



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Auto-scoping for OpenMP tasks

S. Royuela, A. Duran, C. Liao, D. Quinlan

March 1, 2012

International Workshop on OpenMP (2012)
Rome, Italy
June 11, 2012 through June 13, 2012

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Auto-scoping for OpenMP tasks

Sara Royuela, Alejandro Duran, Chunhua Liao, Daniel J. Quinlan

Lawrence Livermore National Laboratory
{royuelaalcaz1, liao6, dquinlan}@llnl.gov
Barcelona Supercomputing Center
{alex.duran}@bsc.es

Abstract. Different analysis and optimisations have been devised for OpenMP over the years. However, as OpenMP added asynchronous parallelism in the form of tasks several of them need to be revisited. One such analysis is auto-scoping, the process of automatically determine the data-sharing attributes of variables. Auto-scoping relieved the programmer of determining the data-sharing of variables used in worksharing or parallel regions. Based on the previous work for worksharing and parallel regions, we present an auto-scoping algorithm to work with OpenMP tasks. This is a much complex challenge due to the uncertainty of when a task will be executed, which makes harder to determine with what other parts of the program will be executed concurrently. We also present an implementation of the algorithm and results with a number of benchmarks showing that the algorithm is able to correctly scope a large percentage of the variables present in them.

1 Introduction

Parallel programming models play an important role in increasing the productivity of high-performance systems. In this regard, not only performance is necessary but also convenient programmability is valuable to make these models appealing to programmers. OpenMP provides an API with a set of directives that define blocks of code to be executed by multiple threads. This simplicity has been a crucial aspect in the proliferation of OpenMP users.

Each application has its own specific requirements regarding to the parallel model. Loop-centric parallel designs are useful for certain problems where the inherent parallelism relies in bounded iterative constructs. This model becomes useless when dealing with unbounded iterations, recursive algorithms or producer/consumer schemes, adding excessive overhead, redundant synchronizations and therefore, getting poor performance. Because of these limitations, OpenMP has evolved from its previous loop-centric design by defining adaptative parallelism with the concept of explicit asynchronous tasks. Tasks are units of work that may be either deferred or executed immediately. The use of synchronization constructs ensures the completion of all the associated tasks. This model offers better solutions for parallelizing irregular problems as the ones mentioned above. Furthermore, tasks are highly composable since they can appear in parallel regions, worksharings and other tasks.

When using OpenMP task directives, one of the responsibilities left to the programmer is to determine the appropriate data-sharing attributes of the variables used inside the task. According to the OpenMP specification, the data-sharing attributes of the variables referenced in an OpenMP task can be one of **shared**, **private** or **firstprivate**. Although OpenMP defines a default data-sharing attribute for each variable, this might not always be the one that ensures the correctness of the code, thus programmers still need to scope¹ them manually most of the time. Due to the large amount of variables that can potentially appear in each construct, this process is tedious and error-prone. Rules for the automatic scope of variables in OpenMP parallel regions have been presented and tested in the past. However, the process to analyse tasks is quite different due to the uncertainty introduced by tasks regarding to the variable moment when they can be executed. Because of this uncertainty, the challenge is first to determine the regions of code that execute concurrently with a task in order to be able to find out possible race conditions, and then compute how variables are used within the task and outside the task to assign the proper data-sharing attribute according to the information collected.

The contributions of this paper are the following:

- A new algorithm for the automatic discovery of the data-sharing attributes of variables in OpenMP tasks to enhance the programmability of OpenMP. The algorithm determines the code that executes concurrently with a task and the possible race conditions of the variables within the task. Then scopes these variables with the data-sharing that ensures the correctness of the code.
- An implementation of the proposed algorithm in the Mercurium source-to-source compiler and the proof of the benefits of this automatic process with the test of the implementation with several OpenMP task benchmarks. We present the results of the variables that have been automatically scoped and those which the compiler has not been able to determine the scope.

2 Motivation and Related Work

OpenMP data-sharing attributes for variables referenced in a construct can be predetermined, explicitly determined or implicitly determined. *Predetermined* variables are those that, regardless their occurrences, have a data-sharing determined by the OpenMP model. *Explicitly determined* variables are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. *Implicitly determined* variables are those that are referenced in a given construct, do not have predetermined data-sharing attributes and are not listed in a data-sharing attribute clause on the construct (See OpenMP Specifications 3.1 [6] for more details). All variables appearing within a construct have a default data-sharing defined by the OpenMP specifications (either are predetermined or can be implicitly determined); nonetheless, users are duty bound to explicitly scope most of these variables changing the default data-sharing values in order to fulfill the correctness of their codes (i.e. avoiding data race conditions) and enhance their performance (i.e., privatizing shared variables).

