Verifying an OpenMP Parallelization with the Intel® Thread Checker

Rolf Rabenseifner, Bettina Krammer
rabenseifner@hlrs.de, krammer@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Dec. 20, 2007

Used version of Intel® Thread Checker under Linux: 3.1 (and 3.0)
Goal

To detect race conditions
• and other parallelization errors (e.g., missing firstprivate)
• in OpenMP parallel application programs

OpenMP parallelizations should never be used in production without verification with race-condition checking tools (like Assure or Intel® Thread Checker)

Content

- Overview slides 2 – 6
- Seven Examples slides 9 – 27
- Restrictions in TCI mode slides 28 – 29
- End of checking slide 30
- Other Products slide 31
- Wrap-Up and Summary slide 32 – 33
- Practical slide 34
- Appendix slides 35 – 53
Data-Race

Def.: Two threads access the same shared variable and at least one thread modifies the variable and the accesses are concurrent, i.e. unsynchronized.

This type of race condition is detected by the Intel® Thread Checker.
Data-Race: Def.: Two threads access the same shared variable and
at least one thread modifies the variable and
the accesses are concurrent, i.e. unsynchronized

There are a lot of programs, that are wrong,
but without data-races according to that definition, e.g.

```c
#include <stdio.h>
int main(){
  int i,tmp,x = 0;
  #pragma omp parallel for private(tmp)
  for (i=0; i<100; i++)
  {
    #pragma omp critical
    tmp = x;
    tmp = tmp + 1;
    #pragma omp critical
    x   = tmp;
  }
  printf("x = %d \n",x);
}

> ecc data-race.c
> ./a.out
x = 100
> ecc -openmp data-race.c
> ./a.out
x = 60
> ./a.out
x = 36
> 

This type of race condition is not detected by the Intel® Thread Checker
Intel® Thread Checker has **two modes**

- **Thread count independent analysis mode**
  - Projection technology
    executes the sequential version of the application and treats it as the specification for the OpenMP application
  - Source code instrumentation
  - Compares sequential code with maximal parallel execution
  - Can detect many different kinds of errors
  - Can detect far more errors than possible with simulation approach
  - Not applicable if parallel code can not be projected to sequential code
  \(\Rightarrow\) **Used on most slides**

- **Simulation approach**
  - Using binary instrumentation
  - Supported for x86 code on x86 and EM64T platforms
  - Thread-parallel execution

\(\Rightarrow\) Not supported on Itanium!
Method

- Compile your OpenMP application with Thread Checker
- Start and execute with Thread Checker
  - executed on 1 thread
  - verifying all memory accesses
  - ~300 times slower than normal execution!!!
- Invoke analysis tool
  - Error report
  - with references to your source file
- Try to find the parallelization bugs in your application
- Try to correct these parallelization bugs
  - without modifying the serial semantics of the program
- Compile and execute again
  - until all errors are resolved
Processing

Source program
my_prog.c or my_prog.f

compile with options
-tcheck -openmp -g

Executable my_prog

execute with command
tcheck_cl ./my_prog

threadchecker.thr
and ascii myprog.txt

Further analysis with
tcheck_cl -f ...
tcheck_cl -s ...

Example:
module load itt

icc -tcheck -openmp -g -o my_prog my_prog.c
ifort -tcheck -openmp -g -o my_prog my_prog.f

tcheck_cl -w 90 -o my_prog.txt ./my_prog

Text output (130 chars line width):
tcheck_cl -f txt -w 130 threadchecker.thr

Comma separated output (e.g., as input for Excel):
tcheck_cl -f csv threadchecker.thr

Call stack of all events (e.g. depth = 4):
tcheck_cl -s 4 threadchecker.thr

In ITC version 3.0.22356 and older, only one event is printed with -s <event_id>
### Example with demo_with_bugs.c

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Con-text [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exa.1</td>
<td>Write -&gt; Read</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory read of a[] at &quot;demo.c&quot;:30 conflicts with a prior memory write of a[] at &quot;demo.c&quot;:29 (flow dependence)</td>
<td>&quot;demo.c&quot; :29</td>
<td>&quot;demo.c&quot; :30</td>
</tr>
<tr>
<td>Exa.2</td>
<td>Write -&gt; Read</td>
<td>Error</td>
<td>197</td>
<td>omp parallel reg.</td>
<td>Memory read of a[] at &quot;demo.c&quot;:54 conflicts with a prior memory write of a[] at &quot;demo.c&quot;:49 (flow dependence)</td>
<td>&quot;demo.c&quot; :49</td>
<td>&quot;demo.c&quot; :54</td>
</tr>
<tr>
<td>Exa.3</td>
<td>Read -&gt; Write</td>
<td>Error</td>
<td>197</td>
<td>omp parallel reg.</td>
<td>Memory write of a[] at &quot;demo.c&quot;:49 conflicts with a prior memory read of a[] at &quot;demo.c&quot;:54 (antidependence)</td>
<td>&quot;demo.c&quot; :54</td>
<td>&quot;demo.c&quot; :49</td>
</tr>
<tr>
<td>Exa.4</td>
<td>Write -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory write of x at &quot;demo.c&quot;:71 conflicts with a prior memory read of x at &quot;demo.c&quot;:54 (output dependence)</td>
<td>&quot;demo.c&quot; :71</td>
<td>&quot;demo.c&quot; :71</td>
