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

Sep 11, 2006
Used version of Intel® Thread Checker under Linux: 3.0

Overview

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 7 – 25
• Restrictions in TCI mode slides 26 – 31
• End of checking slide 32
• TCI and Binary Mode slides 33 – 41
• Wrap-up and Summary slides 42 – 46
• Practical slide 47
• Appendix slides 48 – 51
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
  - Used on most slides

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

---

**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
Verifying an OpenMP Parallelization with the Intel® Thread Checker

Processing

Source program
my_prog.c or my_prog.f

Example:
module load itt/tcheck-3.0

compile with options
-tccheck -openmp -g

Executable
my_prog

execute with command
tcheck_cl -w 90 -o my_prog.txt ./my_prog

Thread Checker
Rolf Rabenseifner

Output – Example with demo_with_bugs.c

<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>Exa.1</td>
<td>Write &lt; 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>demo.c:29</td>
<td>demo.c:30</td>
</tr>
<tr>
<td>Exa.2</td>
<td>Write &lt; Read</td>
<td>Error</td>
<td>197</td>
<td>omp par alloc.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>demo.c:49</td>
<td>demo.c:54</td>
</tr>
<tr>
<td>Exa.3</td>
<td>Write &lt; 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 write of x at &quot;demo.c&quot;:71 (output dependence)</td>
<td>demo.c:71</td>
<td>demo.c:71</td>
</tr>
<tr>
<td>Exa.4</td>
<td>Write &lt; Write</td>
<td>Caution</td>
<td>1</td>
<td>omp par alloc.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 exec</td>
<td>demo.c:84</td>
<td>demo.c:88</td>
</tr>
<tr>
<td>Exa.5</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>demo.c:88</td>
<td>demo.c:91</td>
</tr>
<tr>
<td>Exa.6</td>
<td>Read &lt; 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 write of sum at &quot;demo.c&quot;:105 (antidependence)</td>
<td>demo.c:105</td>
<td>demo.c:105</td>
</tr>
<tr>
<td>Exa.7</td>
<td>Read &lt; 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 write of sum at &quot;demo.c&quot;:105 (flow dependence)</td>
<td>demo.c:105</td>
<td>demo.c:105</td>
</tr>
<tr>
<td>Exa.8</td>
<td>- 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>demo.c:136</td>
<td>demo.c:136</td>
</tr>
<tr>
<td>Exa.9</td>
<td>- 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>demo.c:178</td>
<td>demo.c:178</td>
</tr>
<tr>
<td>Exa.10</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 5416 bytes</td>
<td>demo.c:17</td>
<td>demo.c: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 2 – source code

```c
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</th>
<th>Count</th>
<th>Context</th>
<th>Description</th>
<th>1st Access</th>
<th>2nd Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Write data-race</td>
<td>Error</td>
<td>197</td>
<td>omp parallel region</td>
<td>Memory <code>read</code> of <code>a[]</code> at <code>demo_with_bugs.c</code>:54 conflicts with a prior <code>memory write</code> of <code>a[]</code> at <code>demo_with_bugs.c</code>:49</td>
<td><code>demo_with_bugs.c</code>:54</td>
<td><code>demo_with_bugs.c</code>:49</td>
</tr>
<tr>
<td>3</td>
<td>Read data-race</td>
<td>Error</td>
<td>197</td>
<td>omp parallel region</td>
<td>Memory <code>read</code> of <code>a[]</code> at <code>demo_with_bugs.c</code>:49 conflicts with a prior <code>memory write</code> of <code>a[]</code> at <code>demo_with_bugs.c</code>:54</td>
<td><code>demo_with_bugs.c</code>:49</td>
<td><code>demo_with_bugs.c</code>:54</td>
</tr>
</tbody>
</table>

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 */
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</th>
<th>Count</th>
<th>Context</th>
<th>Description</th>
<th>1st Access</th>
<th>2nd Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Write -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp</td>
<td>Memory write of x at <code>demo_with_bugs.c</code>:71 conflicts with a prior memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data-race</td>
<td></td>
<td></td>
<td>for</td>
<td>write of x at <code>demo_with_bugs.c</code>:71 (output)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>demo_with_bugs.c</code>:71</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Read -&gt; Write</td>
<td>Error</td>
<td>98</td>
<td>omp</td>
<td>Memory write of x at <code>demo_with_bugs.c</code>:71 conflicts with a prior memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data-race</td>
<td></td>
<td></td>
<td>for</td>
<td>read of x at <code>demo_with_bugs.c</code>:72 (anti dependence)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>demo_with_bugs.c</code>:72</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
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 */
```
### Example 4 – source code

```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;
```

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</th>
<th>Description</th>
<th>1st Access</th>
<th>2nd Access</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>demo_with_bugs.c&quot;:88</td>
<td>demo_with_bugs.c&quot;:84</td>
</tr>
<tr>
<td>7</td>
<td>OpenMP -- undefined in the serial code (original prog.)</td>
<td>Warning</td>
<td>1</td>
<td>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>demo_with_bugs.c&quot;:91</td>
<td>demo_with_bugs.c&quot;:88</td>
</tr>
</tbody>
</table>

Solution: use "firstprivate(f) lastprivate(x)" instead of "private(f,x)"

see appendix
Example 5 – source code

```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 */
```

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 omp for</td>
<td>Memory write of sum at *demo_with_bugs.c&quot;:105 conflicts with a prior memory read of sum at *demo_with_bugs.c&quot;:105 (anti dependence)</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Write -&gt; Read data-race</td>
<td>Error</td>
<td>98 omp for</td>
<td>Memory read of sum at *demo_with_bugs.c&quot;:105 conflicts with a prior memory write of sum at *demo_with_bugs.c&quot;:105 (flow dependence)</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Write -&gt; Write data-race</td>
<td>Error</td>
<td>98 omp for</td>
<td>Memory write of sum at *demo_with_bugs.c&quot;:105 conflicts with a prior memory write of sum at *demo_with_bugs.c&quot;:105 (output)</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td>*demo_with_bugs.c&quot;:105</td>
<td></td>
</tr>
</tbody>
</table>

```
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 */
```
### Example 6 – nothing wrong – unexpected “Caution”

```c
sum = 0;
#pragma omp parallel private(psum)
{
    psum = 0;
    #pragma omp for
    for (i=1; i<N; i++)
    {
        psum = psum + b[i];
    }
    #pragma omp critical
    {
        sum = sum + psum;
    }
    /* end of omp critical */
    #pragma omp for
    for (i=1; i<N; i++)
    {
        psum = psum + b[i];
    }
    /* end of omp for */
}
/* end of omp parallel */
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Severity</th>
<th>Count</th>
<th>Context</th>
<th>1st Access</th>
<th>2nd Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>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_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>
<td></td>
</tr>
</tbody>
</table>

---

### 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++)
    {
        sum = sum + psum;
    }
    /* end of omp parallel */
}
/* end of omp parallel */
```

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

Schedule is fixed

Reduction ordering is fixed

Nothing is wrong. But there are unexpected warnings!
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)

```c
174 psum = 0;
175 #pragma omp for schedule(static,(N-1)/num_threads+1)
176 for (i=0; i<N; i++)
177 {
178   psum = psum + b[i];
179 }
/* end of omp for */
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Count</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</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>

Example 7 – analysis – Compile time error A

- 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
167 #pragma omp parallel private(psum, num_threads)
168 {
169 # ifdef _OPENMP
170 num_threads=omp_get_num_threads();
171 # else
172 num_threads=1;
173 # endif
175 #pragma omp for schedule(static,(N-1)/num_threads+1)
176 for ...
```

- 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
167 #pragma omp parallel private(psum,-num_threads)
168 {int num_threads;
169 #ifdef __OPENMP
170 num_threads=omp_get_num_threads();
171 #else
172 num_threads=1;
173 #endif...
175 #pragma omp for schedule(static,(N-1)/num_threads+1)
176 for ...
```

• Don't use shared(num_threads)
  – because assignment
    ```c
    num_threads = omp_get_num_threads();
    ```
  would cause a write-write race condition
  – ➔ cache-line false sharing

Example 7 – analysis – Compile time error cont’d

• 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 */
```

```c
167 #pragma omp parallel private(psum) shared(num_threads)
168 { ...
169-173 /* calculation of num_threads is removed */
174 psum = 0;
175 #pragma omp for schedule(static,(N-1)/num_threads+1)
176 for (i=0; i<N; i++)
177 ...
```
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 \(\rightarrow\) 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 of omp parallel */
```

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 */
    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 */
```
Verifying an OpenMP Parallelization with the Intel® Thread Checker

<table>
<thead>
<tr>
<th>ID</th>
<th>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>13</td>
<td>Thread termination</td>
<td>Info</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 on OpenMP library routines in TCI mode

- Get functions
  - returned values outside par.regions in parallel regions
    - omp_get_thread_num() 0 0
    - omp_get_num_threads() 1 2
      - this value is wrong, see Example 7, next slides

TCI: Restrictions in TCI mode

- Overview
- Seven Examples
- Restrictions in TCI mode
- End of checking
- TCI and Binary Mode
- Wrap-up and Summary
- Practical
- Appendix
Restrictions on OpenMP library routines

<table>
<thead>
<tr>
<th>Get functions</th>
<th>returned values outside par.regions</th>
<th>in parallel regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_in_parallel()</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>omp_get_max_threads()</td>
<td>2 (see next slides)</td>
<td>2 (see next slides)</td>
</tr>
<tr>
<td>omp_get_num_procs()</td>
<td>2 (see next slides)</td>
<td>2 (see next slides)</td>
</tr>
<tr>
<td>omp_get_dynamic()</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp_get_nested()</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>omp_get_wtick()</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>omp_get_wtime()</td>
<td>call counter</td>
<td>call counter</td>
</tr>
</tbody>
</table>