¹ In this paper we use the word scope referring to data-sharing attributes

In Listing 1.1 we show a section of code from the *Floorplan* benchmark contained in the BOTS [3]. In this code we find a task within a *for-loop* construct that requires the manual specification of the scope of 15 variables. We aim to improve the programmability of OpenMP by defining a new algorithm that can analyze the access to variables appearing in OpenMP tasks and can automatically define the proper scope of these variables. The compiler is capable of accurately scope the variables by analyzing a) the immediately previous and following synchronization points of the task, b) the accesses done to the variables appearing within the task in all the concurrent codes to the task and c) the liveness of these variables after the task. In the cases the compiler cannot automatically scope a variable, it warns the user to manually do this work. Due to the effectiveness of the algorithm, we prove that the work of the manual scope of variables can be considerably reduced and, therefore, the programmability of OpenMP can be highly improved.

Lin et al. [5] proposed a set of rules that allows the compiler to automatically define the appropriate scope of variables referenced in an OpenMP parallel region. They use a data scope attribute called **AUTO** that activates the automatic discovery of the scope of variables. These rules apply to variables that have not been implicitly scoped, like the index of worksharing *do-loops*. Their algorithm aims to help in the auto-parallelization process but it has some limitations such as:

- It is only applicable to parallel, parallel do, parallel sections and parallel workshare constructs.
- It recognizes OpenMP directives, but not API function calls such as *omp_set_lock* and *omp_unset_lock*, enabling the report of false positives in the data race condition process.
- Their interprocedural analysis and array subscripts analyses are limited. Conservatively, most of the times arrays are scoped as shared while they could be privatized.

They implemented their rules in the Sun Studio 9 Fortran 95 compiler and tested the enhancement in the programmability with the PANTA 3D Navier-Stokes solver. They found that OpenMP required the manual scoping of 1389 variables, rather than the 13 variables that need to be manually scoped using the process of automatic scoping. They proved that the performance obtained by the two versions is the same.

Voss et al. [9] presented an evaluation of auto-scoping in OpenMP based on the work of Lin et al. In this context, they observed significant limitations in the usability of the auto-scoping method for automatic parallelization techniques. Implementing the same **AUTO** data-sharing attribute as Lin et al. did, but in the Polaris parallelizing compiler, they evaluated their implementation with a subset of the SPEC benchmark suite. They revealed that many parts cannot be scoped by the compiler, thus disabling the auto-parallelization of those sections of the program. Their limitations are the same as in the work of Lin et al. [5] since the rules used in the automatic scoping process are the same.

Oracle Solaris Studio 12.2 [7] extends the rules already implemented for the automatic scope of variables in parallel regions to deal with tasks. They define a set of five rules that helps in the automatic scope of variables. However, they do not define an algorithm to find the code concurrent with a task and the way to determine the occurrence of data race conditions. Furthermore, their implementation has several restrictions:

Listing 1.1. Code with OpenMP task from Floorplan BOTS benchmark

```
1 static int add_cell(int id, coor FOOTPRINT, ibrd BOARD, struct cell *CELLS) {
2   int i, j, nn, area, nnc = 0, nnl = 0;
3   ibrd board;
4   coor footprint, NWS[DMAX];
5
6   for (i = 0; i < CELLS[id].n; i++) {
7     nn = starts(id, i, NWS, CELLS);
8     nnl += nn;
9     for (j = 0; j < nn; j++)
10 #pragma omp task untied private(board, footprint, area) \
11 firstprivate(NWS, i, j, id) \
12 shared(FOOTPRINT, BOARD, CELLS, MIN_AREA, MIN_FOOTPRINT, N, BEST_BOARD, nnc)
13 {
14     struct cell cells[N+1];
15     memcpy(cells, CELLS, sizeof(struct cell)*(N+1));
16     cells[id].top = NWS[j][0];
17     cells[id].bot = cells[id].top + cells[id].alt[i][0] - 1;
18     cells[id].lhs = NWS[j][1];
19     cells[id].rhs = cells[id].lhs + cells[id].alt[i][1] - 1;
20     memcpy(board, BOARD, sizeof(ibrd));
21
22     if (!lay_down(id, board, cells))
23       goto _end;
24
25     footprint[0] = max(FOOTPRINT[0], cells[id].bot+1);
26     footprint[1] = max(FOOTPRINT[1], cells[id].rhs+1);
27     area = footprint[0] * footprint[1];
28
29     if (cells[id].next == 0) {
30       if (area < MIN_AREA) {
31 #pragma omp critical
32         if (area < MIN_AREA) {
33           MIN_AREA = area;
34           MIN_FOOTPRINT[0] = footprint[0];
35           MIN_FOOTPRINT[1] = footprint[1];
36           memcpy(BEST_BOARD, board, sizeof(ibrd));
37         }
38       } else if (area < MIN_AREA) {
39 #pragma omp atomic
40         nnc += add_cell(cells[id].next, footprint, board, cells);
41       }
42     }
43 _end;
44 }
45 }
46
47 #pragma omp taskwait
48   bots_number_of_tasks = nnc + nnl;
49 }
```

