International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:5, 2010

Programming Aid Tool for Detecting Common Mistakes of Novice Programmers in OpenMP Code

Jae Young Park, Seung Wook Lee, and Jong Tae Kim

***Abstract*—**OpenMP is an API for parallel programming model of shared memory multiprocessors. Novice OpenMP programmers often produce the code that compiler cannot find human errors. It was investigated how compiler coped with the common mistakes that can occur in OpenMP code. The latest version(4.4.3) of GCC is used for this research. It was found that GCC compiled the codes without any errors or warnings. In this paper the programming aid tool is presented for OpenMP programs. It can check 12 common mistakes that novice programmer can commit during the programming of OpenMP. It was demonstrated that the programming aid tool can detect the various common mistakes that GCC failed to detect.

***Keywords*—** parallel programming, OpenMP, programming aid

I. Introduction

SINCE increasing the frequency of clock rate for the performance improvement may reach the limit multi-core architecture introduced. Multi-core processor takes advantage of parallel processing, but it faces software challenges. Especially programmer productivity may get worse with parallel programming and it is very difficult to directly compose the parallel program. OpenMP is an API for parallel programming model of shared memory multiprocessors. OpenMP enables the creation of shared-memory parallel programs. It is comprised of a set of compiler directives that describe the parallelism in the source code, along with a supporting library of subroutines available to applications[1]. With the appearance of OpenMP directive programming with directive is relatively easy when comparing to writing message passing code[2]. But it is still not easy for the programmer who only has experiences with sequential program using OpenMP directives for enhancing performance. To deal with these difficulties, the automation tools like CAPO[2] were developed. They have some limitations such that they cannot perform semantic analysis. Assume that the automation tool encounters with 'for loop' like in Fig. 1. The func_A does not have any dependency. If there exists some information exclusively known to programmer such that when func_A is used 5 times or less than that, internal fork and barrier produced to make parallel can harm the function, the automation tool cannot cope with this. To alleviate this problem, programmer should modify it like in Fig. 2. However, intervention of the programmer may introduce other problems of occurring human error. To easily correct the human programming error, it is required to get the information about where and what kind of problem happened.

Jae Young Park, Seung Wook Lee, and Jong Tae Kim are with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea(e-mail: jyp8389@gmail.com, seungwooks@gmail.com, jongtkim@gmail.com)

```
#pragma omp parallel for
for(i=0;i<n;i++) {
func_A();
}
```

Fig. 1 Limitation of automation tool

```
#pragma omp parallel for if(n>5)
for(i=0;i<n;i++) {
func_A();
}
```

Fig. 2 Modified code by programmer

Generally, when a certain tool or language is used, possible mistakes that programmer can commit are informed through error or warning statements. For OpenMP, the compiler supporting OpenMP takes this role, and programmer conducts debugging to correct them. If the compiler supporting OpenMP fails to inform programmers of possible errors and warning it waste lots of time to find and amend them even for simple mistakes. This kind of problem is gradually being solved with development of the compiler supporting OpenMP, but there still exists many shortcomings. Therefore the new programming aid tool is developed to help detecting mistakes which the compiler such as GCC fails to find. The tool can produce the solution for common mistakes which OpenMP programmers are easily committed. It is named as OPAT (OpenMP Programming Aid Tool).

II. Problem Definition

S¨uß and Leopold studies the programming errors of students who took the parallel programming course for 2 years, along with the numbers frequency occurred like in Table 1[3]. There were total 84 students in 43 groups. Correctness Mistake in Table 1 means the types of mistakes which can function differently from the intention of the users and Num is number of groups who committed the certain mistake. Investigated

mistakes are the mistakes which students failed to find and correct within the given hour. New mistakes such as (11) and (12) that are not found in [3] are added. They are closely connected with (7) and (9) respectively.

