Comparing Intel Thread Checker and Sun Thread Analyzer

Christian Terboven
terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany
Agenda

• Introduction

• Simple example walkthrough
  • Intel Thread Checker
  • Sun Thread Analyzer

• Further comparison
  • C++
  • Runtime & Memory consumption
  • Other features

• Conclusion
Introduction

• The fundamental difference between MPI and OpenMP:

  • Shared-Memory (OpenMP):
    • Data resides in shared address spaces of all threads
      → Danger of data races

  • Distributed-Memory (MPI):
    • Data is (manually) distributed between all processes
      → Data has to be sent explicitly

• Virtually every multithreaded program we examined had at least one data race …
Data Race detection

- A data race occurs when all following conditions happen concurrently:
  - Two or more threads access the same memory location,
  - Between two synchronization points in an OpenMP program,
  - At least one thread modifies that location,
  - The accesses to the location are not protected, e.g. by locks.

- Principle design of a data race detection tool:
  - Instrument application
  - Trace memory references
  - Trace thread management operations
  - Trace synchronization operations
  - Compare event pairs (two threads), check for possible data race
History

• Assure for Threads was first commercial product
  • OpenMP
  • Available on many platforms

• 2000: Intel acquired KAI
  • Renamed the product to Intel Thread Checker
  • Available on Linux and Windows
  • On Intel-compatible architectures

• 2007: Sun Thread Analyzer
  • Available since Sun Studio 12
  • Available on Linux and Solaris
  • On Intel-compatible architectures and UltraSPARC architectures
Agenda

- Introduction

- Simple example walkthrough
  - Intel Thread Checker
  - Sun Thread Analyzer

- Further comparison
  - C++
  - Runtime & Memory consumption
  - Other features

- Conclusion
Example program

- C version of Jacobian solver from OpenMP website:

```c
#pragma omp parallel private(i)
{
    [...] /* compute stencil, residual and update */
#pragma omp for
    for (j=1; j<m-1; j++)
        for (i=1; i<n-1; i++){
            resid = (ax * (UOLD(j,i-1) + UOLD(j,i+1))
                    + ay * (UOLD(j-1,i) + UOLD(j+1,i))
                    + b * UOLD(j,i) - F(j,i) ) / b;
            U(j,i) = UOLD(j,i) - omega * resid;
            error = error + resid*resid;
        }
} /* end of parallel region */
```

We deliberately introduced two parallelization mistakes related to the variables resid and error.
Expectations

- Correction:
  - Declare variable resid private
  - Declare variable error as reduction

- Why declaring error private would not be correct:
  - There would not be a data race! But …
  - \( error = error + resid \times resid; \)
  - Contributions from all threads (\( resid \)) are accumulated
  - It is used in the sequential part later on \( \rightarrow \) reduction

- Expectations (for the Jacobian solver):
  - Minimal: report data races in variables resid and error
  - Provide guides how to resolve the race conditions
  - Optimal: Propose to declare error as reduction
Intel Thread Checker

![Thread Checker Screenshot]

ParCo’07  C. Terboven

Center for Computing and Communication
Intel Thread Checker

- Analysis results with binary instrumentation:
  - Allows checking of existing binary code (debug info helpful)
  - Program has to be executed with at least two threads
  - In total 10 errors for 3 different program locations
    - Unsynchronized write/write and read/write access to resid
    - Unsynchronized read from resid in write to U[]
    - Unsynchronized write/write and read/write access to error
    - Unsynchronized read from resid in write to error

- Together with the call stacks a correction proposal is given:
  - Protect access to variable resid/error by using either locks or critical regions
  - Make variable resid/error private by using either thread-local storage or private clauses
  → This is not correct in the case of error!
Intel Thread Checker

- Analysis results with source instrumentation:
  - Compilation with Intel Compilers required
  - Additional analysis capabilities for OpenMP programs – if program flow does not depend on the thread id
  - In total only 5 errors for 2 different program locations
  - The variable names `error` and `resid` are given

- The following correction proposal is given:
  - Protect access to variable `resid` by using either locks or critical regions
  - Make variable `error` private by using either thread-local storage or private clause
  - Consider declaring variable `error` as reduction
    → Declaring `error` as reduction is the optimal resolution!
Sun Thread Analyzer

Total Races: 2

Race #1, Vaddr: 0x80456c4
Access 1: Write, jacobi -- MP doall from line 74 [$_dlB74.jacobi] + 0x00000753,
line 82 in "jacobi_omp_error_1.c"
Access 2: Write, jacobi -- MP doall from line 74 [$_dlB74.jacobi] + 0x00000753,
line 82 in "jacobi_omp_error_1.c"
Total Traces: 1

