Revisiting OpenMP Auto-Scoping Rules

Vassilios V. Dimakopoulos
Department of Computer Science and Engineering,
University of Ioannina
P.O. Box 1186, Ioannina, GR-45110, Greece

Agelos Mourelis
BETA CAE Systems SA,
Kato Scholari,
Thessaloniki, GR-57500 Epanomi, Greece

Abstract

Auto-scoping in OpenMP has been proposed as a means for relieving the programmer from the non-trivial effort of identifying the data sharing attributes of variables used within code regions that produce concurrency, such as `parallel` and `task` constructs. In this work we reconsider autoscoping on `parallel` constructs, including combined `parallel`-worksharing constructs. We first show that the current implementations do not always scope variables correctly in the presence of *nested* parallel constructs. We then proceed to extend the set of rules that guide the autoscoping decisions so as to handle nested constructs successfully. We also discuss how this functionality is implemented in the OMPi source-to-source OpenMP compiler.

1 Introduction

OpenMP [1] is undeniably one of the most successful parallel programming models. Through a relatively simple and flexible programming interface it facilitates the development of portable and scalable parallel applications. It consists mainly of compiler directives and a set of runtime library routines, supporting the C, C++, and Fortran languages.

Over the years, the applicability of OpenMP has expanded from regular, loop-based applications, to more complex, task-based applications, which may exhibit recursive and/or irregular parallelism. More recently, OpenMP has entered the realm of accelerators, harnessing their power through its device interface and related constructs.

Key to the success of OpenMP is the fact that it allows for incremental parallelization of existing sequential programs. Typically, one starts with existing sequential code and gradually adds parallelization directives, parametrizing them with specific clauses until the desired performance behavior is achieved. The "catch" is that an OpenMP compiler does not have to check the correctness of the program code. Assuring that there will be no data dependencies, race conditions, deadlocks, etc. is the programmer's responsibility. As a result, while it may be easy to write an OpenMP program, it is also easy to write a *wrong* OpenMP program.

A crucial part in the parallelization of sequential code is identifying the data-sharing attributes of the involved variables, i.e. determine whether they should be shared among the team of threads that will execute the parallelized regions, or privatized and in what way. Realizing that this process can be non-trivial and error prone, especially if there exists a large number of variables, Liu, Terboven, Mey and Copty [2] proposed *autoscoping*, relieving the programmer of the burden: the compiler analyzes the region in question and automatically assigns data-sharing attributes to variables.

The motivation of our work is somewhat different but winds up to the same problem. In particular, we have observed that scoping errors are quite frequent especially for beginning or casual OpenMP users. They can also be found among the most common mistakes in OpenMP [3]. It would be thus beneficial if the compiler analyzed the manually scoped variables and warned the programmer of any inconsistencies among the data-sharing clauses and the usage patterns of the involved variables. In fact this is one of the programmer-centric mechanisms we are introducing in our OMPi source-to-source OpenMP compiler [4] in order to assist developers with their code.

Nested parallelism is an important feature of OpenMP since its conception. The specifications allow existing threads to create their own teams of threads at arbitrary depths and include facilities to control them. This is because in various application scenarios, nested parallelism may exploit the multiplicity of processing elements beyond what is possible with a single level of parallelism. This is particularly relevant today where contemporary systems pack a large number of cores. Whether there is not enough outer-loop parallelism or there exist load balance problems in parts of a larger parallel region, nested parallelism in an invaluable tool if it is used properly.

However, we have found that the implementation of autoscoping with the rules set in [2] does not always produce correct results in the presence of lexically nested parallel regions. We thus provide simple additions to the autoscoping rules so as to guarantee they are applied correctly in such situations.

In this work:

- We review the autoscoping state-of-the-art for OpenMP programs.
- We augment the original autoscoping rules for parallel regions and provide additional rules so that they work correctly for nested parallel regions.
- We present their application within the OMPi compiler.

