Towards a Performance Engineering Workflow for OpenMP 4.0

TU Darmstadt:
Christian M. Iwainsky
Prof. Christian H. Bischof

RWTH Aachen University
Christian Terboven
Dirk Schmidl
Prof. Matthias S. Müller
Motivation

- Multi-Core Systems Common
- OpenMP Attractive Solution
  - Incremental parallelization
  - No complete rewrite of the code
  - Easy benefit from multi-core systems
- Parallel Programming Expertise Cumbersome to Obtain
- Need: Step by Step Guide-Lines for Good OpenMP Performance
  - Distilled our OpenMP course experience
- Test Approach on a CG-kernel
Agenda

1. Motivation
2. OpenMP Course Contents
3. CG-Solver Kernel
4. Workflow
5. Verification
6. Conclusion
OpenMP Workshop Contents

1) Basic Computer Architecture (½ Day)
   E.g.: Memory Hierarchy, NUMA, OS-Features for NUMA Support, Typical Performance Problems

2) OpenMP Language Tutorial (3/4 Day)
   E.g.: Basic Concept of Parallel Regions, Worksharing, Scoping, Memory Model, Synchronization, Tasking

3) Optimization and Tuning Aspects of OpenMP Programs (3/4 Day)
   E.g.: Thread Binding, NUMA Memory Allocation, Dynamic & Custom Scheduling, etc.

4) Basic Introduction to Performance Analysis (1 Day)
   E.g.: Hotspot Identification, Analysis of Thread Utilization
The CG Algorithm for Sparse Matrices

\[ r_0 = b - Ax_0 \]
\[ d_0 = r_0 \]
\[ \text{for } (k = 0 \ldots k_{\text{max}}) \]
\[ z = Ad^k \]
\[ \alpha_k = \frac{r_k^T - r_k}{d_k^T z} \]
\[ x_{k+1} = x_k - \alpha_k d_k \]
\[ \beta_k = \frac{r_{k+1}^T - r_{k+1}}{r_k^T r} \]
\[ d_{k+1} = r_{k+1} + \beta_k d_k \]
\[ \text{until } \|r_{k+1}\| < \text{tol} \]

1 Sparse Matrix Vector (SMXV)
2 Vector Operations:
   - Norm
   - Dot-Product
1 Memory Copy

1 SMXV
5 Vector Operations:
   - Dot-Product
   - Scaled-Vector-Addition

\begin{verbatim}
for(i=0; i<n; i++)
{
    y[i]=0;
    for(j=ptr[i]; j<ptr[i+1]; j++)
    {
        y[i]+=value[j]*x[index[j]];
    }
}
\end{verbatim}
First Draft for Performance Engineering

Workflow

Classic OpenMP Workflow

1) Naive Parallelization
2) Extend Parallel Regions
3) Add NUMA Awareness
4) Implement Load Balancing

Use Performance Analysis Tools
Naive Parallelization

- Create Profile
- Parallelize all Hotspots (e.g. Loopnests) Separately
- Enable Thread Binding
- Target System: 16-Socket Nehalem-EX System with BCS

Graph showing runtime and speedup with and without thread binding for different numbers of threads.
Extended Parallel Regions

- Extend all Parallel Regions to create a Single Parallel Region

- Overhead of Open Constructs

<table>
<thead>
<tr>
<th>OpenMP Construct</th>
<th>Overhead / Construct @128 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel For</td>
<td>660 µs</td>
</tr>
<tr>
<td>For</td>
<td>360 µs</td>
</tr>
</tbody>
</table>

Naive Implementation: 6 Parallel For Constructs - > only 1500 µs Saving per Iteration
NUMA Awareness

- Performance Measurement with PAPI Counters (4 Socket Setup)

<table>
<thead>
<tr>
<th>Counter</th>
<th>Memory Access Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM_UNCORE_RETIRED:LOCAL_DRAM</td>
<td>25.65%</td>
</tr>
<tr>
<td>MEM_UNCORE_RETIRED:REMOTE_DRAM</td>
<td>74.35%</td>
</tr>
</tbody>
</table>

- Data Locality Issue

- Initialize all data structures on the NUMA node used by its thread
NUMA Awareness: Result

The graph shows the comparison of runtime and speedup for different numbers of threads. The x-axis represents the number of threads, ranging from 1 to 128, and the y-axis represents the runtime in seconds. The graph includes three lines:

- Orange line: Runtime
- Blue line: Runtime - Naive
- Brown line: Speedup

As the number of threads increases, the runtime decreases, indicating better performance. The speedup line shows the improvement in runtime compared to the naive approach, with significant gains as the number of threads increases.
Load Balancing

- Performance Measurement (Score-P Profile)

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>OpenMP Waittime @ Barrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sparse Matrix Vector Multiplication</td>
<td>196 s</td>
</tr>
<tr>
<td>Vector Addition</td>
<td>2 s</td>
</tr>
</tbody>
</table>

Load Balancing Issue in SMXV

- Simple solution by using dynamic loop scheduling
Load Balancing and NUMA revisited

- Performance Measurement showed increase in remote memory access: 70% remote accesses
  NUMA Aware Memory Initialization fails to match processing by threads
- Solution: Manual, precalculated thread-distribution for SMXV

![Graph showing runtime and speedup for different number of threads]
First Draft for Performance Engineering Workflow