- The rules are restricted to scalar variables, not dealing with arrays.
- The set of rules is not applicable to global variables.
- The algorithm cannot handle nested tasks or untied tasks.
- It recognizes OpenMP directives, but not API function calls such as *omp_set_lock* and *omp_unset_lock* enabling the report of false positives in the data race condition process.

The analysis of data race conditions in OpenMP programs is needed for many analysis purposes, such as auto-scoping and auto-parallelization. Y. Lin [10] presented a methodology for the static race detection. His method distinguishes between *general*

races (the order of two accesses, where at least one is write, to the same memory location is not enforced by synchronizations) and *data races* (a general race where the access to the memory is not guarded by a critical section). We have based our detection of data race situations in the method presented by Lin, taking account only of data race conditions, because are the only ones that can affect the correctness of OpenMP programs.

We base our work on the increasing need of using asynchronous parallelism and the good results obtained with algorithms that auto-scope variables in OpenMP parallel regions. Since there is no existing work with an exhaustive definition of auto-scoping rules in OpenMP tasks, we define a new algorithm that accurately determines the scope of variables in task regions. The algorithm differs from the previous proposals because it has a methodology based on a parallel control flow graph with synchronizations that discovers all regions executing concurrently with a task. Furthermore, our algorithm takes into account API functions to determine data race situations and it can deal with arrays and global variables.

3 Proposal

Our proposal is to extend the clause **default** used with OpenMP task directive in order to accept the keyword **AUTO**. The clause **default (AUTO)** attached to a **task** construct will launch the automatic discovery of the scope of variables in that task. In Algorithm 1 we present the high-level description of the proposed algorithm. For each variable referenced inside the task region, which is not local to the task, the algorithm returns one of the following results:

- UNDEFINED: The algorithm is not able to determine the behavior of a variable. This variable will be reported to the user to be manually scoped.
- PRIVATE: The variable is to be scoped as **private**.
- FIRSTPRIVATE: The variable is to be scoped as **firstprivate**.
- SHARED: The variable is to be scoped as **shared**.
- SHARED_OR_FIRSTPRIVATE: The variable can be scoped as either **shared** or **firstprivate** without altering the correctness of the results. It is an implementation decision to scope them as SHARED or FIRSTPRIVATE.

Algorithm 1 High-level description of the auto-scoping algorithm for OpenMP tasks

1. Define the regions of code that execute concurrently with a given task. This regions are defined by the immediately previous and following synchronizations of the task and belong to:
 - Other tasks scheduled in the region described above.
 - Other instances of the task if it is scheduled within a loop or in a parallel region.
 - Code from the parent task between the task scheduling point and the synchronization of the task.
2. Scope the variables within the task depending on the use of these variables in all regions detected in the previous step and the liveness properties of the variables after the execution of the task.

The algorithm works under the hypothesis that the input code is correct and that the input code comes from an original sequential code that has been parallelized with OpenMP. Based on this, suppose the analysis of a task t , then the algorithm will proceed as it is shown in the Algorithm 2.

Algorithm 2 Detailed algorithm for the auto-scoping of OpenMP tasks

1. Determine the regions of code that execute concurrently with t , referred to as *concurrent regions* in this paper. These regions can be other tasks, other instances of the same task and code from the parent task. In order to do that, we first define the following points:

Scheduling The task scheduling point of t as defined in the OpenMP specification. Any previous code in the parent task is already executed before the task starts its execution.