The rest of the paper is organized as follows: Section 2 reviews related work; it also summarizes the data-sharing attribute jargon of OpenMP and the original autoscoping rules for parallelized regions. Section 3 proposes a new rule, expanding the autoscoping domain while in Section 4 we present additional rules that allow the original ones to be applicable to nested parallelism. Section 5 discusses implementation details in the OMPi source-to-source OpenMP compiler. A concrete application is used to demonstrate the effectiveness of our new autoscoping rules in Section 6. Finally, Section 7 concludes the paper.

2 OpenMP data sharing attributes and autoscoping

According to the OpenMP specifications [1], the data-sharing attributes of variables that are referenced in a given construct can be *predetermined*, *explicitly determined*, or *implicitly determined* as follows:

- Predetermined variables have a specific data-sharing attribute which is dictated by the OpenMP API for the particular type of construct. For example, worksharing loop iteration variables are always *private*.
- Explicitly determined variables are the ones the programmer specifies through the data-sharing attribute clauses of the construct, such as `private()`, `firstprivate()`, `shared()` etc.
- Implicitly determined variables are those that, while used in a construct, do not have predetermined data-sharing attributes and are not listed in a data-sharing attribute clause. The OpenMP specifications have rules for specific types of constructs which define how to scope such variables. For example, in a `parallel` construct, if there is no `default()` clause present, these variables are *shared*.

OpenMP specifications always assign a default data-sharing attribute (either predetermined or implicitly determined) for all variables appearing in a construct. However, this default value is rarely enough to guarantee correctness and/or performance; consequently the programmer is obliged to explicitly scope (i.e. dictate the data-sharing attributes of) most of the variables used in a construct. For example, a variable could be scoped as *private* for performance reasons, or for eliminating race conditions.

Explicitly scoping constructs with a large number of variables can be quite cumbersome and error-prone. Liu, Terboven, Mey and Copty [2] proposed *autoscoping*, that is a set of rules which allow the compiler to automatically discover and define the appropriate data-sharing attributes of variables referenced in a `parallel` region. There are two ways to activate the mechanism: one is to use the clause `default(__auto)` which auto-scopes all implicitly determined variables; the other is to list the variables that the compiler should scope automatically within an `__auto()` clause. Autoscoping is implemented in the Oracle Developer Studio [5].

The advent of version 3.0 of the OpenMP specifications brought a significant new feature: tasks. Since then, all related works have targeted autoscoping for tasking regions. Five new rules were added to support tasks in the Oracle Developer Studio [5], extending the rules already implemented for automatically scoping variables in `parallel` regions.

Royuela et al. [6] proposed an algorithm to determine automatically the data-sharing attributes of variables for `task` regions, with an improved accuracy over the scheme in the Oracle Developer Studio. The algorithm uses control flow graphs, liveness and use-def analyses to identify the code regions that can be concurrently executed with a given `task`, and determines the relationships between the involved variables. According to the relationships, specific rules are used to scope the variables.

Wang and Cheng [7] proposed a simpler scheme, which does not need to consider the regions that are executed concurrently with the analyzed task. It offers 100% accuracy, but at the cost of inserting `taskwait` synchronization directives at appropriate points to ensure the data dependence relationships between the concurrent regions and the task.

Finally, Munera et al. [8] consider autoscoping for the tasks of the OmpSs-2 model. They draw from the work of Royuela et al. [6], adapting it to the particularities of their model.

Autoscoping is used for classifying variables, as an essential step when auto-parallelizing sequential code [9]. As mentioned in the introduction, our motivation is not to perform auto-parallelization but to assist OpenMP programmers by warning them for possible scoping mistakes. We consider this an important facility, especially nowadays where the OpenMP API has expanded substantially and has become quite complex. The `parallel` and combined `parallel`-worksharing constructs we target are among the most frequently used ones [10]. It is important for OpenMP toolchains to help and guide developers so as to avoid common mistakes and pitfalls. For our purposes here, a compiler should be in a position to understand the use of each variable, even if it is not explicitely marked as auto-scoped by the programmer. To this end, the compiler could automatically scope all variables used and report mismatches between what the programmer specifies in data-sharing attribute clauses and how the variables are actually used in the code.