</tr>
<tr>
<td>Exa.5</td>
<td>Read -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory write of x at &quot;demo.c&quot;:71 conflicts with a prior memory read of x at &quot;demo.c&quot;:72 (anti dependence)</td>
<td>&quot;demo.c&quot; :72</td>
<td>&quot;demo.c&quot; :71</td>
</tr>
<tr>
<td>Exa.6</td>
<td>undefined in access</td>
<td>Caution</td>
<td>1</td>
<td>omp parallel reg.</td>
<td>OpenMP – the access at &quot;demo.c&quot;:88 is undefined, the expected value was defined at &quot;demo.c&quot;:84 in serial execution</td>
<td>&quot;demo.c&quot; :84</td>
<td>&quot;demo.c&quot; :88</td>
</tr>
<tr>
<td>Exa.7</td>
<td>undefined in the serial c.</td>
<td>Warning</td>
<td>1</td>
<td>&quot;demo.c&quot;:17</td>
<td>OpenMP – undefined in the serial code (original program) at &quot;demo.c&quot;:91 with &quot;demo.c&quot;:88</td>
<td>&quot;demo.c&quot; :91</td>
<td>&quot;demo.c&quot; :88</td>
</tr>
<tr>
<td>Exa.8</td>
<td>Read -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory write of sum at &quot;demo.c&quot;:105 conflicts with a prior memory read of sum at &quot;demo.c&quot;:72 (anti dependence)</td>
<td>&quot;demo.c&quot; :105</td>
<td>&quot;demo.c&quot; :105</td>
</tr>
<tr>
<td>Exa.9</td>
<td>Write -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory read of sum at &quot;demo.c&quot;:105 conflicts with a prior memory write of sum at &quot;demo.c&quot;:105 (output dependence)</td>
<td>&quot;demo.c&quot; :105</td>
<td>&quot;demo.c&quot; :105</td>
</tr>
<tr>
<td>Exa.10</td>
<td>Write -&gt; Read</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory write of sum at &quot;demo.c&quot;:105 conflicts with a prior memory write of sum at &quot;demo.c&quot;:105 (output dependence)</td>
<td>&quot;demo.c&quot; :105</td>
<td>&quot;demo.c&quot; :105</td>
</tr>
<tr>
<td>Exa.11</td>
<td>OpenMP – cannot be private</td>
<td>Caution</td>
<td>98</td>
<td>omp for</td>
<td>OpenMP – The access at &quot;demo.c&quot;:136 cannot be private because it expects the value previously defined at &quot;demo.c&quot;:136 in the serial execution</td>
<td>&quot;demo.c&quot; :136</td>
<td>&quot;demo.c&quot; :136</td>
</tr>
<tr>
<td>Exa.12</td>
<td>OpenMP – cannot be private</td>
<td>Caution</td>
<td>99</td>
<td>omp for</td>
<td>OpenMP – The access at &quot;demo.c&quot;:178 cannot be private because it expects the value previously defined at &quot;demo.c&quot;:178 in the serial execution</td>
<td>&quot;demo.c&quot; :178</td>
<td>&quot;demo.c&quot; :178</td>
</tr>
<tr>
<td>Exa.13</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program</td>
<td>Thread termination at &quot;demo_with_bugs.c&quot;:17 - includes stack allocation of 10485760 and use of 2416 bytes</td>
<td>&quot;demo.c&quot; :17</td>
<td>&quot;demo.c&quot; :17</td>
</tr>
</tbody>
</table>
Example 1 — source code

25     a[0] = 0;
26     #pragma omp parallel for
27     for (i=1; i<N; i++)
28     {
29         a[i] = 2.0*i*(i-1);
30         b[i] = a[i] - a[i-1];
31     } /* end of omp parallel for */

What's wrong?

How to solve?
## Example 1 – analysis

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write -&gt; Read data-race</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory read of a[] at &quot;demo_with_bugs.c&quot;: 30 conflicts with a prior memory write of a[] at &quot;demo_with_bugs.c&quot;: 29 (flow dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:29</td>
<td>&quot;demo_with_bugs.c&quot;:30</td>
</tr>
</tbody>
</table>

```c
25    a[0] = 0;
26    # pragma omp parallel for
27    for (i=1; i<N; i++)
28    {
29      a[i] = 2.0*i*(i-1);
30      b[i] = a[i] - a[i-1];
31    } /* end of omp parallel for */
```

### Solution

- Two separate loops  will cause bad cache reuse!
- Re-computing of a[i-1], i.e., b[i] = a[i] - 2.0*(i-1)*(i-2);

In the printed version, the solutions can be found in the appendix.
Example 2 – source code

43   a[0] = 0;
44   # pragma omp parallel
45   {
46     # pragma omp for nowait
47     for (i=1; i<N; i++)
48       {
49         a[i] = 3.0*i*(i+1);
50     } /* end of omp for nowait */
51     # pragma omp for
52     for (i=1; i<N; i++)
53       {
54         b[i] = a[i] - a[i-1];
55       } /* end of omp for */
56   } /* end of omp parallel */

What's wrong?

How to solve?
Example 2 – analysis

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Write -&gt; Read data-race</td>
<td>Error</td>
<td>197</td>
<td>omp parallel region</td>
<td>Memory read of a[] at &quot;demo_with_bugs.c&quot;: 54 conflicts with a prior memory write of a[] at &quot;demo_with_bugs.c&quot;: 49 (flow dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:49</td>
<td>&quot;demo_with_bugs.c&quot;:54</td>
</tr>
<tr>
<td>3</td>
<td>Read -&gt; Write data-race</td>
<td>Error</td>
<td>197</td>
<td>omp parallel region</td>
<td>Memory write of a[] at &quot;demo_with_bugs.c&quot;: 49 conflicts with a prior memory read of a[] at &quot;demo_with_bugs.c&quot;: 54 (anti dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:54</td>
<td>&quot;demo_with_bugs.c&quot;:49</td>
</tr>
</tbody>
</table>

```c
43  a[0] = 0;
44  # pragma omp parallel
45  {
46  #   pragma omp for nowait
47    for (i=1; i<N; i++)
48      { a[i] = 3.0*i*(i+1); }
49    } /* end of omp for nowait */
51  #   pragma omp for
52    for (i=1; i<N; i++)
53      { b[i] = a[i] - a[i-1]; }
54  } /* end of omp parallel */
```

Solution: Remove "nowait" see appendix

Höchstleistungsrechenzentrum Stuttgart
Example 3 – source code

```c
68  # pragma omp parallel for
69    for (i=1; i<N; i++)
70    {
71        x = sqrt(b[i]) - 1;
72        a[i] = x*x + 2*x + 1;
73    } /* end of omp parallel for */
```

What's wrong?

How to solve?
**Example 3 – analysis**

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Write -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp</td>
<td>Memory <strong>write</strong> of <em>x</em> at &quot;demo_with_bugs.c&quot;:71 conflicts with a prior memory <strong>write</strong> of <em>x</em> at &quot;demo_with_bugs.c&quot;:71 (output)</td>
<td>&quot;demo_with_bugs.c&quot;:71</td>
<td>&quot;demo_with_bugs.c&quot;:71</td>
</tr>
<tr>
<td>5</td>
<td>Read -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp</td>
<td>Memory <strong>write</strong> of <em>x</em> at &quot;demo_with_bugs.c&quot;:71 conflicts with a prior memory <strong>read</strong> of <em>x</em> at &quot;demo_with_bugs.c&quot;:72 (anti dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:71</td>
<td>&quot;demo_with_bugs.c&quot;:71</td>
</tr>
</tbody>
</table>

```c
68  # pragma omp parallel for
69    for (i=1; i<N; i++)
70    {
71        x = sqrt(b[i]) - 1;
72        a[i] = x*x + 2*x + 1;
73    } /* end of omp parallel for */
```

*See appendix*
Example 4 – source code

84     f = 2;
85     # pragma omp parallel for private(f,x)
86     for (i=1; i<N; i++)
87     {
88         x = f * b[i];
89         a[i] = x - 7;
90     } /* end of omp parallel for */
91     a[0] = x;

What’s wrong?

How to solve?
### Example 4 – analysis

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>OpenMP – undefined in access</td>
<td>Caution</td>
<td>1</td>
<td>omp parallel region</td>
<td>OpenMP – the access at &quot;demo_with__bugs.c&quot;: 88 is undefined, the expected value was defined at &quot;demo_with__bugs.c&quot;: 84 in the serial execution</td>
<td>&quot;demo_with_bugs.c&quot;: 84</td>
<td>&quot;demo_with_bugs.c&quot;: 88</td>
</tr>
<tr>
<td>7</td>
<td>OpenMP – undefined in the serial code (original prog.)</td>
<td>Warning</td>
<td>1</td>
<td>&quot;demo_with_bugs.c&quot;: 17</td>
<td>OpenMP – undefined in the serial code (original program) at &quot;demo_with_bugs.c&quot;: 91 with &quot;demo_with_bugs.c&quot;: 88</td>
<td>&quot;demo_with_bugs.c&quot;: 88</td>
<td>&quot;demo_with_bugs.c&quot;: 91</td>
</tr>
</tbody>
</table>

```c
84  f = 2;
85  # pragma omp parallel for private(f,x)
86  for (i=1; i<N; i++)
87  {
88      x = f * b[i];
89      a[i] = x - 7;
90  } /* end of omp parallel for */
91  a[0] = x;
```

see appendix
Example 5 – source code

```
101  sum = 0;
102  # pragma omp parallel for
103  for (i=1; i<N; i++)
104  {
105      sum = sum + b[i];
106  } /* end of omp parallel for */
```

What's wrong?

How to solve?
### Example 5 – analysis

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Read -&gt; Write data-race</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory <strong>write</strong> of sum at &quot;demo_with_bugs.c&quot;:105 conflicts with a prior memory <strong>read</strong> of sum at &quot;demo_with_bugs.c&quot;:105 (anti dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
</tr>
<tr>
<td>9</td>
<td>Write -&gt; Read data-race</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory <strong>read</strong> of sum at &quot;demo_with_bugs.c&quot;:105 conflicts with a prior memory <strong>write</strong> of sum at &quot;demo_with_bugs.c&quot;:105 (flow dependence)</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
</tr>
<tr>
<td>10</td>
<td>Write -&gt; Write data-race</td>
<td>Error</td>
<td>98</td>
<td>omp for</td>
<td>Memory <strong>write</strong> of sum at &quot;demo_with_bugs.c&quot;:105 conflicts with a prior memory <strong>write</strong> of sum at &quot;demo_with_bugs.c&quot;:105 (output)</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
<td>&quot;demo_with_bugs.c&quot;:105</td>
</tr>
</tbody>
</table>