Next_sync The point where t is either implicitly or explicitly synchronized with other tasks in execution. Any code after this point will be executed after the completion of t .

Last_sync The immediately previous synchronization point to the *Scheduling* point of t . If this synchronization is a **taskwait**, then we take into account the previous nested tasks because they may not be finished.

With these points, we can define the concurrent regions to be the following:

–The region of code of the parent task that runs concurrently with t , bounded by *Scheduling* and *Next_sync*.

–The regions defined by the tasks that run concurrently with t , bounded by *Last_sync* and *Next_sync*. Other instances t' of t are concurrent with t when t is scheduled within a parallel construct or within a loop construct.

2. For each variable s , that is a scalar, appearing within t , apply the following rules in order:

(a) If s is a parameter passed by reference or by address in a call to a function that we do not have access to, then s is scoped as UNDEFINED. s is also scoped as UNDEFINED if it is a global variable and t contains a call to a function that we do not have access to.

(b) If s is not used in the concurrent regions, then:

i. If s is only read within t , then s is scoped as SHARED_OR_FIRSTPRIVATE.

ii. If s is written within t , then:

A. If s is a global variable and/or s is alive after the *exit* of t (that means after the point where t reaches a synchronization point), then s is scoped as SHARED.

B. If s is dead after the *exit* of the task, then:

–If the first action performed in s is a write, then s is scoped as PRIVATE.

–If the first action performed in s is a read, then s is scoped as SHARED_OR_FIRSTPRIVATE.

(c) If s is used in the concurrent regions, then:

i. If s is only read in both the concurrent regions and within the task, then s is scoped as SHARED_OR_FIRSTPRIVATE.

- ii.If s is written either in the concurrent regions or within the task, then we look for data race conditions (data race analysis specifics are explained at the end of the algorithm). Thus,
 - A.If we can assure that no data race can occur, then s is scoped as SHARED.
 - B.If a data race can occur, then s is scoped as RACE.
- 3.For each variable a , that is an array, appearing within t
 - (a)For each use of $a_i, i \in [0..N]$ of the array variable, where N is the number of uses of a or a section of a appearing within t , apply the same methodology used for scalars.
 - (b)Since OpenMP does not allow different scopes for the subparts of a variable, we need to mix the results get in the previous step following the rules below:
 - i.If all $a_i, i \in [0..N]$ have the same scope sc , then a is scoped as sc .
 - ii.If there are different regions of the array with different scopes, then:
 - A.If some a_i has been scoped as UNDEFINED or some a_i is scoped as RACE and some a_j where $i \llcorner j$ is scoped as SHARED, then a is scoped as UNDEFINED.
 - B.If at least one a_i is FIRSTPRIVATE and all $a_j, j \in [0..N]$ where $j \llcorner i$ are PRIVATE, SHARED_OR_FIRSTPRIVATE or RACE, then a is scoped as FIRSTPRIVATE.
 - C.If at least one a_i is TPRIVATE and all $a_j, j \in [0..N]$ where $j \llcorner i$ are RACE, then a is scoped as PRIVATE.
 - D.If at least one a_i is SHARED, and all $a_j, j \in [0..N]$ where $j \llcorner i$ are PRIVATE, FIRSTPRIVATE or SHARED_OR_FIRSTPRIVATE, then, based on the hypothesis that the input code is a parallelized version of a sequential code and fulfilling the sequential consistency rules, a is scoped as SHARED.
- 4.For each v scalar or array variable that has been scoped as RACE in the previous steps and, based on the hypothesis that the input code is correct, we privatize the variable as otherwise a synchronization would have existed to avoid the race condition. Therefore,
 - If the first action performed in v (or some part of v if v is an array) within the task is a write, then v is scoped as PRIVATE.
 - If the first action performed in v (or all parts of v if v is an array) within the task is a read, then v is scoped as FIRSTPRIVATE.

Data Race conditions In the previous algorithm we need to analyze if the variables within a task are in a data race situation. Data race conditions can appear when two threads can access to the same memory unit at the same time and at least one of these accesses is a write. To determine data race conditions, we have to analyze the code appearing in all the concurrent regions and the task. Any variable appearing in two of these regions where, at least one of the accesses is a write and none of the two accesses is protected by either and atomic construct, a critical construct or a lock routine (omp_set_lock / omp_unset_lock), can trigger a data race situation. Under the assumption we make that the code is correct and, since OpenMP defines an unexpected behavior for the algorithms containing race conditions, the variables scoped as RACE are privatized. Depending on the use made of the variable within the task it will be PRIVATE (when it is first written) or FIRSTPRIVATE (when it is first read).