2.1 Autoscoping in parallel regions

Autoscoping for `parallel` and related combined constructs (`parallel for`, `parallel sections`) was proposed by Lin et al. [2] and was implemented in the Oracle Developer Studio [5] as well as in the Polaris parallelizing compiler [11] for Fortran. When autoscoping a scalar variable that is referenced in a `parallel` construct and that does not have predetermined or implicitly determined scope, the compiler checks the use of the variable against the following rules P1–P3 in the given order:

P1 *If the use of the variable in the `parallel` construct is free of data race*

```
int x, y = 1, z, w;
#pragma omp parallel __auto(x,y,z,w)
{
  #pragma omp single nowait
  {
    z = 0;
  }

  x = y + w;

  #pragma omp single
  {
    w = y + z;
  }
}
```

Figure 1: Example of autoscoping

conditions for the threads in the team executing the construct, then the variable is scoped as shared*.*

P2 *If in each thread executing the parallel construct the variable is always written before being read by the same thread, then the variable is scoped as* private*. The variable is scoped as* lastprivate *if it can be scoped* private *and it is read before it is written after the `parallel` construct, and the construct is either a `parallel for/do` or a `parallel sections`.*

P3 *If the variable is used in a reduction operation that can be recognized by the compiler, then the variable is scoped as* reduction *with that particular operation type.*

If none of the above rules apply, the scope of the variable cannot be safely determined. In order to guarantee correctness, Oracle Developer Studio then serializes the `parallel` region.

A data race exists when two or more threads can access (i.e. read or write) the same shared variable at the same time and at least one of them writes the variable. Data race conditions can be eliminated by protecting the accesses to the variable through synchronization constructs such as `atomic`, `critical` or runtime library lock routines (`omp_set_lock()` and `omp_unset_lock()`), which force an ordering on the accesses.

2.2 Example

An example of how the above rules work is given in Figure 1, where 4 variables are auto-scoped as follows:

- Because the write of variable `x` in line 9 causes a data race, rule P1 fails to apply. All threads write `x` before reading it, so according to rule P2, `x` is scoped as *private*.
- Variable `y` is categorized as *shared* according to rule P1; it cannot be involved in a data race since in lines 9 and 13 it is read but it is never written.
- Regarding variable `w` rule P1 does not apply since the read in line 9 causes a data race with the write in line 13. Rule P2 does not apply either since all threads read `w` before one of them writes it. The usage of `w` does not follow any reduction pattern and thus rule P3 is also not applicable. Consequently `w` cannot be scoped.
- Variable `z` cannot be scoped either, since none of the rules applies; one thread will write it in line 6 but, because of the `nowait` clause in line 4, another one may read it at line 13 asynchronously, causing a data race.

3 Proposed addition of a fourth rule

We propose the following addition to the above rules:

P4 *If in each thread participating in the parallel construct the variable is first read before being written by the same thread, the variable is scoped as* firstprivate.

We note that a rule of similar spirit (albeit in the context of a `task` construct) was given in the autoscoping algorithm of [6].

The logic behind the proposed rule is the following: since rules are applied in strict order, reaching rule P4 means that rule P1 cannot be applied; thus there is a data race around the variable in question and the variable cannot be scoped as *shared*. Furthermore, in view of the fact that all threads initially read its value, the variable should not be scoped as *private* (since *private* variables are uninitialized). Rule P3 also failed to apply, hence the scoping of the variable as *firstprivate*.

This new rule can scope successfully cases where the original three rules fail. Let us revisit the example in Fig. 1, where variable `w` could not be scoped. Rules P1–P3 do not apply; all threads read its value (line 9) before one of them writes on it at line 13. Consequently, rule P4 applies and variable `w` is now scoped as *firstprivate*.

4 Nested parallelism

In this section we show that autoscoping in a `parallel` region as implemented in Oracle Developer Studio [2, 5] works as indented only if the region is not part of a lexical nesting of multiple `parallel` regions. If this is not the case, special

```
int x;
#pragma omp parallel __auto(x)      // R1
{
  #pragma omp parallel __auto(x)    // R2
  {
    x = 1;
  }
}
```

Figure 2: First nested `parallel` region code sample

```
int y;
#pragma omp parallel __auto(y)      // R3
{
  #pragma omp parallel __auto(y)    // R4
  {
    #pragma omp single
    {
      y = 2;
    }
  }
}
```

Figure 3: Second nested `parallel` region code sample

consideration is required so that the rules given in Section 2.1 scope variables correctly.