```c
101    sum = 0;
102    # pragma omp parallel for
103    for (i=1; i<N; i++)
104    {
105        sum = sum + b[i];
106    } /* end of omp parallel for */
```

See appendix.
### Example 6 – nothing wrong – unexpected “Caution”

```c
129  sum = 0;
130  # pragma omp parallel private(psum)
131  {
132      psum = 0;
133  # pragma omp for
134      for (i=1; i<N; i++)
135          {
136              psum = psum + b[i];
137          } /* end of omp for */
138  # pragma omp critical
139  {
140      sum = sum + psum;
141          } /* end of omp critical */
142  } /* end of omp parallel */
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>OpenMP - cannot be private</td>
<td>Cautio n</td>
<td>98</td>
<td>omp for</td>
<td>OpenMP -- The access at &quot;demo_with_bugs.c&quot;:136 cannot be private because it expects the value previously defined at &quot;demo_with_bugs.c&quot;:136 in the serial execution</td>
</tr>
</tbody>
</table>

**1st Access [Best]**
- "demo_with_bugs.c":136

**2nd Access [Best]**
- "demo_with_bugs.c":136
Example 7 – source code

```c
omp_set_dynamic(0);
b[0] = 0;
sum = 0;
#pragma omp parallel private(psum, num_threads)
{
    #ifdef _OPENMP
        num_threads = omp_get_num_threads();
    #else
        num_threads = 1;
    #endif
    psum = 0;
    #pragma omp for schedule(static,(N-1)/num_threads+1)
        for (i=0; i<N; i++)
        {
            psum = psum + b[i];
        } /* end of omp for */
    #pragma omp for ordered schedule(static,1)
        for (i=0; i<num_threads; i++)
        {
            #pragma omp ordered
                sum = sum + psum;
        } /* end of omp for */
} /* end of omp parallel */
```

Nothing is wrong. But there are unexpected warnings!

From OpenMP Introduction:
Parallelization trick to achieve reproducible & efficient reduction results if OMP_NUM_THREADS is fixed

Schedule is fixed

Reduction ordering is fixed
Example 7 – analysis

- Three problems
  - Compile time error  ➞  see A on next slides
  - Runtime error  ➞  see B on next slides
  - Analysis error: a caution although the code is correct → may be ignored (same as in Example 6)

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenMP - cannot be private</td>
<td>Cautio</td>
<td>99</td>
<td>omp for</td>
<td>OpenMP -- The access at &quot;demo_with_bugs.c&quot;:178 cannot be private because it expects the value previously defined at &quot;demo_with_bugs.c&quot;:178 in the serial execution</td>
</tr>
</tbody>
</table>

```cpp
174
175
psum = 0;
#pragma omp for schedule(static, (N-1)/num_threads +1)
for (i=0; i<N; i++)
{
    psum = psum + b[i];
} /* end of omp for */
```
Example 7 – analysis – Compile time error

- Compile time error:
  - `Demo_with_bugs.c(167): warning #1378: Variable "num_threads" in OpenMP schedule clause should appear on shared list
    # pragma omp parallel private(psum, num_threads)

  
  ```c
  # pragma omp parallel private(psum, num_threads)
  
  168 169
  170
  
  171 172
  173
  174
  
  # pragma omp for schedule(static,(N-1)/num_threads+1)
  ```

- Same value in `num_threads` on all threads, but compiler doesn’t know!
Example 7 – analysis – Compile time error cont’d

• Work-around:
  `num_threads` as **automatic variable**, defined in the parallelized block

```c
#pragma omp parallel
private(psum, num_threads)
{
  int num_threads;
  #ifdef _OPENMP
  num_threads=omp_get_num_threads();
  #else
  num_threads=1;
  #endif
  ...
  #pragma omp for
  schedule(static, (N-1)/num_threads + 1)
  for ...
```

• Don’t use `shared(num_threads)`
  because assignment
  `num_threads = omp_get_num_threads();`
  would cause a write-write race condition
  ➔ cache-line false sharing
Example 7 – analysis – Compile time error  

Solution:

- Calculate `num_threads` with an own parallel region
- before numerical parallel region (lines 167-186)

```c
int num_threads;
omp_set_dynamic(0);
#pragma omp parallel
{
#pragma omp single
{
    #ifdef _OPENMP
        num_threads=omp_get_num_threads();
    #else
        num_threads=1;
    #endif
    } /* end of omp single */
} /* end of omp parallel */
```

```
167
#pragma omp parallel
private(psum)
shared(num_threads)
{
    psum = 0;
#pragma omp for schedule(stat, (N-1)/num_threads+1)
    for (i=0; i<N; i++)
    {
        ...
    }
    /* calculation of num_threads is removed */
```
Example 7 – analysis – Run time error