Limitations The algorithm has two limitations: it does not deal with aggregates and, when the tasks contain calls to functions that are not accessible at compile time, then all variables that can be involved in this functions cannot be scoped. This variables are global variables and parameters to the function that are addresses or passed by reference. Regarding to the implementation, the compiler might be unable to determine the previous synchronization point, *Last_sync*, and/or the next synchronization point, *Next_sync* (i.e., the point belongs to a function that calls the function where the analyzed task is scheduled); in these cases only the variables that are local to the function (including its parameters) where the task is scheduled can be automatically scoped. The rest of variables must be scoped as UNDEFINED and reported to the user to be manually scoped.

Strengths The algorithm is perfectly accurate, so it never results on false positives and the reported results are always correct. Specific rules are defined for determine the cases when the algorithm cannot define an specific data-sharing so these variables can be reported back to the user. The methodology we use to determine the regions of code that run concurrently with a task, based on the definition of the synchronization points of the task, models an algorithm that is insensitive to the scheduling policy used in runtime.

Example In the code presented in Listing 1.1 a task is defined within a loop construct. The algorithm proposed will compute the following result for the variables appearing within the task: variable *cells* does not need to be scoped because it is local to the task; global variables *N*, *MIN_AREA*, *MIN_FOOTPRINT* and *BEST_BOARD* are scoped as UNDEFINED due to the occurrence of the system call *memcpy* and because that we do not have access to the code of this function; the same happens to the parameter *board*, passed by reference to *memcpy*; variable *nnc* is scoped as SHARED because it is written within the task, it cannot produce a data race because the access is protected in an atomic construct and its value is alive at the exit of the task; variables *area* and *footprint* are PRIVATE because the algorithm detects a race condition (different instances of the task can write to the variable at the same time) and their values are written without being read; variables *i*, *j*, *id*, *BOARD*, *CELLS*, *FOOTPRINT* and *NWS* are scoped as SHARED_OR_FIRSTPRIVATE because they are only read.

4 Implementation

Mercurium [2] is a source-to-source compiler for C/C++ and Fortran that has a common internal representation for the three languages. The compiler defines a pipeline of phases that transform the input source. We have implemented the algorithm presented in Section 3 within a new phase along with other analyses that are required for the computation of the scope such as *control flow analysis*, *use-definition chains* and *liveness analysis*. We define a parallel control flow graph (PCFG) [8] with edges connecting 1) the scheduling point of the task with the task entry and 2) the task exit with the synchronization point that synchronizes the task with its parent (i.e., a *taskwait*) or with the threads of the team (i.e., a *barrier*). In Fig. 1 we show a code scheme with different

tasks and synchronization points. In Fig. 2 we show a simplified version of the resultant PCFG. Tasks are linked with special edges representing synchronization because they are analyzed distinctly. We take into account nested tasks (including nesting due to recursive functions) and the semantics of the different synchronization points when we connect the tasks in the PCFG: a barrier synchronizes any previously scheduled task whereas a taskwait synchronizes just the previous tasks that are scheduled by the encountered task of the taskwait (i.e., previous child tasks).

```

// section 1
#pragma omp task
{
  // task A code
  #pragma omp task
  {
    // task B code
  }
  // task A code
}
// section 2
#pragma omp taskwait
// section 3
#pragma omp task
{
  // task C code
}
// section 4
#pragma omp task
{
  // task D code
}
// section 5
#pragma omp barrier
// section 6

```

Fig. 1. Code scheme with tasks

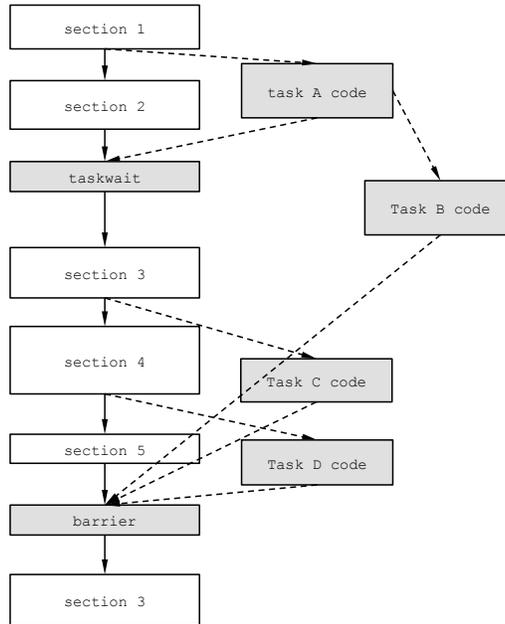


Fig. 2. Abstraction of the PCFG used during the auto-scoping that shows the connections of the tasks in Fig. 1