Consider the code example in Figure 2, which includes a `parallel` region ($R2$) nested within another `parallel` region ($R1$). The autoscoping results of Oracle Developer Studio are as follows:

```
Variables autoscoped as PRIVATE in R1: x
...
Variables autoscoped as PRIVATE in R2: x
...
```

While this is not an erroneous scoping per se, it does not actually follow from the given autoscoping rules. When autoscoping variable `x` in the inner region, rule P1 does not apply because `x` is written by all threads in the (inner) team, consequently rule P2 decides the variable should be scoped as *private*. Moving to the outer region ($R1$), variable `x` is never written (since in $R2$ the threads have privatized it) and rule P1 applies, so `x` should actually be autoscoped as *shared*.

In some cases though, the scoping results may be inappropriate. Such an example in shown in Fig. 3, where Oracle Developer Studio reports:

```
Variables autoscoped as SHARED in R3: y
```

```
...
Variables autoscoped as SHARED in R4: y
...
```

If we apply the autoscoping rules in the inner region ($R4$), variable `y` is only written by one thread in a `single` region with an implied barrier at the end; consequently there is no data race and rule P1 correctly scopes `y` as *shared*. Scoping `y` in region $R3$ is more complicated. In line 4, every thread of the `parallel` region $R3$ creates its own team. One thread from each team writes a new value to `y` (line 8) without the teams being synchronized among them. As a consequence, there is a data race around `y` among the different teams. Rule P1 does not apply and thus `y` *should not* be scoped as *shared*; it should be scoped as *private* according to the next rule (P2).

From the above examples it should be obvious that scoping inner regions should yield results that direct scoping decisions for outer regions. The autoscoping rules of Lin et al. [2] continue to be correct, but they should be applied in a hierarchical/recursive way and utilize nested autoscoping outcomes.

4.1 Augmenting the rules in the presence of nesting

Let us symbolize by L the lexical nesting level of a `parallel` construct, that is, the number of `parallel` constructs that surround it. For example, $R3$, the first (outer) `parallel` construct in Fig. 3, has a nesting level of 0, while the `parallel` construct enclosed within it ($R4$) has a nesting level of 1.

Definition. A read or write operation on a variable `x` is considered *protected*, if it lies within a critical section of the program; otherwise it is considered *plain* or *unprotected*.

Practically, a variable access is protected if it is within a `single` construct without a `nowait` clause, within an unnamed `critical` construct or in a code block delimited by explicit runtime locking/unlocking calls (`omp_set_lock()` and `omp_unset_lock()`). In the first case, the `single` construct guarantees that only one thread of the team will be operating on `x`; in the second case, there is a global lock protecting all unnamed `critical` regions, so that mutual exclusion is always guaranteed. The same holds for the third case assuming that all threads use the same lock.

Autoscoping should be performed recursively, scoping the most internal region first and then moving to the outer regions. We derive necessary rules for autoscoping to work correctly, augmenting the original rules of Section 2.1. Consider a variable `x` in a `parallel` construct at nesting level $L + 1$, where $L \geq 0$. We propose the following rules:

NP1 *If `x` is scoped as* private*, scoping at level L may ignore this `parallel` construct.*

NP2 *If `x` is scoped as* firstprivate*, the `parallel` construct is considered equivalent to a plain read operation on `x` in nesting level L.*

NP3 *If `x` is scoped as* lastprivate*, the `parallel` construct is considered equivalent to a plain write operation on `x` in nesting level L.*

NP4 *If `x` is scoped as* reduction*, the `parallel` construct is considered equivalent to a plain read followed by a plain write operation on `x` in nesting level L.*

NP5 *If `x` is scoped as* shared*, then a read (write) operation in level $L + 1$, protected or not, is considered to be a plain read (write) operation in nesting level L.*

4.2 Explanation of the new rules

Rule NP1 should be clear; if `x` is scoped as *private* in nesting level $L + 1$, uninitialized private copies of `x` are created for all threads at this level and these copies get destroyed at the end of the `parallel` region, irrespectively of the data-sharing attributes of variable `x` at level L.