- Run time error
  - `omp_get_num_threads()` returns 2
  - but only 1 thread is used!
  - `sum = sum + psum` executed twice → wrong result
  - Feature (not a bug) of Intel® Thread Checker

- Solution:
  - Calculate `num_threads` at the beginning and without OpenMP functions

```c
int num_threads;
omp_set_dynamic(0);
num_threads=0;
#pragma omp parallel
{
    #pragma omp critical
    {
        num_threads++;
    }  /* end of omp critical */
}  /* end ofomp parallel */
```

Only ITC version ≤ 3.0

ITC version ≥ 3.1: `omp_get_num_threads()` returns 1
Example 7 – solution to A and B

```c
int num_threads;
omp_set_dynamic(0);
num_threads=0;
#pragma omp parallel
{
  #pragma omp critical
    { num_threads++;
    } /* end of omp critical */
} /* end of omp parallel */
b[0] = 0;
sum = 0;
#pragma omp parallel private(psum)
{ psum = 0;
  #pragma omp for schedule(static,(N-1)/num_threads+1)
    for (i=0; i<N; i++)
    {
      psum = psum + b[i];
    } /* end of omp for */
  #pragma omp for ordered schedule(static,1)
    for (i=0;i<num_threads;i++)
    {
      #pragma omp ordered
        sum = sum + psum;
    } /* end of omp for */
} /* end of omp parallel */
```
### Stack info

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 1</td>
<td>Thread termination at &quot;demo_with_bugs.c&quot;:17 - includes stack allocation of 10485760 and use of 2416 bytes</td>
<td>&quot;demo_with_bugs.c&quot;:17</td>
<td>&quot;demo_with_bugs.c&quot;:17</td>
</tr>
</tbody>
</table>
Restrictions in TCI mode

• Runtime check
  – Therefore error detection only in software branches that are executed
• Often more than **300 times slower** than serial execution and more than **20 times memory consumption**
  – Use small number of iterations
  – Use small data set
  – But large & complex enough that all software branches are executed
Domain decomposition

```c
int num_threads;
omp_set_dynamic(0);
#pragma omp parallel
{
#pragma omp single
{
    num_threads=omp_get_num_threads();
} /* end of omp single */

// Create a domain decomposition
with num_threads domains...

#pragma omp for
for (domain_no=0;
domain_no < num_threads;
domain_no++)
{
    Process_domain( D[domain_no] );
} /* end of omp for */
} /* end of omp parallel */
```

ITC version ≥ 3.1:
- `omp_get_num_threads()` returns 1
- Loop is checked, although # iterations == 1
- Recommendation: use num_threads=13 or any other meaningful value

ITC version ≤ 3.0:
- `omp_get_num_threads()` returns 2
- Although internally, only 1 thread is used!
- Loop is checked, because # iterations == 2
End of checking

• When “no errors, cautions, warnings” are reported, multi-threaded run is assured to be free of semantical OpenMP parallelization errors (i.e., data-races, …, as shown in Examples 1-5), but only in software branches touched by the checked simulation run.

• *We are not sure whether this statement is correct and formally proven.*
Other Products

- Sun Studio Express: Data Race Detection Tool (DRDT)
  - Automatic OpenMP source code instrumentation
  - Analysis of data races in a parallel execution

- Jprobe Threadanalyzer from Quest Software
  - Detects data races in Java programs
  - [www.quest.com/jprobe](http://www.quest.com/jprobe)

- Visual Threads from HP
  - Detects data races in POSIX threaded programs

- Helgrind of the Valgrind tool suite
  - Open source data race detection tool
  - For POSIX threaded programs on Linux
Acknowledgements

- This lecture is based on the Assure presentation by Hans-Joachim Plum, Pallas GmbH
- The exercises *race1/2* and *conflict* were developed by Matthias Müller
- The Intel® Thread Checker 3.0 beta was installed by Dmitri Chubarov and Bettina Krammer, the Intel® Thread Checker 3.0 by Danny Sternkopf
- Thanks to James Cownie and Paul Petersen for their review comments.
- Thanks to Matthias Lieber, ZIH for the update on the call stack depth.
- Thanks to Dmitri Chubarov hints about non-detectable “races”
Summary

- **Intel® Thread Checker** finds the locations of race conditions, but the **programmer** must detect the reasons!
- Source code instrumentation (TCI) – executed with 1 thread – returns an important error report
- Programmer has to **eliminate** all these errors – or **must be sure** that the reported error can be ignored
- Binary instrumentation works – executed in parallel with multiple threads – should be used if TCI is not applicable (or after all TCI reports are resolved)

It is **absolutely** necessary to **verify** OpenMP parallelizations with a **race-condition detection tool**.