In Fig. 3 we show the flow chart with the analyses performed in the compiler in order to have enough information to compute the scope of variables. The steps we follow are showed below:

1. Create a PCFG as it is defined previously.
2. Compute the Use-Definition chains: the sets of variables that are read before than defined (*UPPER_EXPOSED*) and the variables that are defined (*KILLED*) in each node; variables that have an undefined behavior are classified as *UNDEFINED* (Function calls to methods that are not defined in the same file as the function being analyzed will cause global variables and the parameters passed by reference or with pointer type from the function to have an unexpected behavior).

3. With this information, we then perform Liveness analysis. This analysis computes the sets of variables that are alive at the entry of each node (*LIVE_IN*) and the variables that are alive at the exit of each node (*LIVE_OUT*).
4. Apply the algorithm showed in Section 3 for the automatic discovery of the scope of variables in OpenMP tasks. In our implementation we have decided to further specify the variables scoped as *SHARED_OR_FIRSTPRIVATE* as follows:
 - All scalar variables are defined as *FIRSTPRIVATE* because the cost of the privatization should be comparable to the cost of one access to a shared variable. In the worst case (*s* is only used once), we do not loose performance and, in the best case (*s* is used more than once) we improve the performance.
 - All array variables are defined as *SHARED* because the cost of privatizing an array is usually high. Only in cases when the positions of the array are accessed many times may be advantageous to privatize the array.

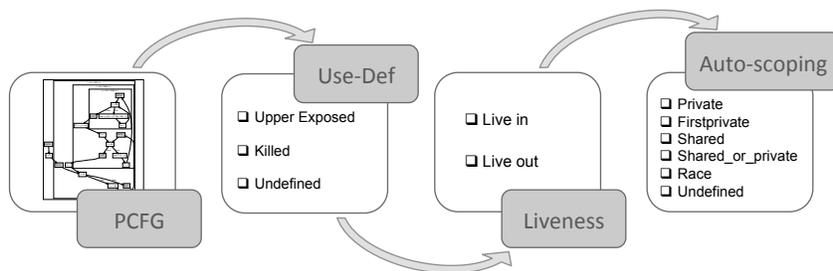


Fig. 3. Flow chart of the Mercurium analyses used in the Auto-Scoping process

5 Evaluation

For the evaluation of the proposed algorithm, we have used the Barcelona OpenMP Tasks Suite [3] (BOTS) and other benchmarks, all developed in the Barcelona Computing Center. Table 1 presents the benchmarks used in our evaluation. Since we want to evaluate the enhancement of programmability, we have computed the number of variables that have been automatically scoped. A summary of the results is shown in Table 2. The table shows the amount of variables that have been automatically scoped organized depending on their scope, the amount of variables that the algorithm has been unable to scope and the percentage of successfulness for each benchmark.

The overall result is that a significant amount of variables, more than the 70%, can be automatically and correctly scoped by our algorithm. Most of the cases where some variables cannot be automatically scoped correspond to global variables in benchmarks where either the tasks contain calls to functions without accessible source code or the previous synchronization point of the tasks cannot be specified; therefore, the compiler cannot determine the accesses done to variables defined in an outer scope to the scope of the function that schedules the task (most of them global variables). In the case of the *Health* benchmark, we obtain a poor result because the variables used in tasks are aggregates and the algorithm does not yet deal with this kind of variables. In some cases

Benchmark	Description
Alignment	Dynamic programming algorithm that aligns sequences of proteins.
FFT	Spectral method that computes the Fast Fourier Transformation.
Fib	Recursive version of the Fibonacci numbers computation.
Health	Simulation method for a country health system.
Floorplan	Optimization algorithm for the optimal placement of cells in a floor plan.
NQueens	Search algorithm that finds solutions for the N Queens problem.
Sort	Integer sorting algorithm that uses a mixture of sorting algorithms to sort a vector.
SparseLU	Linear algebra algorithm that computes the LU factorization of a sparse matrix.
UTS	Search algorithm that computes the number of nodes in an Unbalanced Tree.
Stencil	Stencil algorithm over a matrix structure.
Cholesky	Linear algebra algorithm that computes the Cholesky decomposition of a matrix.

