Microbenchmarking for architectural exploration

Probing of the memory hierarchy
Saturation effects
OpenMP barrier overhead
Motivation for Microbenchmarking as a tool

- Isolate small kernels to:
  - Separate influences
  - Determine specific machine capabilities (light speed)
  - Gain experience about software/hardware interaction
  - Determine programming model overhead
  - …

- Possibilities:
  - Readymade benchmark collections (epcc OpenMP, IMB)
  - STREAM benchmark for memory bandwidth
  - Implement own benchmarks (difficult and error prone)
  - likwid-bench tool: Offers collection of benchmarks and framework for rapid development of assembly code kernels
double striad_seq(double* restrict a, double* restrict b, double* restrict c, double* restrict d, int N, int iter) {
    double S, E;
    S = getTimeStamp();
    for(int j = 0; j < iter; j++) {
        #pragma vector aligned
        for (int i = 0; i < N; i++) {
            a[i] = b[i] + d[i] * c[i];
        }
        if (a[N-1] > 2000) printf("Ai = %f\n",a[N-1]);
    }
    E = getTimeStamp();
    return E-S; }

- Report performance for different N, choose iter so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all architectures, ever!
Schönauer triad on **one** CascadeLake **core** 2.5GHz

![Graph showing performance metrics](image-url)

- **L1**: 32kB
- **L2**: 1MB
- **L3**: 28MB
Schönauer triad on **one** CascadeLake core **2.5GHz**

What are the theoretical limits?
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark → pure hardware probing, no impact from OpenMP overhead

```c
#pragma omp parallel
{
    double* al;
    posix_memalign((void**) &al, ARRAY_ALIGNMENT, N * sizeof(double));
#pragma omp single
    S = getTimeStamp();
    for(int j=0; j<iter; j++) {
#pragma vector aligned
        for (int i=0; i<N; i++) {
            al[i] = b[i] + d[i] * c[i];
        }
        if (al[N-1] > 2000) printf("Ai = %f\n",al[N-1]);
    }
#pragma omp single
    E = getTimeStamp();
}
```

Every thread works on private copy of a!
Throughput vector triad on one CascadeLake node (2.5 GHz)

Performance scales in L1 / L2 cache levels!

Drop stays at the same place for private caches!

L3 cache is not scalable

Adding another socket doubles the performance without changing the shape!
Throughput vector triad on CascadeLake (memory close-up)

- Saturating L3 cache performance
- Second sockets adds another memory interface!
- Performance saturation in main memory!
Attainable memory bandwidth (UPDATE!)

Intel Broadwell (22 cores)  
CoD enabled

AMD Naples (24 cores)

BW saturation in NUMA domain

Single core does not saturate BW

Cavium ThunderX2 (ARM)  
DDR4-2400 memory

NVIDIA P100 (Pascal)
The OpenMP-parallel vector triad benchmark

OpenMP **work sharing** in the benchmark loop

```
S = getTimeStamp();
#pragma omp parallel
{
    for(int j = 0; j < iter; j++) {
#pragma omp for
#pragma vector aligned
        for (int i=0; i<N; i++) {
            a[i] = b[i] + d[i] * c[i];
        }
        if (a[N-1] > 2000) printf("Ai = %f\n",a[N-1]);
    }
}
E = getTimeStamp();
```

**Implicit barrier**
OpenMP vector triad on CascadeLake node (2.2 GHz)

- **Sequential**
- **T1**
- **T20 (1 socket)**
- **T40 (2 sockets)**

- **Impact on performance even with 1 thread**
- **Sync overhead grows with number of threads**
OpenMP performance issues on multicore

Synchronization (barrier) overhead
Welcome to the multi-/many-core era

Synchronization of threads may be expensive!

 Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP programs.

Determine costs via simple benchmark

On x86 systems there is no hardware support for synchronization!

- Next slide: Test OpenMP Barrier performance…
- for different compilers
- and different topologies:
  - shared cache
  - shared socket
  - between sockets
- and different thread counts
  - 2 threads
  - full domain (chip, socket, node)
Thread synchronization overhead on IvyBridge-EP

Barrier overhead in CPU cycles

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>599</td>
<td>425</td>
</tr>
<tr>
<td>SMT threads</td>
<td>612</td>
<td>423</td>
</tr>
<tr>
<td>Other socket</td>
<td>1486</td>
<td>1067</td>
</tr>
</tbody>
</table>

Strong topology dependence!

<table>
<thead>
<tr>
<th>Full domain</th>
<th>Intel 16.0</th>
<th>GCC 5.3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket (10 cores)</td>
<td>1934</td>
<td>1301</td>
</tr>
<tr>
<td>Node (20 cores)</td>
<td>4999</td>
<td>7783</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>5981</td>
<td>9897</td>
</tr>
</tbody>
</table>

2.2 GHz

- Strong dependence on compiler, CPU and system environment!
- `OMP_WAIT_POLICY=ACTIVE` can make a big difference

Overhead grows with thread count
Scaling of barrier cost

Comparison of barrier synchronization cost with increasing number of threads

1. 2x Haswell 14-core CoD mode
2. Optimistic measurements (repeated 1000s of times)
3. No impact from previous activity in cache
4. Ideal scaling: logarithmic

![Graph showing barrier overhead with increasing number of cores for Intel 17.0.4 and gcc 6.2.0.](image)
Conclusions from the microbenchmarks

- **Microbenchmarks can yield surprisingly deep insights**

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - *Know where your threads are running*
    - *Know where your data is* (see later for that)

- **Bandwidth bottlenecks are ubiquitous**

- **Synchronization overhead may be an issue**
  - … and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization