OpenMP and Performance

Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
schmidl@itc.rwth-aachen.de
Performance Tuning aims to improve the runtime of an existing application.

Tuning Cycle

- Collecting
- Analyzing
- Optimizing
- Testing
Hotspots

- A Hotspot is a source code region where a significant part of the runtime is spent.

90/10 law

90% of the runtime in a program is spent in 10% of the code.

- Hotspots can indicate where to start with serial optimization or shared memory parallelization.
- Use a tool to identify hotspots. In many cases the results are surprising.
Performance Tools
VTune Amplifier XE

- **Performance Analyses for**
  - Serial Applications
  - Shared Memory Parallel Applications

- **Sampling Based measurements**

- **Features:**
  - Hot Spot Analysis
  - Concurrency Analysis
  - Wait
  - Hardware Performance Counter Support
Stream

- Standard Benchmark to measure memory performance.
- Version is parallelized with OpenMP.

Measures Memory bandwidth for:

\[
\begin{align*}
    & y=x \text{ (copy)} \\
    & y=s\times x \text{ (scale)} \\
    & y=x+z \text{ (add)} \\
    & y=x+s\times z \text{ (triad)}
\end{align*}
\]

for double vectors \( x, y, z \) and scalar double value \( s \)

---

```c
#pragma omp parallel for
for (j=0; j<N; j++)
    b[j] = scalar*c[j];
```
Amplifier XE – Measurement Runs

1. Basic Analysis Types
2. Hardware Counter Analysis Types, choose Nehalem Architecture, on cluster-linux-tuning.
3. Analysis for Intel Xeon Phi coprocessors, choose this for OpenMP target programs.
Amplifier XE – Hotspot Analysis

Double clicking on a function opens source code view.

1. Source Code View (only if compiled with -g)
2. Hotspot: Add Operation of Stream
3. Metrics View
Load Balancing
Load imbalance

- **Load imbalance occurs in a parallel program**
  - when multiple threads synchronize at global synchronization points
  - and these threads need a different amount of time to finish the calculation.

**imbalanced workload**

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
<th>T3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier 1</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
<tr>
<td>barrier 2</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
</tbody>
</table>

**balanced workload**

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
<th>T3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier 1</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
<tr>
<td>barrier 2</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
</tbody>
</table>
Case Study: CG

- **Sparse Linear Algebra**
  
  - Sparse Linear Equation Systems occur in many scientific disciplines.
  
  - Sparse matrix-vector multiplications (SpMxV) are the dominant part in many iterative solvers (like the CG) for such systems.
  
  - number of non-zeros $\ll n^2$
Case Study: CG

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 4 & 4 \end{pmatrix}$$

- Format: compressed row storage
  - store all values and columns in arrays (length $nnz$)
  - store beginning of a new row in a third array (length $n+1$)

```
for (i = 0; i < A.num_rows; i++){
    sum = 0.0;
    for (nz=A.row[i]; nz<A.row[i+1]; ++nz){
        sum+= A.value[nz]*x[A.index[nz]];
    }
    y[i] = sum;
}

\vec{y} = A \cdot \vec{x}
```
Load Imbalance in VTune

- Grouping execution time of parallel regions by threads helps to detect load imbalance.
- Significant portions of Spin Time also indicate load balance problems.
- Different loop schedules might help to avoid these problems.
Load Imbalance in Vtune

- The Timeline can help to investigate the problem further.

- Zooming in, e.g. to one iteration is also possible.
Parallel Loop Scheduling
Load Balancing
Influencing the For Loop Scheduling

- **for-construct**: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the `schedule` clause:

  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.

  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.

  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.

