HPC Tools Portfolio:
Tuning and Shared-Memory Parallelization on Windows

Christian Terboven
terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University

2nd Windows-HPC User Group Meeting
March 31st, TU Dresden, Germany
Agenda

- Methodology and Motivation
- Profiling Tools
- Parallelization Tools
- Debugging and Correctness Tools
- Summary
Motivation for this Tool Inventory

○ Microsoft is pushing Windows HPC Server
  – Opportunity: Integration
    • Operating System + Development Environment + Compiler + Debugger (+ Tools) are all from Microsoft
  – Risk: Windows-HPC is the New Kid on the Block
    • Microsoft might not focus support on legacy (HPC) code

○ Interest in Windows-HPC as development platform is high
  – Example from last week’s PPCES at Aachen: About 40 percent of the course participants indicated interest in trying the lab exercises on our Windows-HPC platform
    • Switchers? People just interested in the other side? People new to HPC / parallel programming? Is the IDE approach appealing?

○ What has Windows-HPC to offer *today*?
Methodology: Tuning and Parallelization

- Process of (incremental) parallelization with OpenMP:
  1. Runtime analysis of a given program
  2. Identification of so-called *Hotspots*, that are compute intensive parts of a program → Think about serial tuning
     1. If serial tuning was applicable → Restart at (1)
  3. Parallelization of identified *Hotspots*
  4. Restart at (1), but switch to parallel runtime analysis

- Important tools in this process:
  - Profiler for serial and parallel program
  - Debugger
  - If applicable: tools for correctness checking
Application used in this Case Study: DROPS

- Numerical Simulation of two-phase flow
- (Adaptive) Tetrahedral Grid Hierarchy
- Finite Element Method (FEM)

- Written in C++: is object-oriented, uses nested templates, uses STL types, uses compile-time polymorphism, ...

- Typical Setup:
  - Visual Studio Release Configuration, 64bit
  - Tiny dataset: 77.5 seconds per run
  - Benchmark kernels for identified Hotspots

Example:
Silicon oil drop in D_2O (fluid/fluid)
Tuning and Shared-Memory Parallelization Tools on Windows

31.03.2009 – C. Terboven

Agenda

- Methodology and Motivation
- Profiling Tools
- Parallelization Tools
- Debugging and Correctness Tools
- Summary
Profiler: Requirements

- Goals of this analysis step:
  - Get an overview of the program’s call tree
  - Get an overview of how much compute time is spent in which parts of the program
    - Inclusive time per function
    - Exclusive time per function
  \(\rightarrow\) Derive program’s critical path with respect to performance

- Comparison of three tools:
  - Visual Studio 2008 Profiler
  - Intel Parallel Studio: Amplifier
  - Intel VTune
### Visual Studio 2008 Profiler (1/2)

- **Execution time**: 81.7 seconds over 77.5 seconds without tool.

- **Functions**: Inclusive time versus Exclusive time, per function.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Inclusive Samples</th>
<th>Exclusive Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROPS::y_Ax</td>
<td>8.398</td>
<td>8.398</td>
</tr>
<tr>
<td>DROPS::y_ATx</td>
<td>5.950</td>
<td>5.950</td>
</tr>
<tr>
<td>[ntdll.dll]</td>
<td>22.716</td>
<td>1.642</td>
</tr>
<tr>
<td>std::valarray::Grow</td>
<td>2.084</td>
<td>1.312</td>
</tr>
<tr>
<td>DROPS::operator*&lt;class DROPS::SparseMatBaseCL&lt;double&gt;,class DROPS::SparseMatBaseCL&lt;&gt;,class DROPS::SparseMatBaseCL&lt;&gt;()&gt;</td>
<td>12.773</td>
<td>474</td>
</tr>
<tr>
<td>DROPS::InstantNavierStokes2PhaseP2P1CL&lt;class DROPS::ZeroFlowCL&gt;::SetupNonlinear_P2(cdouble)</td>
<td>2.509</td>
<td>393</td>
</tr>
<tr>
<td>DROPS::dot&lt;class DROPS::GridFunction&lt;classe DROPS::SSVectorCL&lt;3&gt; &gt; const &amp;class DROPS::SparseMatBuilderCL&lt;double&gt;::operator()(unsigned _int64, unsigned _int64)</td>
<td>759</td>
<td>383</td>
</tr>
<tr>
<td>std::operator&lt;double&gt;(double const &amp;class std::valarray&lt;double&gt; const &amp;class std::valarray&lt;double&gt;::Build(void)</td>
<td>370</td>
<td>245</td>
</tr>
<tr>
<td>[MSVCR90.dll]</td>
<td>615</td>
<td>230</td>
</tr>
</tbody>
</table>

→ Provides a very good overview of where the time is spent.
Caller / Callee: Call tree browsing, metrics per call site

- Allows for very good examination of the (dynamic) program structure and making of tuning and/or parallelization decisions.

Missing:
- Simple identification of Hotspots
- Time spent per code line
### Intel Parallel Studio: Amplifier (1/2)

- **Execution time:** 87.6 seconds over 77.5 seconds **wo/ tool**
- **Available from within VS:**
- **Hotspot-based display of analysis result:**
  - Easily accessible.
  - Leads user focus right towards Hotspots.
Examination of analysis result down to source line level:

```c
897 // y = A*x
898 // fails, if num_rows==0.
899 // Assumes, that none of the arrays involved do alias.
900 template <typename T>
901 inline void
902 y_Ax(T* __restrict y,
903 size_t num_rows,
904 const T* __restrict Aval,
905 const size_t* __restrict Arow,
906 const size_t* __restrict Acol,
907 const T* __restrict x)
908 {
909   T sum;
910   size_t rowend;
911   size_t nz= 0;
912   do {
913     rowend = *++Arow;
914     sum  = T();
915     for (; nz<rowend; ++nz)
916       sum+= (*(Aval++))*x[*(Acol++)];
917     (*y++)= sum;
918   } while (num_rows > 0);
919 }
```

Results surprise users very often, thus this view should be examined before the first efforts towards parallelization are carried out.

Missing:
- Hardware Counter measurements
- Better representation of call graph
Execution time:
- **Sampling:** Failed for this application on our systems
- Call Graph Profiling: 77.6 Seconds over 77.5 seconds wo/ tool

Weak function profile, but call graph view:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LocNorm_sq</td>
<td>DROPS::ExchangeCL</td>
<td>9.327</td>
<td>176.337</td>
<td>177.574</td>
<td>2</td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>1</td>
<td>2.092</td>
<td>76.613,213</td>
<td>1</td>
</tr>
<tr>
<td>mainCRTStartup</td>
<td></td>
<td>1</td>
<td>0</td>
<td>76.613,811</td>
<td>1</td>
</tr>
<tr>
<td>MakeInitialTriang</td>
<td>double __cdecl(DROPS::S...</td>
<td>DROPS::AdapTriangCL</td>
<td>1</td>
<td>159.924</td>
<td>1</td>
</tr>
<tr>
<td>Map</td>
<td>DROPS::AffineSquareCL</td>
<td>2.754</td>
<td>19</td>
<td>19</td>
<td>1</td>
</tr>
<tr>
<td>map&lt;...</td>
<td>DROPS::MLSparseMatBas...</td>
<td>342.387</td>
<td>987</td>
<td>22.242</td>
<td>1</td>
</tr>
<tr>
<td>MatDescBaseCL&lt;...</td>
<td>DROPS::MatDescBaseCL&lt;...</td>
<td>10</td>
<td>4</td>
<td>20</td>
<td>4</td>
</tr>
</tbody>
</table>

→ Displays the Critical Path, but only few details.
Agenda

- Methodology and Motivation
- Profiling Tools
- Parallelization Tools
- Debugging and Correctness Tools
- Summary
Parallel Profiling: Requirements

○ Goals of this analysis:
  – Compare how the performance profile has changed to the serial program by the current parallelization
  – Evaluate the scalability and efficiency of the parallelization
    • How much overhead has been introduced?
    • How good is the work distributed to the threads (load balance)?
→ Understand how far you still have to go ...

○ Comparison of four tools:
  – Visual Studio 2008 Profiler
  – Intel Parallel Studio: Inspector
  – Intel Thread Profiler
  – Intel VTune
Test: Visual Studio 2008 Profiler (1/1)

- No explicit support for threads or OpenMP:
  
  **Functions Causing Most Work**

<table>
<thead>
<tr>
<th>Name</th>
<th>Samples</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>[kernel32.dll]</td>
<td>17.367</td>
<td>100.00</td>
</tr>
<tr>
<td>[ntdll.dll]</td>
<td>17.3</td>
<td></td>
</tr>
<tr>
<td>_vcomp::fork_helper</td>
<td>17.3</td>
<td></td>
</tr>
<tr>
<td>_vcomp::ParallelRegion::HandlerThreadFunc(void *,unsigned long)</td>
<td>17.3</td>
<td></td>
</tr>
<tr>
<td>main$omp$1</td>
<td>17.23</td>
<td></td>
</tr>
</tbody>
</table>

  → Just calls to OpenMP runtime functions are presented.

- Comparison of two profiles (serial vs. parallel) is possible:

  → One can derive the improvements, but that is about it.

- Missing:
  - Support for threading
Simpe thread utilization presentation per function:

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>CPU Time by Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_times</td>
<td>smxv-benchmark.exe</td>
<td>0.125s</td>
</tr>
<tr>
<td>y_Ax_omp</td>
<td>smxv-benchmark.exe</td>
<td>0.337s</td>
</tr>
<tr>
<td>vcomp::fork_helper</td>
<td>smxv-benchmark.exe</td>
<td>0.077s</td>
</tr>
<tr>
<td>vcomp::ParallelRegion::HandlerThreadFunc</td>
<td>smxv-benchmark.exe</td>
<td>0.051s</td>
</tr>
<tr>
<td>main</td>
<td>smxv-benchmark.exe</td>
<td>0.031s</td>
</tr>
<tr>
<td>drv_init</td>
<td>smxv-benchmark.exe</td>
<td>1.425s</td>
</tr>
<tr>
<td>mainomp1</td>
<td>smxv-benchmark.exe</td>
<td>0.000s</td>
</tr>
</tbody>
</table>

- Only very basic information. More details with respect to load balancing (for example) are very much wanted.

Missing:
- Expert information
- More fine-grained details
Load balance statistics - whole program:

- A0

and per parallel region:

- A0T0
- A0T1
- A0T2
- A0T3

→ Allows for mapping of dataset load characteristics to execution profile:

→ Different views for non OpenMP programs are available as well.
Call graph profile for multiple threads:

→ In many cases of only little use.
Agenda

- Methodology and Motivation
- Profiling Tools
- Parallelization Tools
- Debugging and Correctness Tools
- Summary
Parallel Debugging and Checking: Requirements

- Goals of this analysis:
  - Traditional bug hunting, but with parallel programs
  - Verification that no errors have been introduced by parallelization
    - Check for serial equivalence?
  - Find and eliminate any kind of bug in a parallel program

- Comparison of four tools:
  - Visual Studio 2008 Debugger
  - Visual Studio 2008 Debugger with Allinea DDTlite
  - Intel Parallel Studio: Inspector (read: Intel Thread Checker)
  - Intel Parallel Studio: Composer
Visual Studio 2008 brings well-known debugging experience to multi-threaded programs:

- Individual control of threads:

```
x = 1;

while (|x <= maxit && error > tol) {
  error = 0.0;
  #pragma omp parallel private (i)
  {
    #pragma omp for
```

→ Very good C++ debugger with all required functions for multi-threaded debugging.

Missing:
- Better control of thread groups
- Laminated view of private variables
Individual thread grouping and switching:


- Selected Processes and Threads
  - All: 8 threads
  - Break when: Every selected thread reaches a breakpoint
  - Select all 9 threads in program

- Display of variable values per thread:

→ Significant productivity improvement.

Missing:
- Support for task-based paradigms
Intel Parallel Studio: Inspector (1/1)

- **Data Race**: the typical OpenMP programming error, when:
  - two or more threads of a *single process* access the same memory location concurrently (between two synchronization points), and at least one of these accesses modifies this location, and the accesses to this location are not protected by locks or critical regions.
  - Non-deterministic occurrence

- (OpenMP-specific) automated data race detection:

<table>
<thead>
<tr>
<th>Relation Sets</th>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Description</th>
<th>Count</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Read → Write data-race</td>
<td>X</td>
<td>Memory write at &quot;main.c&quot;:196 conflicts with a prior memory read at &quot;main.c&quot;:125 (anti...</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Write → Read data-race</td>
<td>X</td>
<td>Memory read at &quot;jacobi.c&quot;:61 conflicts with a prior memory write at &quot;jacobi.c&quot;:53 (flow dependence)</td>
<td>70</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Write → Read data-race</td>
<td>X</td>
<td>Memory read at &quot;jacobi.c&quot;:61 conflicts with a prior memory write at &quot;jacobi.c&quot;:52 (flow dependence)</td>
<td>70</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Write → Read data-race</td>
<td>X</td>
<td>Memory read at &quot;jacobi.c&quot;:61 conflicts with a prior memory write at &quot;jacobi.c&quot;:51 (flow dependence)</td>
<td>70</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Write → Read data-race</td>
<td>X</td>
<td>Memory read at &quot;jacobi.c&quot;:61 conflicts with a prior memory write at &quot;main.c&quot;:136 (flow dependence)</td>
<td>59</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Write → Write data-race</td>
<td>X</td>
<td>Memory write at &quot;jacobi.c&quot;:66 conflicts with a prior memory write at &quot;jacobi.c&quot;:69 (output race)</td>
<td>65</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Read → Write</td>
<td></td>
<td>Memory write at &quot;jacobi.c&quot;:66 conflicts with a prior memory write at &quot;main.c&quot;:196 (output race)</td>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

→ Easily accessible.

→ Our recommendation: Never put an OpenMP program in production without using this tool.
Provides new debugging views, especially for tasks:

```java
static void quick_sort(int lt, int rt, float *data) {
    if ((rt - lt) < LOW_LIMIT) {
        serial_quick_sort(lt, rt, data);
    } else {
        int md = partition(lt, rt, data);
        #pragma omp task
        quick_sort(lt, md-1, data);
        #pragma omp task
        quick_sort(md+1, rt, data);
    }
}
```

→ Significantly extends Visual Studio debugger capabilities. First OpenMP 3.0 debugger I know of.
Agenda

- Methodology and Motivation
- Profiling Tools
- Parallelization Tools
- Debugging and Correctness Tools
- Summary
## Summary (1/2)

<table>
<thead>
<tr>
<th>Task</th>
<th>Windows</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Microsoft</strong></td>
<td><strong>3rd Party</strong></td>
</tr>
<tr>
<td>Compiler</td>
<td>C/C++</td>
<td>Intel C/C++ &amp; F95</td>
</tr>
<tr>
<td>IDE</td>
<td>Visual Studio 2008</td>
<td>Eclipse</td>
</tr>
<tr>
<td></td>
<td><em>(Parallel) Profiling</em></td>
<td>Intel Parallel Studio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel VTune</td>
</tr>
<tr>
<td></td>
<td><em>(Parallel) Debugging</em></td>
<td>Intel Parallel Studio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allinea DDTlite</td>
</tr>
<tr>
<td>Parallelization</td>
<td></td>
<td>Intel Threading Tools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>but wo/ GUI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sun Studio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TotalView</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Allinea DDT (support for tasks)</td>
</tr>
</tbody>
</table>

- **Black** = well-suited tool for task presented, **Gray** = tool is not well-suited
Windows offers a very complete development environment
- Centered around Visual Studio 2008 → Integrated Dev. Env.
- Market place and opportunities for 3rd Parties
- Would look different without Intel’s software efforts though

HPC can also be read as High Productivity Computing
- IDE with good support for performance analysis and Shared-Memory parallelization is missing on other platforms
- Software should be considered an important ingredient for HPC

Software environment on Windows does not come for free
- Are you willing to pay for (good) software tools?!?

Microsoft is promising much improved support for multi-core parallelization with Visual Studio 2010 (see CTP or Beta1) ...
The End

Thank you for your attention!

WinHP3C:
http://www.rz.rwth-aachen.de/winhp3c