Handling error or warning is an important indicator for the performance of compiler. To examine the performance of compiler supporting OpenMP, latest version of GCC(4.3.3)[4] which is one of the most widely used compiler was used. Using it, it was investigated how to deal with the problems in 12 situations in Table 1 and the result showed that there were no errors or any warning messages when compiling had been done. Table 1 indicates the types of mistakes and shows how to deal with the mistakes in GCC. None mean that compiler cannot detect the error during the compiling process. X indicates the case that mistake is automatically corrected and nothing happened during actual operation. The case of read of shared variable without flush is excluded from our tool because it would not be a problem for structurally potential flush. As shown in Table 1, GCC does not generate warning and error messages for programmers' mistakes. Therefore it is very difficult to debug the code because it is hard to find out where and what the problem happens with GCC. In this work the programming aid tool for OpenMP programs is developed. It detects 12 common mistakes that novice programmer commits during the programming of OpenMP.

Fig. 3 shows an example of C code that calculates mathematical constant π. First one is the original code which is the sequential code to be handled with single thread. Second code is the OpenMP code with a shared variable protection mistake. GCC produces the compiled code without any error or warning messages, but executed code generates wrong value. OPAT checks the shared variable protection mistake and generates a warning report. OPAT found that the shared variable sum is being used without protection at line 13. Programmer pays attention the warning error from OPAT and corrects the mistake. The corrected code is the third code in Fig. 3 and finds the correct value of mathematical constant π. (It is desirable to use reduction rather than critical in this specific example.)

TABLE I
LIST OF COMMON MISTAKES

Type	Problem Correctness Mistake	Num	GCC
(1)	Access to shared variables not protected	18	None
(2)	Use of locks without flush	18	X
(3)	Read of shared variable without ordered construct	15	X
(4)	Forget to mark private variable without ordered construct	11	None
(5)	Use of ordered clause without ordered construct	4	None
(6)	Declare loop variable in #pragma omp parallel for as shared	3	X
(7)	Forget to put down for in #pragma omp parallel for	2	None
(8)	Try to change number of thread in parallel region after start region	2	X
(9)	omp_unset_lock() called from non-owner thread	2	None
(10)	Attempt to change loop variable while in #pragma omp for	2	None
(11)	Use of for directive without parallel construct	New	None
(12)	Using lock as a barrier	New	None

Original code

```
   for (i=0; i<num;i++) {
     x= (i+0.5) * step;
     sum + = 4.0/(1.0 + x*x) ;}
    
  printf("PI = %.8f(sum = %.8f)\n, step*sum, sum);
```

Original result

PI = **3.14159265** (sum = 3.141592653.59213972)

Mistakes code checked with OPAT

```
#pragma omp parallel for private(x)
    for (i=0; i<num;i++) {
      x= (i+0.5) * step;
      sum + = 4.0/(1.0 + x*x); }
     
  printf("PI = %.8f(sum = %.8f)\n, step*sum, sum);
   
/*    Warning Report
LINE_13:Access to shared variables not protected
*/
```

Compiled result in GCC with mistakes

PI = **1.18051801** (sum = 1180518011.76885509)

Corrected code

```
#pragma omp parallel for private(x)
    for (i=0; i<num;i++) {
      x= (i+0.5) * step;
#pragma omp critical {
      sum + = 4.0/(1.0 + x*x); }
      }
     
  printf("PI = %.8f(sum = %.8f)\n, step*sum, sum);
   
/*    Warning Report
*/
```

Corrected result

PI = **3.14159265** (sum = 3.141592653.59027195)

Fig. 3 Comparing example GCC with OPAT

III. FUNCTIONALITY OF OPAT

In this section the flow of OPAT and the functionality of OPAT with examples are presented.

