OpenMP and Performance

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

- $y=x$ (copy)
- $y=s \times x$ (scale)
- $y=x+z$ (add)
- $y=x+s \times z$ (triad)

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];
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Rate (MB/s)</th>
<th>Avg time</th>
<th>Min time</th>
<th>Max time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy:</td>
<td>33237.0185</td>
<td>0.0050</td>
<td>0.0048</td>
<td>0.0055</td>
</tr>
<tr>
<td>Scale:</td>
<td>33304.6471</td>
<td>0.0049</td>
<td>0.0048</td>
<td>0.0059</td>
</tr>
<tr>
<td>Add:</td>
<td>35456.0586</td>
<td>0.0070</td>
<td>0.0068</td>
<td>0.0073</td>
</tr>
<tr>
<td>Triad:</td>
<td>36030.9600</td>
<td>0.0069</td>
<td>0.0067</td>
<td>0.0072</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th></th>
<th>T1:</th>
<th>T2:</th>
<th>T3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>imbalanced</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
<tr>
<td>workload</td>
<td></td>
<td></td>
<td>barrier</td>
</tr>
<tr>
<td>balanced</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>barrier</td>
</tr>
<tr>
<td>balanced</td>
<td>work</td>
<td>work</td>
<td>work</td>
</tr>
<tr>
<td>workload</td>
<td></td>
<td></td>
<td>barrier</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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)

```c
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;
}
```

\[ \vec{y} = A \ast \vec{x} \]
Load Imbalance in VTune

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

<table>
<thead>
<tr>
<th>Basic Hotspots</th>
<th>Hotspots by CPU Usage viewpoint (change)</th>
<th>❜</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grouping:</td>
<td>Process / Function / Thread / Call Stack</td>
<td>❓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process / Function / Thread / Call Stack</th>
<th>CPU Time by Utilization</th>
<th>Overhead and Spin Time</th>
<th>Module</th>
<th>Start Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Idle</td>
<td>Poor</td>
<td>Ok</td>
<td>Ideal</td>
</tr>
<tr>
<td>kernel_smvx-FASTexe</td>
<td>![CPU utilization graph]</td>
<td>![Overhead and Spin Time]</td>
<td>![Module]</td>
<td>![Start Address]</td>
</tr>
<tr>
<td>run_loop$omp$parallel_for@95</td>
<td>14.906s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>_start (TID: 28437)</td>
<td>3.708s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>COMP Worker Thread #1 (TID: 28583)</td>
<td>2.810s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>COMP Worker Thread #2 (TID: 28584)</td>
<td>2.639s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>COMP Worker Thread #3 (TID: 28585)</td>
<td>2.319s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>COMP Worker Thread #4 (TID: 28586)</td>
<td>1.720s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>COMP Worker Thread #5 (TID: 28587)</td>
<td>1.710s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>[OpenMP worker]</td>
<td>7.536s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>7.536s</td>
</tr>
<tr>
<td>run_loop$omp$parallel_for@45</td>
<td>0.891s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td><code>lapack::load_drops_matlab_matrix&lt;double,int&gt;</code></td>
<td>0.030s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0s</td>
</tr>
<tr>
<td>[OpenMP fork]</td>
<td>0.010s</td>
<td>![High CPU utilization]</td>
<td>0s</td>
<td>0.010s</td>
</tr>
</tbody>
</table>
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
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!
Latency on the Intel Westmere-EP 3.06 GHz processor

Visualization of the Memory Hierarchy

Latency in ns

Memory Footprint

L1 cache

L2 cache

L3 cache

1 B, 4 B, 16 B, 64 B, 256 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
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 locality)
  - consecutive data is used which is on the same cache-line (spatial locality)
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
```
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

<table>
<thead>
<tr>
<th>#threads</th>
<th>Standard</th>
<th>With padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

![Graph showing MFLOPS vs. number of threads with and without false sharing](image)

- Blue line: with false sharing
- Orange line: without false sharing

- **cache line 1**
  - Standard: 1 2 3 4
  - With padding: 1 2 3

- **cache line 2**
  - Standard: ...
  - With padding: 2 3 ...

---

OpenMP and Performance  
C. Terboven | IT Center der RWTH Aachen University
double s_priv[nthreads * 8];

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

    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t * 8] += a[i];
    }
}

    // end parallel
for (i = 0; i < nthreads; i++)
{
    s += s_priv[i * 8];
}
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;
}
```
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)
Non-Uniform Memory Arch.