- **Default on most implementations is schedule(static).**
False Sharing
There is a growing gap between core and memory performance:

- memory, since 1980: 1.07x per year improvement in latency
- single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a per-core basis

Source: John L. Hennessy, Stanford University, and David A. Patterson, University of California, September 25, 2012
Caches

- **CPU is fast**
  - Order of 3.0 GHz

- **Caches:**
  - Fast, but expensive
  - Thus small, order of MB

- **Memory is slow**
  - Order of 0.3 GHz
  - Large, order of GB

- **A good utilization of caches is crucial for good performance of HPC applications!**
Visualization of the Memory Hierarchy

Latency on the Intel Westmere-EP 3.06 GHz processor

Memory Footprint

Latency in ns

L1 cache

L2 cache

L3 cache

1 B 4 B 16 B 64 B 1 KB 4 KB 16 KB 64 KB 256 KB 1 MB 4 MB 12 MB 32 MB 128 MB 512 MB 2 GB

Latency in ns

1 2 4 6 8 10 12 14 16 18 20

OpenMP and Performance
Dirk Schmidl | IT Center der RWTH Aachen University
Data in Caches

- When data is used, it is copied into caches.
- The hardware always copies chunks into the cache, so called *cache-lines*.
- This is useful, when:
  - the data is used frequently (temporal consistency)
  - consecutive data is used which is on the same cache-line (special consistency)
False Sharing

- False Sharing occurs when
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line

- As a result the cache line is moved between the threads, also there is no real dependency

- Note: False Sharing is a performance problem, not a correctness issue
Summing up vector elements again
It’s your turn: Make It Scale!

```c
#pragma omp parallel
{

#pragma omp for
  for (i = 0; i < 99; i++)
  {
    s = s + a[i];
  }

} // end parallel
```

do i = 0, 24
  s = s + a(i)
end do

do i = 25, 49
  s = s + a(i)
end do

do i = 0, 99
  s = s + a(i)
end do

do i = 50, 74
  s = s + a(i)
end do

do i = 75, 99
  s = s + a(i)
end do
False Sharing

double s_priv[nthreads];

#pragma omp parallel num_threads(nthreads)
{
    int t=omp_get_thread_num();

    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t] += a[i];
    }
}

} // end parallel
for (i = 0; i < nthreads; i++)
{
    s += s(priv[i]);
}
False Sharing

- no performance benefit for more threads
- Reason: false sharing of s_priv
- Solution: padding so that only one variable per cache line is used

![Graph showing MFLOPS vs. #threads with and without false sharing]

<table>
<thead>
<tr>
<th>#threads</th>
<th>Standard</th>
<th>With padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 4</td>
<td>1 2 3</td>
</tr>
</tbody>
</table>
NUMA Architectures
Non-Uniform Memory Arch.

**How To Distribute The Data?**

```c
double* A;
A = (double*) malloc(N * sizeof(double));

for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
```

Diagram showing the distribution of data across cores and memory.
About Data Distribution

- Important aspect on cc-NUMA systems
  - If not optimal, longer memory access times and hotspots

- OpenMP does not provide support for cc-NUMA

- Placement comes from the Operating System
  - This is therefore Operating System dependent

- Windows, Linux and Solaris all use the “First Touch” placement policy by default
  - May be possible to override default (check the docs)
Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread

```c
double* A;
A = (double*) malloc(N * sizeof(double));

for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
```
First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition.

```c
double* A;
A = (double*) malloc(N * sizeof(double));
omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    A[i] = 0.0;
}
```
Get Info on the System Topology

Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:

→ Intel MPI’s `cpuinfo` tool
  
  → module switch openmpi intelmpi
  
  → cpuinfo

→ Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.

→ `hwloc`’s `hwloc-1s` tool
  
  → hwloc-1s

→ Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.
Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.

→ Putting threads far apart, i.e. on different sockets
  → May improve the aggregated memory bandwidth available to your application
  → May improve the combined cache size available to your application
  → May decrease performance of synchronization constructs
→ Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
  → May improve performance of synchronization constructs
  → May decrease the available memory bandwidth and cache size

If you are unsure, just try a few options and then select the best one.
OpenMP 4.0: Places + Binding Policies (1/2)

Define OpenMP Places

→ set of OpenMP threads running on one or more processors
→ can be defined by the user, i.e. `OMP_PLACES=cores`

Define a set of OpenMP Thread Affinity Policies

→ SPREAD: spread OpenMP threads evenly among the places
→ CLOSE: pack OpenMP threads near master thread
→ MASTER: collocate OpenMP thread with master thread

Goals

→ user has a way to specify where to execute OpenMP threads for
→ locality between OpenMP threads / less false sharing / memory bandwidth
Places

Assume the following machine:

- 2 sockets, 4 cores per socket, 4 hyper-threads per core

Abstract names for OMP_PLACES:

- threads: Each place corresponds to a single hardware thread on the target machine.
- cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
- sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.
Example’s Objective:
→ separate cores for outer loop and near cores for inner loop

Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)
→ spread creates partition, compact binds threads within respective partition
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4 = cores
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)

Example
→ initial

→ spread 4

→ close 4
Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:

- serial init. / no binding
- serial init. / close binding
- serial init. / spread binding
- NUMA aware init. / close binding
- NUMA aware init. / spread binding

**Serial vs. Parallel Initialization**

**Bandwidth in MB/s**

**Number of Threads**

1 2 4 8 12 16 20 24
Detecting remote accesses
Hardware Counters

Definition: Hardware Performance Counters
In computers, hardware performance counters, or hardware counters are a set of *special-purpose registers* built into modern microprocessors to *store the counts of hardware-related activities* within computer systems. Advanced users often rely on those counters to conduct *low-level performance analysis* or tuning.

(from: http://en.wikipedia.org)
## Hardware Counters of our Intel Nehalem Processor:

### L1I.HITS:
Counts all instruction fetches that hit the L1 instruction cache.

### BR_MISP_EXEC.COND:
Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.

### Hardware Performance Counters

- **OpenMP and Performance**
- **Dirk Schmidl | IT Center der RWTH Aachen University**

---

<table>
<thead>
<tr>
<th>Counter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNC_DRAM_OPEN.CH1, UNC_DRAM_OPEN.CH2, UNC_DRAM_PAGE_CLOSE.CH0, UNC_DRAM_PAGE_CLOSE.CH1, UNC_DRAM_PAGE_CLOSE.CH2, UNC_DRAM_PAGE_CLOSE.CH3</td>
<td>Hardware Counters of our Intel Nehalem Processor:</td>
</tr>
<tr>
<td><strong>L1I.HITS:</strong></td>
<td>Counts all instruction fetches that hit the L1 instruction cache.</td>
</tr>
<tr>
<td><strong>BR_MISP_EXEC.COND:</strong></td>
<td>Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.</td>
</tr>
</tbody>
</table>
Derived Metrics

- **Clock cycles per Instructions (CPI)**
  - CPI indicates if the application is utilizing the CPU or not
  - Take care: Doing “something” does not always mean doing “something useful”.

- **Floating Point Operations per second (FLOPS)**
  - How many arithmetic operations are done per second?
  - Floating Point operations are normally really computing and for some algorithms the number of floating point operations needed can be determined.
Hardware Performance Counters

CPI rate (Clock cycles per instruction): In theory modern processors can finish 4 instructions in 1 cycle, so a CPI rate of 0.25 is possible. A value between 0.25 and 1 is often considered as good for HPC applications.

### Performance Counters

**Elapsed Time:** 1.872s

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Event Count</td>
<td>125,574,000,000</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED.THREAD</td>
<td>6.3462e+10</td>
</tr>
<tr>
<td>INST RETIRED.ANY</td>
<td>6.2112e+10</td>
</tr>
</tbody>
</table>

**CPI Rate:** 1.022

The CPI may be too high. This could be caused by issues such as memory instructions. Explore the other hardware-related metrics to identify what is happening. For example:

- **Retire Stalls:** 0.570s
  - A high number of retire stalls is detected. This may result from branch mispredictions. Use this metric to find where you have stalled instructions. Once
- **LLC Miss:** 0.013s
- **LLC Load Misses Serviced By Remote DRAM:** 0.001s
- **Instruction Starvation:** 0.098s
- **Branch Mispredict:** 0.001s
- **Execution Stalls:** 0.288s
Counters for Remote Traffic

Stream example \( \vec{a} = \vec{b} + s \cdot \vec{c} \) with and without parallel initialization.