Number of threads = 2

Reported behavior in “Thread Count Independent Mode” (TCI = projection mode):

- omp_get_num_threads(), omp_get_max_threads(), and omp_get_num_procs() \( \rightarrow \text{return 2} \)
- Internally only one thread is used
- But race condition checking is done with number of threads = number of loop iterations or sections! \( \rightarrow \text{maximally parallel!} \)
Number of threads = 2 – Reasons

```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 */
```

This programming style needs num_threads > 1

Reason: No race detection if num_threads == 1

Restrictions on OpenMP library routines

- **Set functions**
  - `omp_set_num_threads()`: 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()`. Real number of threads is still 1!
  - `omp_set_dynamic()`: returned by subsequent `omp_get_dynamic()`
  - `omp_set_nested()`: ignored, i.e., `omp_get_nested()` returns always 0

- **Lock functions**
  - e.g., deadlocks due to wrong nesting of different locks are reported

```c
int main(void)
{
    double x0, y0; 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*/
    return 0;
}
```
Restrictions

- 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

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., race conditions, 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.
## Modes

- **Thread count independent analysis mode (TCI)**
  - Source Code Instrumentation
  - Compilation with
    - `icc / ifort -g -openmp -tcheck`

- **Simulation approach = thread-parallel execution (BIN)**
  - 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`

### Example 8 – Mixed mode compilation – Main program

caller.c

```c
13 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 }
```

### TCI and Binary Mode

- Overview
- Seven Examples
- Restrictions in TCI mode
- End of checking
- TCI and Binary Mode
- Wrap-up and Summary
- Practical
- Appendix
Example 8 – Mixed mode compilation

mylib.c

```c
#include <omp.h>

void scalar_mult_vector(float *b, float c, float *a, int n) {  /* B = c*A */
  for (int i = 0; i < n; i++) b[i] = c * a[i];
}

void scalar_plus_vector(float *b, float c, float *a, int n) {  /* B = c+A */
  for (int i = 0; i < n; i++) b[i] = c + a[i];
}
```

Example 8 – Mixed mode compilation

- Mixed compilation
  - icc -tcheck -openmp -g -c caller.c
  - icc -g -c mylib.c
  - icc -tcheck -openmp -g -o a.out caller.o mylib.o

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

- Call stack for each entry
  - tcheck_cl -s 1 threadchecker.thr
  - tcheck_cl -s 2 threadchecker.thr
  - ...

Thread Checker
Rolf Rabenseifner
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>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write, 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>
</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)

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

Binary instrumentation and parallel execution
- Compilation: only with Intel® compilers, as usual, but with -g option:
  icc -openmp -g -o my_prog my_prog.c
  ifort -openmp -g -o my_prog my_prog.f
- Parallel execution and analysis with tcheck_cl
  export OMP_NUM_THREADS=4
tcheck_cl -w 90 -o my_prog.txt ./my_prog
- Problems with Intel® 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

➤ But may help if TCI is not applicable
### 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

### Wrap-up

- Overview
- Seven Examples
- Restrictions in TCI mode
- End of checking
- TCI and Binary Mode
- Wrap-up and Summary
- Practical
- Appendix
History

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

- **Assure**
  - a KAP/pro tool
  - it was available for most shared memory platform
  - sold to Intel → basis for Intel® Thread Checker

- Paul Petersen, Sanjiv Shah:
  *OpenMP Support in the Intel® Thread Checker.*
  WOMPAT 2003, pp 1-12.
  http://sunsite.informatik.rwth-aachen.de/dblp/db/conf/wompat/wompat2003.html#PetersenS03

- **Intel® 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/ittc/
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.

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**.

Currently¹ 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
Verifying an OpenMP Parallelization with the Intel® Thread Checker

- Overview
- Seven Examples
- Restrictions in TCI mode
- End of checking
- TCI and Binary Mode
- Wrap-up and Summary
- Practical
- Appendix

**Intel® Thread Checker – Practical**

(on cacau.hww.de)

- module load itt/tcheck-3.0 (setup, only once)
- cd ~/OpenMP/#nr/pitfa/ll

- 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**

- Example 1: Two separate loops → 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)”

**Thread Checker**

Rolf Rabenseifner

[H] Höchstleistungsrechenzentrum Stuttgart
Verifying an OpenMP Parallelization with the Intel® Thread Checker

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
  http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&
  p=1&u=%2Fnetacgi%2FPTO%2FSRCHNUM.htm&f=G&l=50
  – This patent describes implementation and goals of the projection mode

- Utpal Banerjee, Brian Bliss, Zhiqiang Ma, Paul Petersen (Intel):
  Unraveling Data Race Detection in the Intel Thread Checker
  Proceedings, First Workshop on Software Tools for Multi-Core Systems (STMCS’06), March 26, 2006,
  http://www.isi.edu/~kintali/stmcs06/
  (at The 4th Annual International Symposium on Code Generation and Optimization (CGO)
  http://www.cgo.org/cgo2006/)
  – 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.
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