A. Flow of OPAT

The way to inspect file with OPAT is pretty simple as shown in Fig. 4. It receives input C code file with the command word of OPAT. It performs the phrasing and starts to analyze the code. If the code section is not the area applying OpenMP it send the corresponding section to new file, and if it is the area applying OpenMP, it inspects the mistakes by applying the checking rule and saves the problem with the reason. After file inspection is over it prints out warning report message and line number

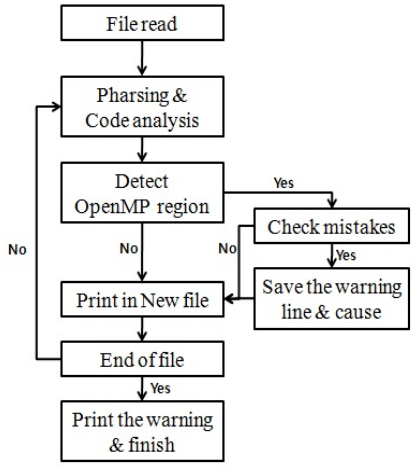

Fig. 4 Flow chart of OPAT

B.Access to Shared Variable not Protected

Because specified data are shared by many threads, when 'shared' is used without protection, it influences the data value which is supposed to be used for other threads, so it cannot secure the variable value. It arises when task code accesses shared data non-atomically. It is necessary to generate the warning message. But OPAT does not generate the warning message if variable is declared as private and is accessed only one thread at a time.

Fig. 5 is a full code of the example used in Fig. 3. As a result of using OPAT, it indicates the problem of approaching at line number 13 without protecting the shared variable.

shared_ex.ocat.c

```
#include <stdio.h>
#include <stdlib.h>

int num_steps=1000000000;
int main()
{
      int i;
      double x, step, sum = 0.0;
      step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,i) shared(sum)
      for (i=0; i<num_steps; i++) {
      x = (i+0.5) * step;
      sum += 4.0/(1.0 + x*x);
      }

      printf("PI = %.8f (sum = %.8f)\n", step*sum, sum);
    return EXIT_SUCCESS;
}
/*    Warning Report
LINE_13:Access to shared variables not protected
*/
```

Fig. 5 Access to shared variables not protected

C.Forget to Mark Private Variables as Such

Private variable is used when many threads take one variable independently. When the value of variable is changed in parallel region, special protection method should be taken or the variable should be designated as private, last private or first private. If variable is not apparently designated as private in sharing construct, compiler considers it as private and there is no warning.

The code in Fig. 6 is an example of OpenMP example distribution fig. 4.36[5] which can be obtained from OpenMP web page. Variables *i* and *a* are designated as private. It is compiled normally without any problem in GCC, but the value of a_shared is equal to 4 that is the wrong value. OPAT generates the warning messages of not designating variable as private. By correcting the mistake, the right value for a_shared was achieved.

```
//private(i)  private(a)
#pragma omp parallel for shared(a_shared)
    for (i=0; i<n;i++) {
      a = i + 1;
      printf("Thread  %d has a value of a = %d for I =
%d\n",omp_get_thread_num(),a,i);
      if (I == n-1) a_shared = a;
      }

/*    Warning Report
LINE_66:Forget to mark private variables as such
LINE_68:Forget to mark private variables as such
LINE_71:Access to shared variables not protected
*/
```

Fig. 6 Forget to mark private variables as such

D. Use of Ordered Clause without Ordered Construct

The code in Fig. 7 is from the OpenMP example distribution fig. 4.74 [5]. It indicates the case which declares the directive #pragma omp parallel for ordered, but ordered is not used in the for loop. It means that there are some parts of code which should be processed in order. Therefore it can be considered as a mistake not putting the part of code that should be ordered. A warning is generated.

```
#pragma omp parallel for ordered schedule(runtime)\
      Private(I,TID) shared(n,a)
for (i=0; i<n; i++) {

//#pragam omp ordered
  {
  printf("Thread %d prints value of a[%d] = %d\n",TID,I,a[i]);
  }

/*    Warning Report
LINE_70:Use of ordered clause without ordered construct
 */
```

Fig. 7 Use of ordered clause without ordered construct

E. Forget to Put Down for in #pragma omp parallel for

This is the problem occurring when the parallel area is designated but for directive is not used as shown in Fig. 8. When it encounters a for loop, it does not divide the loop and process them in parallel. Several threads execute the same loop separately. Compiler does not find it as an error or a warning, but it should be considered as a mistake. OPAT generates a warning.

```
#pragma omp parallel shared(n) private(i)
//#pragma omp for
for (i=0; i<n; i++){
  printf("Thread %d executes loop iteration %d\n",
omp_get_thread_num(),i);
```

```
  }
/*    Warning Report
LINE_66:Forget to put down for in #pragma omp parallel for
 */
```