Currently\(^1\) we see on the market only one OpenMP race-condition detection tool that can be used under Linux and Windows:

This is the Intel® Thread Checker

\(^1\) June 30, 2006
Intel® Thread Checker – Practical
(on cacau.hww.de)

- module load itt/tcheck-3.0 (setup, only once)
- cd ~/OpenMP/#nr/pitfalls/
- Compiling the application together with the Intel® Thread Checker:
 icc -tcheck -openmp -g -o my_prog my_prog.c
  or
  ifort -tcheck -openmp -g -o my_prog my_prog.f
- Executing & analyzing the application together with the Intel® Thread Checker
  on one processor, detecting all race conditions:
  tcheck_cl -w 90 -o my_prog.txt ./my_prog
- with myprog = conflict, race1, and race2 (and as additional exercise: demo_with_bugs.c)
- Tasks:
  - find the reasons in all 3 examples
  - correct the source (without modifying the numerical semantics)
  - verify again with Intel® Thread Checker
  - until ”tcheck_cl” does not report any further error, caution, warning, …
Appendix

- Restrictions on OpenMP library functions in TCI mode
- Modes: TCI, BIN and MIXED
- Example 8 – Mixed mode compilation
- Binary instrumentation and parallel execution
- When to use … / Recommendation
- Thread Checker Requirements
- History
- Further information and reading / References
- Solutions to Examples 1-5
Restrictions on OpenMP library routines in TCI mode

<table>
<thead>
<tr>
<th>Returned Values</th>
<th>Outside Par. Regions</th>
<th>In Parallel Regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omp_get_thread_num()</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp_get_num_threads()</td>
<td>1</td>
<td>1 (ITC version ≥ 3.1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 (ITC version ≤ 3.0)</td>
</tr>
</tbody>
</table>

Appendix
- Library functions
  - TCI, BIN and MIXED
  - Example 8
  - Binary instrumentation
  - Recommendation
  - Requirements
  - History
  - Further information
  - Solutions to Exa. 1-5
## Restrictions on OpenMP library routines

<table>
<thead>
<tr>
<th>Function</th>
<th>in parallel</th>
<th>regions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_in_parallel()</code></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>omp_get_max_threads()</code></td>
<td>1 (see next slides)</td>
<td>1 (ITC version ≥ 3.1)</td>
</tr>
<tr>
<td></td>
<td>2 (see next slides)</td>
<td>2 (ITC version ≤ 3.0)</td>
</tr>
<tr>
<td><code>omp_get_num_procs()</code></td>
<td>1 (see next slides)</td>
<td>1 (ITC version ≥ 3.1)</td>
</tr>
<tr>
<td></td>
<td>2 (see next slides)</td>
<td>2 (ITC version ≤ 3.0)</td>
</tr>
<tr>
<td><code>omp_get_dynamic()</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>omp_get_nested()</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>omp_get_wtick()</code></td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td><code>omp_get_wtime()</code></td>
<td>call counter</td>
<td>call counter</td>
</tr>
</tbody>
</table>
Restrictions on OpenMP library routines

- **Set functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_set_num_threads()</td>
<td>given value is returned by subsequent calls to omp_get_max_threads(), omp_get_num_procs(), and inside of parallel regions: omp_get_num_threads(). <strong>Real number of threads is still 1!</strong></td>
</tr>
<tr>
<td>omp_set_dynamic()</td>
<td>returned by subsequent omp_get_dynamic()</td>
</tr>
<tr>
<td>omp_set_nested()</td>
<td>ignored, i.e., omp_get_nested() returns always 0</td>
</tr>
</tbody>
</table>

- **Lock functions**

  e.g., deadlocks due to wrong nesting of different locks are reported

```c
int main(void){
    double x=0, y=0;  int i;  omp_lock_t lock_x, lock_y;
    omp_init_lock(&lock_x);  omp_init_lock(&lock_y);
    #pragma omp parallel for shared(x,y)
    for(i=1; i<=N; i++){
        if (i < 0.3*N) {
            omp_set_lock(&lock_x);  x = x + i;
            omp_set_lock(&lock_y);  y = y + i;
            omp_unset_lock(&lock_y);
            omp_unset_lock(&lock_x);
        } else {
            omp_set_lock(&lock_y);  y = y + i;
            omp_set_lock(&lock_x);  x = x + i;
            omp_unset_lock(&lock_x);
            omp_unset_lock(&lock_y);
        } /*endif*/
    } /*end for*/
}
```
Modes

- **Thread count independent analysis mode** = Thread Checker  
  - Source Code Instrumentation 
    - Compilation with 
      icc / ifort –g –openmp –tcheck
  - Simulation approach = thread-parallel execution 
    - Binary Instrumentation 
    - Compilation with 
      icc / ifort –g –openmp (without –tcheck)
- **Simulation approach** = thread-parallel execution 
  - Binary Instrumentation 
  - Compilation with 
    icc / ifort –g –openmp (without –tcheck)
- **Mixed mode**, e.g., user application with TCI + all libraries with BIN
- **Execution** always with tcheck_cl

Appendix
- Library functions
- TCI, BIN and MIXED
- Example 8
- Binary instrumentation
- Recommendation
- Requirements
- History
- Further information
- Solutions to Exa. 1-5
Example 8 – Mixed mode compilation – Main program

caller.c

14 void scalar_mult_vector(float *b, float c, float *a, int n); /* B=c*A */
15 void scalar_plus_vector(float *b, float c, float *a, int n); /* B=c+A */