Table 1. Short description of the benchmarks used in the evaluation

as in the *Alignment* or the *Cholesky* benchmarks, the algorithm is able to find data race conditions and privatize the variables; or, as in the case of the *Floorplan* benchmark, the algorithm dismiss data races, and keeps variables as shared.

	SHARED	PRIVATE	FIRSTPRIVATE	UNDEF	(%) success
Alignment	0	4	5	12	42.86%
FFT	5	0	249	96	72.57%
Fib	2	0	2	0	100.00%
Health	0	0	1	2	33.00%
Floorplan	2	1	3	7	46.15%
NQueens	0	0	5	0	100.00%
Sort	0	0	26	9	74.29%
SparseLU	0	0	7	3	70.00%
UTS	2	1	2	2	71.43%
Stencil	0	0	3	1	75%
Cholesky	0	0	16	0	100.00%
TOTAL	11	6	319	132	71.79%

Table 2. Results from the automatic scoping algorithm for different benchmarks

We have seen that the main difficulty when trying to automatically define the data-sharing of variables are the global variables. We have to improve the implementation in order to be able to define the previous and next synchronization points in some cases. This means to inline the PCFG to have enough context information. If we are able to define these two points, all variables that in the current results are undefined (but the case of the *Health* benchmark) will be automatically scoped.

6 Conclusions and Future work

We have developed an automatic mechanism to improve the programmability of OpenMP by relieving the programmer from the tedious work of manually scoping variables within tasks. The mechanism consists of a new algorithm based on compiler analyses such as use-definition chains and liveness analysis, and the OpenMP synchronization points. This algorithm scopes automatically the variables appearing in OpenMP tasks with the use of the clause `default(AUTO)`. We have proved the benefits of this new method implementing the algorithm in the Mercurium compiler and testing it with a set of benchmarks. Our results show the majority of variables can be scoped by the compiler. The variables that cannot be automatically scoped are reported to the user to proceed to manual scoping.

In the future we want to extend the algorithm presented in this paper in order to deal with aggregates. We also plan to implement the algorithm in the ROSE [4] source-to-source compiler and take advantage of its features of dealing with multiple files and system calls. We want to enable the compiler to recognize the most common system calls, such as memory allocation methods, in order to avoid the incapacity of auto-scoping global variables and address parameters in the occurrence of such calls. The automatic scoping of variables is part of the solution to other problems of compiler analyses and optimizations. This analysis can, for example, lead us to enhance auto-parallelizing tools and help with OpenMP correctness tools. We plan to use the auto-scoping analysis to automatically define data-dependencies between tasks[1].

References

1. A. Duran and E. Ayguadé and R.M. Badia and J. Labarta and L. Martinell and X. Martorell and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.
2. Barcelona Supercomputing Center. The NANOS Group Site: The Mercurium Compiler. <http://nanos.ac.upc.edu/mcxx>.
3. A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page 124–131, Vienna, Austria, September 2009. IEEE Computer Society.
4. D. Quinlan et al. Rose compiler infrastructure. <http://http://rosecompiler.org>.
5. Y. Lin, C. Terboven, D. an Mey, and N. Copty. Automatic Scoping of Variables in Parallel Regions of an OpenMP Program. In Barbara M. Chapman, editor, *WOMPAT*, volume 3349 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2004.
6. OpenMP ARB. OpenMP Application Program Interface, v. 3.1, September 2011.
7. Oracle. Oracle Solaris Studio 12.2: OpenMP API User’s Guide, 2010.
8. S. Royuela. Compiler Analysis and its Application to OmpSs. Master’s thesis, Technical University of Catalonia, 1012.
9. M. Voss, E. Chiu, P.M.Y. Chow, C. Wong, and K. Yuen. An Evaluation of Auto-Scoping in OpenMP. In Barbara M. Chapman, editor, *WOMPAT*, volume 3349 of *Lecture Notes in Computer Science*, pages 98–109. Springer, 2004.
10. Y. Lin. Static Nonconcurrency Analysis of OpenMP Programs. In M.S. Müller, B. Chapman, B. de Supinski, A. Malony, and M. Voss, editors, *IWOMP*, volume 4315 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.