- 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;
}
```
Non-Uniform Memory Arch.

- 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` tools
  - `lstopo` *(command line: 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
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
Detecting remote accesses
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

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

1. 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.

Elapsed Time: 1.872s

| Hardware Event Count:              | 125,574,000,000 |
| CPU_CLK_UNHALTED.THREAD:          | 6.3462e+10      |
| INST_RETIRED.ANY:                 | 6.2112e+10      |
| CPI Rate:                         | 1.022           |
| Retire Stalls:                    | 0.570s          |
| 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          |

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 causing the high CPI rate.
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>

ser_init:

\[
\begin{align*}
\text{a}[0,N-1] & \rightarrow \text{CPU 0} & \text{T1} & \text{T2} & \text{T3} \\
\text{b}[0,N-1] & \rightarrow \text{CPU 0} & \text{T4} & \text{T5} & \text{T6} \\
\text{c}[0,N-1] & \rightarrow \text{MEM} & \text{T7} & \text{T8} & \text{T9} \\
\end{align*}
\]

par_init:

\[
\begin{align*}
\text{a}[0,(N/2)-1] & \rightarrow \text{CPU 0} & \text{T1} & \text{T2} & \text{T3} \\
\text{b}[0,(N/2)-1] & \rightarrow \text{CPU 0} & \text{T4} & \text{T5} & \text{T6} \\
\text{c}[0,(N/2)-1] & \rightarrow \text{MEM} & \text{T7} & \text{T8} & \text{T9} \\
\text{a}[N/2,N-1] & \rightarrow \text{CPU 1} & \text{T10} & \text{T11} & \text{T12} \\
\text{b}[N/2,N-1] & \rightarrow \text{CPU 1} & \text{T10} & \text{T11} & \text{T12} \\
\text{c}[N/2,N-1] & \rightarrow \text{MEM} & \text{T10} & \text{T11} & \text{T12} \\
\end{align*}
\]
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.

<table>
<thead>
<tr>
<th>Source</th>
<th>Assembly</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ifdef TUNED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tuned_STREAM_Scale(scalar);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td></td>
<td></td>
</tr>
<tr>
<td>for (j=0; j&lt;N; j++)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b[i] = scalar*c[i];</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#endif</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Ratio of remote memory accesses:**

<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>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.
Case Study: 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></td>
<td></td>
</tr>
<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
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>20,000</td>
<td>0</td>
</tr>
<tr>
<td>MEM_UNCORE RETIRED.REMOTE_DRAM</td>
<td>0</td>
<td>0</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]]
       }
```
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.
Case Study CG: Step 3

- Analyzing load imbalance in the concurrency view:

<table>
<thead>
<tr>
<th>So.. Line</th>
<th>Source</th>
<th>CPU Time: Total by...</th>
<th>Over... and...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Idle</td>
<td>Poor</td>
</tr>
<tr>
<td>49</td>
<td>void matvec(const int n, const int nnz,</td>
<td>22.462s</td>
<td>10.612s</td>
</tr>
<tr>
<td>50</td>
<td>int i,j;</td>
<td>0.050s</td>
<td>0s</td>
</tr>
<tr>
<td>51</td>
<td>#pragma omp parallel for private(j)</td>
<td>0.060s</td>
<td>0s</td>
</tr>
<tr>
<td>52</td>
<td>for(i=0; i&lt;n; i++){</td>
<td>1.741s</td>
<td>0s</td>
</tr>
<tr>
<td>53</td>
<td>y[i]=0;</td>
<td>9.998s</td>
<td>0s</td>
</tr>
<tr>
<td>54</td>
<td>for(j=ptr[i]; j&lt;ptr[i+1]; j++){</td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>y[i]+=value[j]*x[index[j]];</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

![Graph showing runtime and speedup vs. number of threads](image-url)
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.
Roofline Model

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
  -> 2 Flops / 12 Byte = 1/6 Flops/Byte
Questions?