For the rest of the rules, the idea is to summarize the whole nested region with equivalent read/write operations on variable `x`, which can then be used to scope `x` in level L. Rule NP2 works similarly to NP1; in this case however, the private copies of the threads at level $L + 1$ need to be initialized, hence a read operation on variable `x` at level L is required.

If `x` is scoped as *lastprivate* at nesting level $L + 1$ (it must actually be a `parallel for` or `parallel sections` construct), then upon the end of the region, the last thread will use its privatized copy to define the value of `x` at nesting level L. Consequently, the whole region in level $L+1$ can be considered as a plain write operation in level L.

Finally, if `x` is a *reduction* variable, then private copies are made for each thread at level $L + 1$ but at the end they are combined to *update* `x` at level L; hence a read and a write operation in rule NP4.

Rule NP5 is based on the following observation: even if a read or write operation on a shared variable is protected between the threads of nesting level $L + 1$, it remains unprotected for nesting level L. For example, consider Fig. 3 again. If the `parallel` region in nesting level 0 ($R3$) produces n threads, there will be n different teams executing at nesting level 1 (region $R4$). Variable `y` is scoped as shared in nesting level 1. The write operation on `y` in line 8 is enclosed in a `single` construct without a `nowait` clause and is thus protected within a team. However because of the n concurrently executing level-1 teams, those n writes on variable `y` are actually asynchronous accesses.

5 Implementation in the OMPi compiler

The OMPi compiler [4] is a lightweight OpenMP C infrastructure, consisting of a source-to-source translator and a flexible, modular runtime system. It takes an OpenMP program written in C and outputs an equivalent multithreaded C program to be compiled using the system's sequential compiler. OMPi is an open source project and targets general-purpose SMPs and multicore platforms.

It provides a large portion of the OpenMP V4.5 functionalities, including full support for device constructs.

To be able to compare with Oracle Studio, we have implemented both the `default(__auto)` and the `__auto()` clauses as proposed in [2]. In addition, autoscoping in OMPi can also be activated by the `--scopecheck` compilation flag, even for programs that do not employ the above clauses. When the developer specifies this flag, the compiler auto-scopes all variables that do not have predetermined data-sharing attributes, for every `parallel` construct met (including combined `parallel`-worksharing ones). Before such a construct is transformed, the following actions are performed:

- All data-sharing attribute and reduction clauses are temporarily ignored; in essence all used variables become implicitly determined.
- All variables that do not have predetermined data-sharing attributes are treated as if there is a `default(__auto)` clause present.
- Finally, the autoscoping results are compared against the given data-sharing attribute clauses.

If a variable is scoped differently than what the developer explicitly specified, a warning is issued along with a suggestion of what the correct clause would be for the variable in question.

5.1 Data race detection

Based on the control flow graph (CFG) of the relevant code region, OMPi performs typical static analysis in order to decide whether the auto-scoped variables are subjected to data races or not, and categorize them according to the rules presented in the previous sections.

Assuming the program is a correct OpenMP program, the data race detection algorithm operates in CFG portions delimited by barriers (since, according to the OpenMP specifications a barrier will be seen by all threads of the team). After an implicit or explicit barrier is met, a new analysis phase begins up to the next barrier, and so on until the end of the construct is reached.

For a given variable x, the race detection logic is summarized as follows:

- If x is subjected to an unprotected write operation, then it is subjected to data race as well.
- If x is subjected to a read or a protected write operation, then it's considered as *suspicious* for data race.
- All suspicious operations are checked in pairs, in case their asynchronous combination causes a data race. If one of them is a write and they are not both enclosed in `atomic` constructs, `master` constructs or identically-named `critical` regions, then x is subjected to data race.

Based on the above, OMPi uses the rules of Sections 2.1, 3 and 4 to categorize the variables. We note that in the current prototype the static analysis applied has some limitations:

1. Pointers to variables are not tracked; any variable for which its memory location is referenced, is left uncategorized.

2. Mutual exclusion through runtime locking functions is not yet recognized as such; variable accesses between an `omp_set_lock()` and an `omp_unset_lock()` calls are considered unprotected.