17 int main(void)
18 { float a[N], b[N], sum, sum_expected; int i;
...
26   for (i=0; i<N; i++) { a[i]=1; b[i]=999; }
27 # pragma omp parallel
28   {
29 #   pragma omp sections
30     {
31 #     pragma omp section
32       {
33         scalar_mult_vector(a, 3, a, N); /* A=3*A */
34       } /* end of omp section */
35 #     pragma omp section
36       {
37         scalar_plus_vector(b, 4, a, N); /* B=4+A */
38       } /* end of omp section */
39 #     pragma omp section
40       {
41         scalar_mult_vector(a, 5, b, N); /* A=5*B */
42       } /* end of omp section */
43     } /* end of omp sections */
44   } /* end of omp parallel */
51 }
Example 8 – Mixed mode compilation – Library routines

mylib.c

```c
1 #ifdef _OPENMP
2 #include <omp.h>
3 #endif
4
5 void scalar_mult_vector(float *b, float c, float *a, int n); /* B=c*A */
6 void scalar_plus_vector(float *b, float c, float *a, int n); /* B=c+A */
8
9 void scalar_mult_vector(float *b, float c, float *a, int n) /* B=c*A */
10 { int i;
11   for (i=0; i<n; i++) b[i] = c * a[i];
12 }
13
14 void scalar_plus_vector(float *b, float c, float *a, int n) /* B=c+A */
15 { int i;
16   for (i=0; i<n; i++) b[i] = c + a[i];
17 }
```
Example 8 – Mixed mode compilation

• Mixed compilation
  - `icc -tcheck -openmp -g -c caller.c` → **TCI**
  - `icc -g -c mylib.c` → **BIN**
  - `icc -tcheck -openmp -g -o a.out caller.o mylib.o`

  not required, i.e., works with any library

• Execution and analysis in (projection technology)
  - `tcheck_cl -w 90 ./a.out` → **TCI**

• Call stack (e.g., with depth = 4)
  - `tcheck_cl -s 4 threadchecker.thr`

  With TCHECK 3.0.22356 and older [see `tcheck_cl -v`], one must examine each event_id separately:
  - `tcheck_cl -s 1 threadchecker.thr`
  - `tcheck_cl -s 2 threadchecker.thr`
  - ...

---

Höchstleistungsrechenzentrum Stuttgart
Example 8 – analysis → call stack

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>Context [Best]</th>
<th>Description</th>
<th>1st Access [Best]</th>
<th>2nd Access [Best]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write -&gt; Read</td>
<td>Error</td>
<td>100</td>
<td>omp for</td>
<td>Memory read at &quot;my_lib.c&quot;:16 conflicts with a prior memory write at &quot;my_lib.c&quot;:11 (flow dependence)</td>
<td>&quot;my_lib.c&quot;:11</td>
<td>&quot;my_lib.c&quot;:16</td>
</tr>
</tbody>
</table>

> tcheck_cl -s 1 threadchecker.thr

First access stack
#0 my_lib.c : 11 , scalar_mult_vector , caller
#1 caller.c : 33 , main , caller
#2 : 4294967295 , call_gmon_start , caller

Second access stack
#0 my_lib.c : 16 , scalar_plus_vector , caller
#1 caller.c : 37 , main , caller
#2 : 4294967295 , call_gmon_start , caller
Example 8 – analysis

Description

Memory read at "my_lib.c":16 conflicts with a prior memory write at "my_lib.c":11 (flow dependence)

First access stack
#0 my_lib.c : 11 , scalar_mult_vector , caller
#1 caller.c : 33 , main , caller
#2 : 4294967295 , call_gmon_start , caller

Second access stack
#0 my_lib.c : 16 , scalar_plus_vector , caller
#1 caller.c : 37 , main , caller
#2 : 4294967295 , call_gmon_start , caller

Write-Read race condition

caller.c
31  #    pragma omp section
32  {
33   scalar_mult_vector(a, 3, a, N); /* A=3*A */
34  } /* end of omp section */
35  #    pragma omp section
36  {
37   scalar_plus_vector(b, 4, a, N); /* B=4+A */
38  } /* end of omp section */

my_lib.c
9  void scalar_mult_vector(float *b, float c, float *a, int n) /* B=c*A */
10  { int i;
11   for (i=0; i<n; i++) b[i] = c * a[i];
12  }
14  void scalar_plus_vector(float *b, float c, float *a, int n) /* B=c+a */
15  { int i;
16   for (i=0; i<n; i++) b[i] = c + a[i];
17  }
Binary instrumentation and parallel execution

• Compilation:
  only with Intel\textsuperscript{\textregistered} compilers, as usual, but with \texttt{-g} option:
  \begin{verbatim}
  icc -openmp -g -o my_prog my_prog.c
  ifort -openmp -g -o my_prog my_prog.f
  \end{verbatim}

• Parallel execution and analysis with \texttt{tcheck_cl}
  \begin{verbatim}
  export OMP_NUM_THREADS=4
  tcheck_cl -w 90 -o my_prog.txt ./my_prog
  \end{verbatim}

• Problems with Intel\textsuperscript{\textregistered} Thread Checker 3.0:
  – Reports many non-existing errors
  – Cannot find problems, e.g., as in Example 4
  – Cannot report variable names
  – Wrong count-values

\begin{itemize}
  \item \texttt{But may help if TCI is not applicable}
\end{itemize}
When to use …

... Source Code Instrumentation

• Source instrumentation is required for analysis when binary instrumentation is not available.
• When you want to run the instrumented program outside of the VTune™ Performance Environment. For example, use source instrumentation for a server application.

