Performance Tools for Technical Computing

Christian Terboven
terboven@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University

Intel Software Conference 2010
April 13th, Barcelona, Spain
Agenda

- Motivation and Methodology
- Profiling Tools
- Parallel Profiling Tools
- Debugging and Correctness Tools
- Summary
My Starting Point (1/2)

- HPC for Engineers and Scientists:
  - Actual (Engineering / Research) work
  - Familiarizing with the code
  - Debugging
  - Parallelization
  - Scheduling
  - Tuning

→ Productivity becomes more and more of an issue!

- What to aim for?
  - Trivial Scalability? → Parametric Array Jobs
  - Parallel Scalability? → Parallelization with MPI, OpenMP, ...
  - Architecture-specific tuning? → cc-NUMA, Cache Access, ...
HPC systems in Aachen:
- Today: About 700 x86-based systems with two or more sockets, Intel Harpertown + Nehalem-EP, 2-3 GB of memory per core
- Tomorrow: About 210 TFLOPS with Intel Nehalem-EX in four to eight sockets, still about 2-3 GB of memory per core
- CentOS Linux (92% of the nodes) and Windows HPC Server (8%)

Compared to five years ago the usage scenario has changed
- January 2010:
  - Over 64k batch jobs, 10% of CPU time for undergraduate students
  - Heterogeneous job-mix: From large parallel to large job array
  - Applications consist of multiple languages, i.e. Matlab + Fortran
  → We hardly can force users to follow the traditional HPC path
(Incremental) Tuning and Parallelization process:

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$_2$O (fluid/fluid)
Agenda

- Motivation and Methodology
- Profiling Tools
- Parallel Profiling 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
  → Derive program’s critical path with respect to performance

- Comparison of the following tools:
  - Visual Studio 2008 / gprof
  - Intel Parallel Studio: Amplifier
  - Intel VTune
- Execution time: 81.7 seconds over 77.5 seconds without the 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&lt;...(double *,unsigned __int64, double const *,unsigned __int64 const *,...)</td>
<td>8.398</td>
<td>8.398</td>
</tr>
<tr>
<td>DROPS::y_ATx&lt;...(double *,unsigned __int64, double const *,unsigned __int64 const *,...)</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&lt;double&gt;::_Grow(unsigned __int64, double const *, unsigned __int64)</td>
<td>2.084</td>
<td>1.312</td>
</tr>
<tr>
<td>DROPS::operator*(class DROPS::SparseMatBaseCL&lt;...&gt;, class DROPS::SparseMatBaseCL&lt;...&gt;)</td>
<td>12.773</td>
<td>474</td>
</tr>
<tr>
<td>DROPS::InstatNavierStokes2PhaseP2P1CL&lt;...&gt;::SetupNonlinear_P2(c...)</td>
<td>2.509</td>
<td>393</td>
</tr>
<tr>
<td>DROPS::dot(class DROPS::GridFunction&lt;...&gt;::class DROPS::VectorCL&lt;3&gt; &gt; const &amp; class DROPS::SparseMatBuilderCL&lt;...&gt;::operator()(unsigned __int64, unsigned __int64)</td>
<td>759</td>
<td>383</td>
</tr>
<tr>
<td>std::operator&lt;&lt;(double const &amp; class std::valarray&lt;double&gt; const &amp;)</td>
<td>470</td>
<td>246</td>
</tr>
<tr>
<td>DROPS::SparseMatBuilderCL&lt;...&gt;::Build( void)</td>
<td>370</td>
<td>245</td>
</tr>
<tr>
<td>[MSVCR90.dll]</td>
<td>615</td>
<td>230</td>
</tr>
<tr>
<td>std::Tree&lt;class std::_Tmap_traits&lt;...&gt;, struct std::less&lt;...&gt;, Allocat...</td>
<td>184</td>
<td>184</td>
</tr>
<tr>
<td>DROPS::ExchangeCL::AccurLocDotNoAcc&lt;...&gt;::class DROPS::VectorBaseCL&lt;...&gt; const &amp; class I</td>
<td>175</td>
<td>175</td>
</tr>
<tr>
<td>DROPS::ParModGMRES&lt;...&gt;::class DROPS::MLSparseMatBaseCL&lt;...&gt;::class DROPS::VectorBase&lt;...&gt;</td>
<td>3.485</td>
<td>157</td>
</tr>
<tr>
<td>DROPS::Quad5CL&lt;...&gt;::quadp&lt;...&gt;::const</td>
<td>213</td>
<td>154</td>
</tr>
<tr>
<td>DROPS::LevelsetP2CL&lt;...&gt;::SetupSystem&lt;...&gt;::class DROPS::P2EvalCL&lt;...&gt;::class DROPS::SVectorCL&lt;3&gt; &gt; cla...</td>
<td>904</td>
<td>147</td>
</tr>
<tr>
<td>std::valarray&lt;double&gt;::operator=( class std::valarray&lt;double&gt; const &amp;)</td>
<td>141</td>
<td>141</td>
</tr>
<tr>
<td>DROPS::FE_P2CL::val&lt;...&gt;::class DROPS::LocalP2CL&lt;...&gt;::operator=(class DROPS::LocalP2CL&lt;...&gt;::*...)</td>
<td>13.13</td>
<td>45</td>
</tr>
<tr>
<td>std::Tree&lt;class std::_Tmap_traits&lt;...&gt;,class DROPS::FaceCL&lt;...&gt;::struct std::less&lt;...&gt;, Allocat...</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>DROPS::ParAccurPCG&lt;...&gt;::class DROPS::CompositeMatrixBaseCL&lt;...&gt;::class DROPS::SparseMatBaseCL&lt;...&gt;</td>
<td>13.13</td>
<td>45</td>
</tr>
<tr>
<td>DROPS::SetupSystem1&lt;...&gt;::class DROPS::ZeroFlowCL&lt;...&gt;::class DROPS::MultiGridCL const &amp; class DROPS::SetupSystem2&lt;...&gt;::P2P1X&lt;...&gt;::class DROPS::ZeroFlowCL&lt;...&gt;::class DROPS::MultiGridCL const &amp; class DROPS::ZeroFlowCL&lt;...&gt;::class DROPS::MultiGridCL const &amp;</td>
<td>1.25</td>
<td>5.25</td>
</tr>
</tbody>
</table>

→ Provides a 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 tuning and/or parallelization decisions.

 Missing:
- Simple identification of Hotspots
- Good mapping to the source code
- Support for parallelism
- Execution time: 87.6 seconds over 77.5 seconds without tool
- Available from within VS:
- Hotspot-based display of analysis result:

$$\text{→ Easily accessible.}$$

$$\text{→ Leads user focus right towards Hotspots.}$$
Examination of analysis result at the level of source lines:

```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
The big question: When to stop?

- There is no single correct answer for this question.

- Today’s programming languages abstract from the hardware.
  - You can just compile and run your program, and be lucky (or not)
  - You can take a closer look:
    - How many floating point operations does your compute kernel perform per CPU cycle? Your Nehalem-EX can do up to four of them per core per cycle!
    - How much memory bandwidth does your compute kernel consume? Your Nehalem-EX delivers about 10 GB/s for a single thread.
  - And from here you can dive even deeper...
Intel VTune

- **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:**

  ![Call Graph Profiling Image]

  - Displays the Critical Path, and can perform hardware counter measurements.
Agenda

- Motivation and Methodology
- Profiling Tools
- Parallel Profiling 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 the following tools:
  – Intel Parallel Studio: Inspector
  – Intel Thread Profiler
  – Intel Vtune
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_time</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>y_Ax_serial</td>
<td>smxv-benchmark.exe</td>
<td>0.077s</td>
</tr>
<tr>
<td>drv_init</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_initomp1</td>
<td>smxv-benchmark.exe</td>
<td>1.425s</td>
</tr>
<tr>
<td>_CxxSetUnhandledExceptionFilter</td>
<td>smxv-benchmark.exe</td>
<td>0.000s</td>
</tr>
</tbody>
</table>

→ First view: basic information.
Motivation

Profiling

Par. Profiling

Debugging

Summary

Performance tools for technical computing 13.04.2010 – C. Terboven

Intel Parallel Studio: Amplifier (2/3)

→ This view: CPU utilization by thread. Helps us further, but it is still a global view.

→ This view: CPU utilization by OpenMP region.

→ Performance tools should be aware of the paradigm!
The explanation: If you parallelize a sparse matrix-vector multiplication in the naive way, that is by distributing the rows to threads, you may end up in a load balance due to the structure of the matrix.

Missing:
- Expert information, i.e. the ability to add hardware counter measurements

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

In most cases of only little use for parallel algorithms.

For example: Hardware counter measurements should be put into a context – I am not interested in the sheer number of cache misses, but whether my program is doing good or bad, i.e. how much memory bw my program is consuming per thread.
Again: How far to you want to go?

- There is still no single correct answer for this questions.
- Today's parallelization paradigms abstract from the hardware
  - You can just compile and run your program, and be lucky (or not)
  - You can take a closer look:
    - Is your program aware of the memory hierarchy? That means does it care for data-to-thread affinity on cc-NUMA architectures?
    - Could it be that two (or more) threads are writing to the same cache line, thus causing False Sharing?

It is often said that Shared-Memory parallelization is simpler than Message-Passing, but it wouldn't scale. That is not true, there is just more to it than loop-parallelization...
Agenda

- Motivation and Methodology
- Profiling Tools
- Parallel Profiling 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 the following 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 2010 brings well-known debugging experience to multi-threaded programs:

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

- Individual control of threads:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Location</th>
<th>Priority</th>
<th>Suspend</th>
</tr>
</thead>
<tbody>
<tr>
<td>7324</td>
<td>jacobi</td>
<td>jacobiomp$1</td>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>5424</td>
<td>jacobi</td>
<td>jacobiomp$1</td>
<td>Normal</td>
<td>0</td>
</tr>
<tr>
<td>5740</td>
<td>jacobi</td>
<td>jacobiomp$1</td>
<td>Normal</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Missing:
- Better control of thread or process groups
- Laminated view of private variables
Visual Studio 2008 Debugger w/ DDTlite

- Individual thread grouping and switching:

- Display of variable values per thread:

→ Significant productivity improvement.

Missing:
- Support for task-based paradigms
**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><strong>Read -&gt; Write data-race</strong></td>
<td>![X]</td>
<td><strong>Memory write at &quot;main.c&quot;:196 conflicts with a prior memory read at &quot;main.c&quot;:125</strong> (anti-...</td>
<td>1</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Write -&gt; 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 -&gt; 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 -&gt; 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 -&gt; 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;:56 (flow dependence)</td>
<td>70</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Write -&gt; 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>56</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>Read -&gt; Write</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>56</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:

- Significantly extends Visual Studio debugger capabilities. First OpenMP 3.0 debugger I know of.
Performance Tools for Technical Computing

Agenda

- Motivation and Methodology
- Profiling Tools
- Parallel Profiling Tools
- Debugging and Correctness Tools
- Summary
Performance Analysis and Tools require still a lot of Background Knowledge as well as Effort in use
  – But: If you have to deal with parallelization you better forget about printf-debugging right away!

When HPC on x86 went big with Opteron + Linux, there were hardly any tools for Shared-Memory parallelization
  – Luckily this has changed significantly, with more to come in the future (i.e. Parallel Studio for Linux, Visual Studio 2010, ...),
  – But hardware is still evolving at a fast pace: GPGPUs ...

HPC can also be read as High Productivity Computing
  – Software should be considered an important ingredient for HPC
  – Are you willing to pay for (good) software tools?!?
The End

Thank you for your attention!