3. Array handling is restricted; an unprotected concurrent write operation is always considered to cause a data race, unless it occurs within a `for` worksharing loop and the index of the accessed element is the loop index.

While this functionality is not critical, and is used only for assisting the developer, we are currently improving the power and the accuracy of the implementation. For the shake of completeness, we report that we have evaluated our implementation extensively with programs that do not employ nested parallelism. Even with the above limitations, we report a successful scoping of 95% of the variables appearing in all `parallel` regions of the NAS Parallel Benchmarks 3.0; a C version of the benchmarks has been employed [12]. Out of 420 variables scoped in total, 9 of them were variables passed by reference and another 11 cases were accesses to array elements that resulted in false positive data races.

6 A concrete example: Mandelbrot set calculation

In addition to various synthetic codes like the ones in Figs. 1–3, we demonstrate the applicability of the proposed rules using a concrete application. The code in Fig. 4 is a portion of a parallelized application that creates a Mandelbrot fractal image using the so-called optimized escape time algorithm. The image is represented by a two-dimensional array of integer-valued pixels with the provided dimensions (`width`$\times$`height`) and it is passed as a pointer to a vector of consecutive elements in function `mandel`. The algorithm iterates over each row of pixels (`i`-loop) and for each row, it iterates over its columns (`j`-loop). The iterations are independent of each other as the calculation of the value of each pixel requires only the position (x, y) of the pixel. While parallelizing the outer `i`-loop with an appropriate schedule yields satisfactory speedups, parallelizing the inner `j`-loop may improve performance due to the high imbalance of the work of the outer loop iterations.

We used both OMPi and Oracle Studio to scope the involved variables automatically. For regions R1 and R2, OMPi produces the following autoscoping output:

```
[OMPi INFO] parallel region @line 11:
```

```
void mandel(int *array, int width, int height) {
  int n, iter, maxiters=400;
  double x, y, u, v, u2, v2;
  double scale_real = 3.5/width, scale_imag = 3.5/height;

  ...
  y = -1.75 - scale_imag;

  // R1
  #pragma omp parallel for auto(x,y,u,v,u2,v2,iter,array,\
      height,scale_imag,width,scale_real,maxiters)
  for (int i = 0; i < height; i++) {
    y += scale_imag;
    x = -2.25 - scale_real;

    // R2
    #pragma omp parallel for auto(x,y,u,v,u2,v2,iter,array,\
        height,scale_imag,width,scale_real,maxiters)
    for (int j = 0; j < width; j++) {
      x += scale_real;
      u=v=u2=v2=0.0;
      for (iter=0; (u2+v2<4.0 && iter<maxiters); iter++) {
        v = 2 * v * u + y;
        u = u2 - v2 + x;
        u2 = u*u;
        v2 = v*v;
      }
      if (iter == maxiters)
        iter = 0;
      array[i*width + j] = iter;
    }
  }
  ...
}
```

Figure 4: Portion of a parallelized Mandelbrot set application.

```
    scoped as shared: height, array, scale_imag, width, scale_real,
        maxiters, x
    scoped as private: iter, v2, u2, v, u
    scoped as firstprivate: y
    (13 autoscoped)

[OMPi INFO] parallel region @line 18:
    scoped as shared: y, array, height, scale_imag, width, scale_real,
        maxiters
    scoped as private: iter, v2, u2, v, u
    scoped as firstprivate: x
    (13 autoscoped)
```

On the other hand, Oracle Studio outputs the following:

```
Variables autoscoped as SHARED in R1: width, scale_real, maxiters, array,
    height, scale_imag
Variables autoscoped as PRIVATE in R1: v2, iter, x, y, u, v, u2
...
Variables autoscoped as SHARED in R2: width, scale_real, maxiters, y,
    array, height, scale_imag
Variables autoscoped as PRIVATE in R2: v2, iter, x, u, v, u2
```

We observe the following:

1. OMPi scopes variable `x` as `firstprivate` in R2, by applying the newly introduced rule P4. Oracle Studio scopes wrongly `x` as `private`, which will lead to an uninitialized read in line 24.