Pure OpenMP Workflow

1) Naive Parallelization
2) Extend Parallel Regions
3) Add NUMA Awareness
4) Implement Load Balancing

Use Performance Analysis Tools

Coprocessor Workflow

a) Naive Coprocessor Parallelization
b) Minimize Data Transfer
c) Extend Parallel Regions
d) Add NUMA Awareness
e) Extend Target Region

Use Performance Analysis Tools
Coprocessor Approach

- **OpenMP 4.0 Target Device Regions**
- **Naive Implementation**
  Individual Hotspots as target-regions
- **Target System: Intel Xeon Phi (Xeon Phi 5110P)**
  Serial version very slow in comparison to the host CPU
Minimizing Data Transfer

- **Accelerator Transfer Investigation** \( (\text{OFFLOAD\_REPORT} = 2) \)

SMXV:
- 3.2 GB Transfer / SMXV Call
- 3.3 s CPU-Time
- 0.5 s Phi-Time

- **Expand OpenMP Region \( \rightarrow \) Create Encompassing Target Data Region**

![Graph](image-url)
Extended Target Region & Tuned Code

- Same Optimization as with the CPU Version
- Facilitated whole CG-Kernel on the Phi-Coprocessor

![Graph showing speedup and runtime for different thread numbers. The x-axis represents the number of threads, ranging from 1 to 240. The y-axis represents runtime in seconds, ranging from 0 to 8000. The blue line represents speedup, peaking around 60.](image)
Result Verification

- **Performance Model for an OpenMP CG Implementation**
  - Input arguments:
    - Number of Non-Zeros, Vector-Length, Number of Iterations, Number of Threads
  - SMXV: 14 Bytes / FP Operation
  - Vector-operations: 8 Bytes / FP Operation

  Roofline: Memory Sub-System
  - Assume streaming performance (CPU + Memory System)
    - Measurement of $T_{FLOP}$; $T_{Byte}$
  - 2 Synchronization points (OpenMP reduction) necessary
    - Measurement of $T_{Red.Overhead(nT)}$
Model vs Best Effort Implementation

Stream Assumption for SMXV

- Predicted Runtime
- Measured Runtime
- Predicted Speedup
- Measured Speedup

Number of Threads:
- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128

Runtime:
- 0,00
- 200,00
- 400,00
- 600,00
- 800,00
- 1000,00
- 1200,00
- 1400,00

Speedup:
- 1
- 2
- 4
- 8
- 16
- 32
- 64
- 128
Conclusion

• Outlined Course Contents Necessary for OpenMP Implementation
• Workflow Suitable for CG Kernel
• Workflow also Applies to OpenMP 4.0 Accelerator Target-Style
• Performance Modeling can Indicate when to Stop

Questions?
Execution + Data Model

- Data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released
- Use target construct to
  - Transfer control from the host to the device
  - Establish a data environment (if not yet done)
  - Host thread waits until offloaded region completed

```c
#pragma omp target
map(alloc::...) \map(to::...) \map(from::...) 
{ ... }
```
Example: Execution and Data Model

- Environment Variable OMP_DEFAULT_DEVICE=<int>: sets the device number to use in target constructs

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

- map variable B to device, then execute parallel region on device, works probably pretty well on Intel Xeon Phi

```c
double B[N] = ...; // some initialization
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
    #pragma omp parallel for
    for (b = i; b < i+num_blocks; b++)
        B[b] += sin(B[b]);
```

- same as above, but code probably better optimized for NVIDIA GPGPUs
Comparing OpenMP with OpenACC

- **OpenMP 4.0 – for Intel Xeon Phi:**
  
  ```cpp
  #pragma omp target device(0) map(tofrom:B)
  #pragma omp parallel for 
  for (i=0; i<N; i++)
      B[i] += sin(B[i]);
  ```

- **OpenMP 4.0 – for NVIDIA GPGPU:**
  
  ```cpp
  #pragma omp target device(0) map(tofrom:B)
  #pragma omp teams num_teams(num_blocks) num_threads(bsize)
  #pragma omp distribute
  for (i=0; i<N; i += num_blocks)
      #pragma omp parallel for
      for (b = i; b < i+num_blocks; b++)
          B[b] += sin(B[b]);
  ```

- **OpenACC – for NVIDIA GPGPU:**
  
  ```cpp
  #pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\vector_length(bsize)
  #pragma acc loop gang vector
  for (i=0; i<N; ++i) {
      B[i] += sin(B[i]);
  }
  ```
Comparing OpenMP with OpenACC

- **OpenMP 4.0 – for Intel Xeon Phi:**
  ```c
  #pragma omp target device(0) map(tofrom:B)
  #pragma omp parallel for
  for (i=0; i<N; i++)
      B[i] += sin(B[i]);
  ```

- **OpenMP 4.0 – for NVIDIA GPGPU:**
  ```c
  #pragma omp target device(0) map(tofrom:B)
  #pragma omp parallel for
  for (i=0; i<N; i++)
      B[i] += sin(B[i]);
  ```

- **OpenACC – for NVIDIA GPGPU:**
  ```c
  #pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\vector_length(bsize)
  #pragma acc loop gang vector
  for (i=0; i<N; i++)
      B[i] += sin(B[i]);
  ```

   **Changed to RC2:**

   Combined directive
   ```c
   #pragma omp teams distribute parallel for
   ```