Fig. 8 Forget to put down for in #pragma omp parallel for

F. Use of for Directive Without Parallel Construct

Fig. 9 show the case that does not designate parallel area but use for directive. Compiler ignores pragma omp directive and treat it as single thread. The code is not executed in parallel and there is no speed up. It is hard to aware because it does not influences the result value.

```
//#pragma omp parallel …
    
#pragma omp for
  for (i=0; i<n; i++) {
    
  }
/*   Warning Report
LINE_66:Use of for directive without parallel construct
*/
```

Fig. 9 Use of for directive without parallel construct

G. omp_unset_lock() Called from Non-owner Thread & Using Lock as Barrier

Lock function should be unset at the thread where lock function is set [6]. In second section of Fig. 11, it generates warning since the thread which calls omp_set_lock and the thread which calls omp_unset_lock are different. Because omp_set_lock function blocks the thread until lock variable is available, so it should call the omp_set_lock function and then calls the omp_unset_lock function before currently active thread finishes. As shown in Fig. 10, if lock function is not unset until the thread finishes, the warning message is generated.

```
#pragma omp parallel sections {
   #pragma omp section {
       
      omp_set_lock(&mylock);
          //omp_unset_lock(&mylock);
   }
   #pragma omp section {
       
      //omp_unset_lock(&mylock);
          omp_set_lock(&mylock);
   }
}
/*   Warning Report
LINE_36:Using lock as a barrier
LINE_41:omp_unset_lock() called from non-owner thread
*/
```

Fig. 10 Using lock as a barrier & omp_unset_lock called from non-owner thread

H. Attempt to Change Loop Variable While in #pragma omp for

Changing loop variable in the loop is prohibited in the parallel handling structure. GCC cannot find changing loop variable error. Using OPAT a warning is generated when the loop variable is changed in for loops as shown in Fig. 11.

```
#pragma omp parallel for shared(n) private(i)
for (i=0; i<n; i++){
  printf("Thread %d executes loop iteration %d\n",
omp_get_thread_num(),i);
  i=5;
 }
/*   Warning Report
LINE_69:Attempt to change loop variable while in #pragma omp
for
 */
```

Fig. 11 Attempt to change loop variable while in #pragma omp for

IV. Conclusion

It is not easy for the programmer who only has experiences with sequential program using OpenMP directives for enhancing performance. Programmers are prone to commit mistakes. Handling error or warning is an important indicator for the performance of compiler. To examine the performance of compiler supporting OpenMP, latest version of GCC which is one of the most widely used compilers was used. It was found that GCC does not generate warning and error messages for the specific programmers' mistakes in OpenMP code. In this paper It is presented the programming aid tool for OpenMP programs. It detects 12 common mistakes that novice programmer commits during the programming of OpenMP. OPAT is very light and easy to install. It is also available for any platform. It is demonstrated and verified that the programming aid tool can detect the various common mistakes that GCC failed to detect.

Acknowledgment

This research was supported by the Converging Research Center Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2009-0081958).

References

[1] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon "*Parallel Programming in OpenMP"*, Morgan Kaufmann Publishers, 2001.
[2] Haoqiang Jin, Michael Frumkin and Jerry Yan, "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamic Codes", in *NASA Ames Research Center. ISHPC 2000, LNCS*, 2000.
[3] Michael S¨uß and Claudia Leopold, "Common Mistakes in OpenMP and How to Avoid Them," IWOMP 2005/2006, LNCS 4315, pp. 312-323,2008.
[4] http://gcc.gnu.org/gcc-4.4/.
[5] http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip.
[6] Babara Chapman, Gabriele Jost and Ruud van der Pas "Using OpenMP", 2007,pp.268.