→ 2 socket system with Xeon X5675 processors, 12 OpenMP threads

<table>
<thead>
<tr>
<th></th>
<th>copy</th>
<th>scale</th>
<th>add</th>
<th>triad</th>
</tr>
</thead>
<tbody>
<tr>
<td>ser_init</td>
<td>18.8 GB/s</td>
<td>18.5 GB/s</td>
<td>18.1 GB/s</td>
<td>18.2 GB/s</td>
</tr>
<tr>
<td>par_init</td>
<td>41.3 GB/s</td>
<td>39.3 GB/s</td>
<td>40.3 GB/s</td>
<td>40.4 GB/s</td>
</tr>
</tbody>
</table>
Counters for Remote Traffic

- Hardware counters can measure local and remote memory accesses.
  - `MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT` accesses to local memory
  - `MEM_UNCORE_RETIRED.REMOTE_DRAM` accesses to remote memory

- Absolute values are hard to interpret, but the ratio between both is useful.
## Counters for Remote Traffic

- Detecting bad memory accesses for the stream benchmark.

### Ratio of remote memory accesses:

<table>
<thead>
<tr>
<th>Source</th>
<th>copy</th>
<th>scale</th>
<th>add</th>
<th>triad</th>
</tr>
</thead>
<tbody>
<tr>
<td>ser_init</td>
<td>52%</td>
<td>50%</td>
<td>50%</td>
<td>51%</td>
</tr>
<tr>
<td>par_init</td>
<td>0.5%</td>
<td>1.7%</td>
<td>0.6%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>

Percentage of remote accesses for ser_init and par_init stream benchmark.

---

*OpenMP and Performance*

**Dirk Schmidl** | IT Center der RWTH Aachen University
Back to the CG Solver
Case Study CG: Step 1

Hotspot analysis of the serial code:

<table>
<thead>
<tr>
<th>Call Stack</th>
<th>CPU Time: Total by Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg</td>
<td>46.7%</td>
</tr>
<tr>
<td>matvec</td>
<td>40.8%</td>
</tr>
<tr>
<td>xpay</td>
<td>1.4%</td>
</tr>
<tr>
<td>axpy</td>
<td>1.4%</td>
</tr>
<tr>
<td>vectorDot</td>
<td>1.2%</td>
</tr>
<tr>
<td>axpy</td>
<td>1.1%</td>
</tr>
<tr>
<td>vectorDot</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

Hotspots are:
1. matrix-vector multiplication
2. scaled vector additions
3. dot product
Case Study CG: Step 1

Tuning:
- parallelize all hotspots with a parallel for construct
- use a reduction for the dot-product
- activate thread binding
Hotspot analysis of naive parallel version:

<table>
<thead>
<tr>
<th>Event Name</th>
<th>MEM_UNCORE RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT</th>
<th>MEM_UNCORE RETIRED.REMOTE_DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM_UNCORE RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM_UNCORE RETIRED.REMOTE_DRAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A lot of remote accesses occur in nearly all places.

```c
void matvec(const int n, const int
int i, j;
#pragma omp parallel for private(j)
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]]
}
```
Case Study CG: Step 2

Tuning:
- Initialize the data in parallel
- Add parallel for constructs to all initialization loops

- Scalability improved a lot by this tuning on the large machine.
Analyzing load imbalance in the concurrency view:

10 seconds out of ~35 seconds are overhead time
other parallel regions which are called the same amount of time only produce 1 second of overhead
Case Study CG: Step 3

- **Tuning:**

  → pre-calculate a schedule for the matrix-vector multiplication, so that the non-zeros are distributed evenly instead of the rows
The Roofline Model
When to stop tuning?

- Depends on many different factors:
  - How often is the code program used?
  - What are the runtime requirements?
  - Which performance can I expect?

- Investigating kernels may help to understand larger applications.
Roofline Model

- Peak performance of a 4 socket Nehalem Server is 256 GFLOPS.
Roofline Model

- Memory bandwidth measured with Stream benchmark is about 75 GB/s.
The “Roofline” describes the peak performance the system can reach depending on the “operational intensity” of the algorithm.
Roofline Model

Example: Sparse Matrix Vector Multiplication \( y = Ax \)

Given:
- \( x \) and \( y \) are in the cache
- \( A \) is too large for the cache
- measured performance was 12 GFLOPS

- 1 ADD and 1 MULT per element
- load of value (double) and index (int) per element

\[ \frac{2 \text{ Flops}}{12 \text{ Byte}} = \frac{1}{6} \text{ Flops/Byte} \]
Questions?