... Binary Instrumentation

• If you do not have access to an compiler.
• If you do not want to rebuild your application, for example, because it might take many hours to do so.
• If you do not have access to the source code.

From user's manual tcheck.chm
Section Instrumenting Code for Intel(R) Thread Checker
Recommendation

- Use Source Code Instrumentation = Projection Mode
  = Thread Count Independent Mode (TCI)
  - If this mode is applicable to your application
  - TCI provides much more information
    and much more value to the users of OpenMP
  - But it can not be used with all applications

- If TCI is not applicable,
  (or after all TCI reports are resolved,)
  then use
  Binary Instrumentation = Thread Count Dependent Mode
Thread Checker Requirements

- Intel® Thread Checker 3.0 beta
  - Linux or Windows XP Professional
  - Intel® processors
  - Intel® compiler
  - Intel® Vtune™

- On Windows XP Professional:
  - Additional GUI interface

→ my personal recommendation: Use text output with line-numbers

Appendix
- Library functions
- TCI, BIN and MIXED
- Example 8
- Binary instrumentation
- Recommendation
- **Requirements**
  - History
  - Further information
  - Solutions to Exa. 1-5
History

- P. M. Petersen: 
  **Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques.**
  [http://citeseer.ist.psu.edu/petersen93evaluation.html](http://citeseer.ist.psu.edu/petersen93evaluation.html)

- **Assure**
  - a KAP/pro tool
  - it was available for most shared memory platform
  - sold to Intel \(\rightarrow\) basis for Intel\(^\circledR\) Thread Checker

- Paul Petersen, Sanjiv Shah: 
  **OpenMP Support in the Intel\(^\circledR\) Thread Checker.**
  WOMPAT 2003, pp 1-12.

- **Intel\(^\circledR\) Thread Checker**
  - Product for Windows since 2005
  - Beta-test for Linux 3/2006
Further information and reading

- `tcheck_cl` without arguments shows list of options
- `export TC_OPTIONS=help
tcheck_cl ./my_prog` will return a list of additional `TC_OPTIONS` and output is mixed with run-time checking analysis, e.g.

  ```
  [Intel(R) Thread Checker Report: OpenMP undefined access]
  Exa.4: a[0] computed= 1188.0, expected= 1188.0, difference= 0.00000
  [Intel(R) Thread Checker Report: OpenMP undefined access]
  ```

- `tcheck.chm` contains the user manual (Windows?)
- Intel Threading Tools:
- At HLRS:
  `http://www.hlrs.de/organization/amt/services/tools/debugger/itc/`
References

• United States Patent 6,286,130 (September 4, 2001)
  David K. Poulsen, Paul M. Petersen, Sanjiv M. Shah:
  *Software implemented method for automatically validating the correctness of parallel computer programs*

  – This patent describes implementation and goals of the projection mode

  "We claim:
  1. A method for detecting individual errors in a parallel computer program by translating a parallel computer program into a sequential computer program, said method comprising the steps of:
     • identifying a parallel computer program having at least one parallelism specification;
     • generating a corresponding sequential computer program to the parallel computer program by ignoring said at least one parallelism specification contained in the parallel computer program;
     • adding to said corresponding sequential computer program at least one first instruction, to generate at least one first trace event, said at least one first instruction relating to said corresponding sequential computer program, and at least one second instruction, to generate at least one second trace event, said at least one second instruction based upon the ignored at least one parallelism specification
     • logically partitioning the sequential computer program into at least one disjoint group based upon the at least one second trace event, said at least one disjoint group comprising at least one of the at least one first trace events; and
     • executing only said sequential computer program a single time, and analyzing said at least one disjoint group of said at least one first trace event based on types of second trace events used to partition said at least one first trace event to detect and report each precise semantic inconsistency between said parallel computer program and said corresponding sequential computer program, thereby detecting one or more semantic inconsistencies associated with a plurality of different executions of the parallel computer program."
References

- Utpal Banerjee, Brian Bliss, Zhiqiang Ma, Paul Petersen (Intel): *Unraveling Data Race Detection in the Intel Thread Checker*
  - This paper describes the theory used in the simulation approach.
  - References to other products (see next slide)

From Section 5.1:
„Happens-before requires logging the history of accesses to every shared memory variable. Instead of a complete access history, we only keep the most recently read and most recently written accesses. The tool is able to catch most data races, but it doesn’t guarantee to catch all data races in a single run. We believe that, in practice, catching as many races of a program run as possible in a detailed manner is better than catching all races of a program run in a brief way.“

Highlighted by the author of this slide.
Solutions to Examples 1-5

Example 1:
• Two separate loops \( \rightarrow \) will cause bad cache reuse!
or
• Re-computing of \( a[i-1] \), i.e., \( b[i] = a[i] - 2.0*(i-1)*(i-2) \);

Example 2: Remove “nowait”

Example 3: Add “private(x)”

Example 4: Use “firstprivate(f) lastprivate(x)” instead of “private(f,x)”

Example 5: Add “reduction(+:sum)”