Race #2, Vaddr: 0x80456bc
Access 1: Write, jacobi -- MP doall from line 74 [$_dlB74.jacobi] + 0x0000081B,
line 82 in "jacobi_omp_error_1.c"
Access 2: Write, jacobi -- MP doall from line 74 [$_dlB74.jacobi] + 0x0000081B,
line 82 in "jacobi_omp_error_1.c"
Total Traces: 1
**Sun Thread Analyzer**

- Analysis results:
  - In total 6 errors for 2 different program locations
    - A data race with read and write to variable `resid` is reported
    - A data race with read and write to variable `error` is reported

- Together with the call stacks a resolution proposal is given:
  - Protect access to variable `resid/error` by either using locks or critical regions
    → This is not correct in the case of `error`!
Guidance in the parallelization process

- In OpenMP the default is shared
- Finding all variables that have to be made private is
  - A lot of work
  - Error-prone

- Use your data race detection tool
  - Identify performance-critical hotspots
  - Insert e.g. OpenMP pragmas
  - Run the analysis with suited datasets
    - Use code coverage tool
  - Extract the list of variables with races
    - Most probably have to be made private / firstprivate / lastprivate
    - Thread Checker even proposes reduction variables
Agenda

• Introduction

• Simple example walkthrough
  • Intel Thread Checker
  • Sun Thread Analyzer

• Further comparison
  • C++
  • Runtime & Memory consumption
  • Other features

• Conclusion
Handling of C++ programs

• We tested a CG solver with external parallelization

```c
#pragma omp parallel firstprivate(iter, [...])
{
    while (iter < max_iter && sqrt(sigma) > tol)
    {
        [...];
        q = s + beta * q;
        [...]
    }
} // end omp parallel
```

• The `operator*` member function contains orphaned OpenMP worksharing constructs

• Good news: The data races are reported where they occur!
• Not so good news: Additional races e.g. in the STL are reported
Runtime and Memory Consumption

• Advice is to use the smallest and still meaningful dataset

<table>
<thead>
<tr>
<th>Program</th>
<th>Jacobi Mem</th>
<th>Jacobi MFLOP/s</th>
<th>SMXV Mem</th>
<th>SMXV MFLOP/s</th>
<th>AIC Mem</th>
<th>AIC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original, Intel with 2 threads</td>
<td>5 MB</td>
<td>621</td>
<td>40 MB</td>
<td>929</td>
<td>4 MB</td>
<td>5.0 sec</td>
</tr>
<tr>
<td>Intel Thread Checker binary instr., 2thr.</td>
<td>115 MB</td>
<td>0.9</td>
<td>1832 MB</td>
<td>3.5</td>
<td>30 MB</td>
<td>9.5 sec</td>
</tr>
<tr>
<td>Intel Thread Checker source instr.</td>
<td>115 MB</td>
<td>3.1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Original, Sun with 2 threads</td>
<td>5 MB</td>
<td>600</td>
<td>50 MB</td>
<td>550</td>
<td>2 MB</td>
<td>8.4 sec</td>
</tr>
<tr>
<td>Sun Thread Analyzer with 2 threads</td>
<td>125 MB</td>
<td>1.1</td>
<td>2020 MB</td>
<td>0.8</td>
<td>17 MB</td>
<td>8.5 sec</td>
</tr>
</tbody>
</table>

• Decrease grid resolution, limit the number of iterations, simulate just a few time steps, …
• Nevertheless: Typical production datasets are impossible to analyze!
• The Sun tool still provides some scalability
Other features

• Re-using components (libraries) is good software engineering practice – but are these thread safe?
  • Bad performance advices from the past.

• Both tools provide deadlock detection capabilities:
  • Inappropriate use of mutex locks in Posix-Threads programs
  • Not an issue for OpenMP programs only using constructs
  • Can be enabled without data race detection capabilities, thus only little overhead is introduced

• If explicit memory flushes are used for implementing locks, no tool recognizes that
  • False positives are reported
  • Our advice: Do not use flushes for synchronization!
Agenda

• Introduction

• Simple example walkthrough
  • Intel Thread Checker
  • Sun Thread Analyzer

• Further comparison
  • C++
  • Runtime & Memory consumption
  • Other features

• Conclusion
Conclusion

• We recommend: Never put a multithreaded program into production before using one of these tools!
  • Both tools are capable of detecting data races in complex applications.

• Source instrumentation of Intel Thread Checker is advantageous for OpenMP programs – if applicable.
• Sun Thread Analyzer still offers scalability in analysis mode.

• Increased memory consumption may render both tools unusable.
End

Thank you for your attention.

Questions?