Lin Y, Terboven C, Mey DA, Copty N (2005) Automatic scoping of variables in parallel regions of an openmp program. In: Proceedings of the 5th international conference on OpenMP applications and tools: shared memory parallel programming with OpenMP, WOMPAT'04, Springer, Berlin, pp 83–97
19. Mattson T, Sanders B, Massingill B (2004) Patterns for parallel programming, 1st edn Addison-Wesley Professional
20. Muchnick S (1997) Advanced compiler design and implementation. Morgan Kaufmann
21. Müller MS, Neytchev P (2003) An openmp validation suite., In: Fifth European workshop on OpenMPAachen University, Germany
22. OpenMP architecture review board (2011) OpenMP application program interface, 3.1 edn. Online available at http://www.openmp.org
23. Oracle (2012) Oracle solaris studio 12.3: OpenMP API USer's Guide. http://docs.oracle.com/cd/E24457_01/html/E21996/, [Online; Accessed 1 Sept 2013]
24. Padua DA, Wolfe MJ (1986) Advanced compiler optimizations for supercomputers. Commun ACM 29(12):1184–1201
25. Quinlan DJ (2000) ROSE: compiler support for object-oriented frameworks. Parallel Process Lett 10(2/3):215–226
26. Royuela S, Duran A, Liao C, Quinlan DJ (2012) Auto-scoping for openmp tasks. In: Proceedings of the 8th international conference on OpenMP in a heterogeneous World, IWOMP'12, Springer, Berlin, pp 29–43
27. Voss M, Chiu E, Chow PMY, Wong C, Yuen K (2005) An evaluation of auto-scoping in openmp. In: Proceedings of the 5th international conference on OpenMP applications and tools: shared memory parallel programming with OpenMP, WOMPAT'04, Springer, Berlin, pp 98–109
28. Website. Programming of parallel computers, assignment 3, gram-schmidt. https://github.com/yohannteston/Parallel-course-Ass3/. [Online; Accessed 1 July 2011]
29. Wolfe MJ (1995) High performance compilers for parallel computing. Addison Wesley, Boston