2. OMPi scopes `x` as `shared` in R1, by using nested rule NP2, since the privatization of `x` in R2 is treated as a plain read operation; therefore `x` is not a subjected to a data race in R1. Oracle Studio on the other hand, privatizes `x` in R1, effectively contradicting rule P1.

3. Both compilers scope `y` as `shared` in R2, since it is only read and not subjected to data races. However, OMPi scopes `y` as `firstprivate` in R1, using P4, while Oracle Studio scopes it as `private`. This will cause an uninitialized read at line 13, resulting in an undefined value for `y`.

7 Conclusions

Auto-scoping is a powerful facility to help with the parallelization of sequential code using OpenMP. Whether it is used as a compromise between manual and automatic parallelization or as a safeguard during development, autoscoping can ease the burden of the OpenMP programmer. Autoscoping was originally suggested and implemented for `parallel` and combined `parallel`-worksharing regions, and later for `task` regions as well as other models with similar constructs. Focusing on the former, in this work we extend the original rules that guide the autoscoping procedure. Our extensions generalize its application and guarantee its correctness in cases where there exist lexical nestings involving the above regions. Finally, we discuss their implementation in the context of the OMPi source-to-source OpenMP compiler. As part of our current and future work, we focus on autoscoping for `task` and `target` OpenMP constructs.

References

[1] OpenMP ARB, OpenMP Application Program Interface V5.2 (Nov. 2021).

[2] Y. Lin, C. Terboven, D. a. Mey and N. Copty, Automatic scoping of variables in parallel regions of an OpenMP program, in *WOMPAT '04, 5th Workshop on OpenMP Applications and Tools* ed. B. M. Chapman (Berlin, Heidelberg, 2004) 83–97.

[3] M. Süß and C. Leopold, Common mistakes in OpenMP and how to avoid them, in *Proc. IWOMP 2006, 2nd International Workshop on OpenMP* (Reims, France, 2006) 312–323.

[4] V. V. Dimakopoulos, E. Leontiadis and G. Tzoumas, A portable C compiler for OpenMP V.2.0, in *Proc. of EWOMP 2003, the 5th European Workshop on OpenMP* (Aachen, Germany, 2003) 5–11.

[5] Oracle, Oracle Developer Studio 12.6: OpenMP API User's Guide (June 2017).

[6] S. Royuela, A. Duran, C. Liao and D. J. Quinlan, Auto-scoping for OpenMP tasks *Proc. IWOMP'12, 8th International Workshop on OpenMP*, (Berlin, Heidelberg, 2012) p. 29–43.

[7] C.-K. Wang and P.-S. Chen, Automatic scoping of task clauses for the OpenMP tasking model, *The Journal of Supercomputing* **71**(3) (2015) 808–823.

[8] A. Munera, S. Royuela, R. Ferrer, R. Peñacoba and E. Quiñones, Static analysis to enhance programmability and performance in OmpSs-2, in *Proc. ISC High Performance 2020 International Workshops* (Cham, 2020) 19–33.

[9] C. Liao, D. J. Quinlan, J. J. Willcock and T. Panas, Semantic-aware automatic parallelization of modern applications using high-level abstractions, *International Journal of Parallel Programming* **38**(5–6) (2010) 361–378.

[10] T. G. Mattson, Y. H. He and A. E. Koniges, *The OpenMP Common Core: Making OpenMP Simple Again* (MIT Press, Cambridge, Massachusetts, 2019).

[11] M. Voss, E. Chiu, P. M. Y. Chow, C. Wong and K. Yuen, An evaluation of auto-scoping in OpenMP, in *Proc. WOMPAT '04, 5th Workshop on OpenMP Applications and Tools* (Berlin, Heidelberg, 2004) 98–109.

[12] M. Popov, C. Akel, F. Conti, W. Jalby and P. De Oliveira Castro, Pcere: Fine-grained parallel benchmark decomposition for scalability prediction, in *Proc. 2015 IEEE International Parallel and Distributed Processing Symposium* (Hyderabad, India, 2015) 1151–1160. Available at: `https://github.com/benchmark-subsetting/NPB3.0-omp-C` (accessed May 2022).