Generating Task Clauses for OpenMP Programs

Chun-Kun Wang and Peng-Sheng Chen

Department of Computer Science and Information Engineering and
Advanced Institute of Manufacturing for High-tech Innovations
National Chung Cheng university, Chia-Yi, Taiwan, R.O.C.

amos76530@gmail.com pschen@cs.ccu.edu.tw

***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.**

I. Introduction

The OpenMP tasking model [1], [2], [3] 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. In the OpenMP task construct, the data-sharing attributes of the variables contain `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. Figure 1(a) shows a sequential code fragment and Figure 1(b) 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.

```
// 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 ++;

}
```

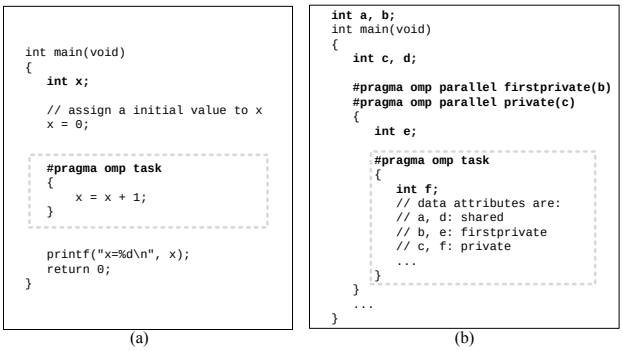
(a) Sequential code

```
// 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. 1. Parallelization using the OpenMP tasking model

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 2(a) is an example of the improper use of the OpenMP task directives. If the directive is ignored, the program 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 2(b) 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`,



(a) (b)

Fig. 2. Examples of using the OpenMP tasking model

`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 [4], [5]. The Barcelona OpenMP Task Suite (BOTS) [6], one program from the website [7], 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.

A. Contributions

This paper makes the following contributions:

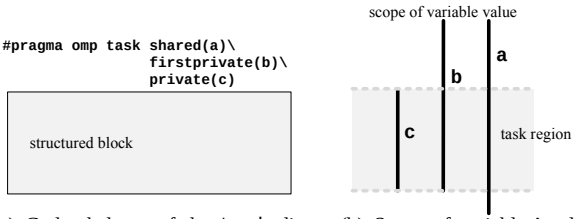
- **Automatic correction algorithm.** An algorithm is presented that automatically generates correct task clauses for the OpenMP tasking model. The algorithm also inserts proper task synchronization to preserve data dependence relationships.
- **Experimental results.** The proposed algorithm was implemented based on the ROSE compiler infrastructure. The tested benchmark programs consist of the BOTS, one program from the website [7], and two hand-coded programs from which all clauses in the task directives are removed. We present experimental results for the tested benchmark programs, which show that our approach can correctly generate clauses and synchronization.

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

II. 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 [8], [9] 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 3(a) shows a code skeleton of the `task` directive, which contains three different attributes of the variables using these clauses. Figure 3(b) correspondingly shows the scopes of the variable values for the example in Figure 3(a). 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 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.



(a) Code skeleton of the `task` directive (b) Scope of variables' values

Fig. 3. Code skeleton of the `task` directive and the scope of its variables' values

Figure 4 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

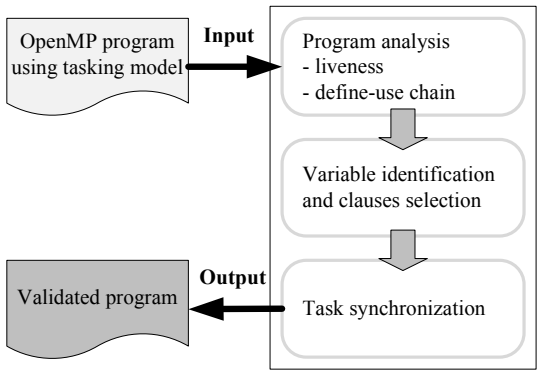


Fig. 4. Algorithm overview

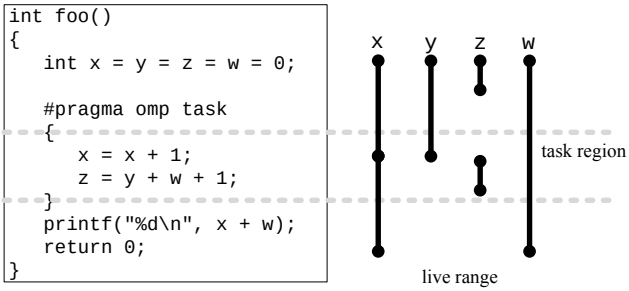


Fig. 5. Variable classification

the algorithm. Liveness analysis [10], [11], [12] is used to identify the data-sharing attributes of the variables for clause selection. The define-use chain [10], [11], [12] 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.

A. 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 5 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 Equation 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

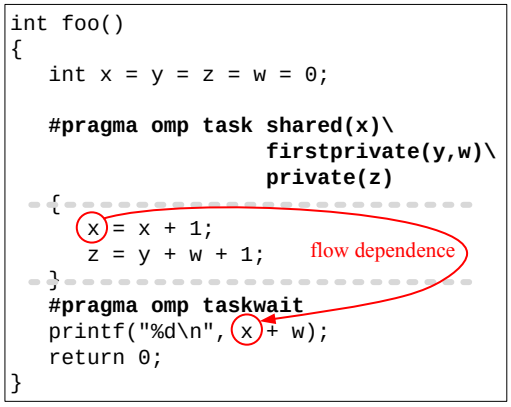


Fig. 6. Example of task synchronization

`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 Figure 5, the variable `x` is set to `shared`, and `w` and `y` are set to `firstprivate`. The variable `z` is described by the clause `private`.

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

B. 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 Figure 5. 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 6 shows the corresponding example code.

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 $\mathbf{live_{out\_mod}}$
          $V_{clause} \longleftarrow shared$;
          break;
        case $\mathbf{live_{out}}$
          $V_{clause} \longleftarrow firstprivate$;
          break;
        case $\mathbf{live_{in}}$
          $V_{clause} \longleftarrow firstprivate$;
          break;
        otherwise
          $V_{clause} \longleftarrow private$;
    `/* Task synchronization: using the dependency relationships obtained from the define-use chain */`
    Insert task synchronization directives;

III. Experiment

A. 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 I 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 [7], and the BOTS 1.1 benchmarks, which each had all clauses in each task directive and all task synchronization directives `taskwait` removed. Table II 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

TABLE I. 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

was compiled using GCC 4.4.3 with the "–O2" option; and it was then executed.

We evaluated the proposed algorithm with the Oracle studio compiler (Oracle Solaris Studio 12.3 [13]) 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.

B. Results

Table III 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 remaining programs, the compiler internally showed that autoscoping for some variables was not successful.

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).

TABLE II. 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 [7]
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

TABLE III. 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

IV. RELATED WORK

Some issues of automatic scoping in OpenMP are covered in works that automatically parallelize programs using OpenMP [14], [15], [16], [17] 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.

Previous studies have considered the automatic scoping of variables in OpenMP programs. Bik et al. [18] 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. [19] 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. [20] implemented automatic scoping in the Polaris source-to-source compiler. Similarly to Lin et al. [19], they focused on the Fortran language and implemented the clause `DEFAULT(AUTO)`. A subset of the SPECOMP benchmark programs [21] 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 [13] and the *Auto-scoping for OpenMP Tasks* proposed by Royuela et al. [22]. The former [13] 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 [19] 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. [22] 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.

V. Conclusion

In this paper, 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.

VI. Acknowledgements

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] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 3.1 ed., July 2011, online available at http://www.openmp.org.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[3] A. Duran, J. Corbalan, and E. Ayguade, "Evaluation of openmp task scheduling strategies," in *OpenMP in a New Era of Parallelism (4th IWOMP'08)*, ser. Lecture Notes in Computer Science (LNCS), R. Eigenmann and B. R. de Supinski, Eds. West Lafayette, IN, USA: Springer-Verlag (New York), May 2008, vol. 5004, pp. 100–110.

[4] D. J. Quinlan, "ROSE: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[5] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-based openmp 3.0 research compiler supporting multiple runtime libraries," in *IWOMP*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 15–28.

[6] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: a set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proc. 2009 International Conference on Parallel Processing (38th ICPP'09) CD-ROM*. Vienna, Austria: IEEE Computer Society, Sep. 2009.

[7] Website, "Programming of parallel computers, assignment 3, gram-schmidt," https://github.com/yohannteston/Parallel-course-Ass3/, [Online; accessed 1-July-2011].

[8] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communication of ACM*, vol. 29, no. 12, Dec. 1986.

[9] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds., 1995.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, January 1986.

[11] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[12] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, October 2001.

[13] Oracle, "Oracle Solaris Studio 12.3: OpenMP API USer's Guide," http://docs.oracle.com/cd/E24457_01/html/E21996/, January 2012, [Online; accessed 1-September-2013].

[14] H. Jin, G. Jost, J. Yan, E. Ayguade, M. Gonzalez, and X. Martorell, "Automatic multilevel parallelization using openmp," *Sci. Program.*, vol. 11, no. 2, pp. 177–190, Apr. 2003.

[15] S. Johnson, E. Evans, H. Jin, and C. Ierotheou, "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*, ser. WOMPAT'04, 2005, pp. 67–82.

[16] Intel, "Automatic parallelization with intel compilers," http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-com [Online; accessed 1-July-2011].

[17] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International Conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

[18] A. Bik, M. Girkar, P. Grey, and X. Tian, "Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems," no. Q1, p. 9, Feb. 2001.

[19] Y. Lin, C. Terboven, D. a. Mey, and N. Copty, "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*, ser. WOMPAT'04, 2005, pp. 83–97.

[20] M. Voss, E. Chiu, P. M. Y. Chow, C. Wong, and K. Yuen, "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*, ser. WOMPAT'04, 2005, pp. 98–109.

[21] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "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*, ser. WOMPAT '01, 2001, pp. 1–10.

[22] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan, "Auto-scoping for openmp tasks," in *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12, 2012, pp. 29–43.