Performance-oriented programming on multicore-based systems, with a focus on the Cray XE6

Georg Hager\(^{(a)}\), Jan Treibig\(^{(a)}\), and Gerhard Wellein\(^{(a,b)}\)

\(^{(a)}\)HPC Services, Erlangen Regional Computing Center (RRZE)

\(^{(b)}\)Department for Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

Cray XE6 optimization workshop, November 5-8, 2012, HLRS
There is no alternative to knowing what is going on between your code and the hardware

Without performance modeling, optimizing code is like stumbling in the dark

Performance x Flexibility = constant
a.k.a. Abstraction is the natural enemy of efficiency
Agenda

- Basics of multicore processor and node architecture
- Probing node topology with likwid-topology
- Data access on modern processors
  - Basic performance benchmarks and properties
  - The balance metric: Bandwidth-based performance modeling
  - Optimizing data access by code transformations
- Enforcing affinity in multicore environments
- Performance properties of parallel code on multicore processors and nodes
  - Exploration by microbenchmarks
  - Sparse matrix-vector multiplication
- Microarchitectural features of modern processors
  - SIMD parallelism
  - A closer look at the cache hierarchy
  - Performance modeling on the microarchitecture level
- ccNUMA: Properties and efficient programming
Multicore processor and system architecture

Basics
The x86 multicore evolution so far
*Intel Single-Dual-/Quad-/Hexa-/-Cores (one-socket view)*

<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>“Fake” dual-core</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>2006</td>
<td>True dual-core</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>2008</td>
<td>Simultaneous Multi Threading (SMT)</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>2010</td>
<td>6-core chip</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>2012</td>
<td>Wider SIMD units AVX: 256 Bit</td>
<td>![Diagram](attachment:wider simd.png)</td>
</tr>
</tbody>
</table>

**Cray XE6 Workshop**  **Performance for Multicore**
There is no longer a single driving force for chip performance!

Floating Point (FP) Performance:

\[ P = n_{\text{core}} \times F \times S \times \nu \]

- \( n_{\text{core}} \) \text{ number of cores: } 8
- \( F \) \text{ FP instructions per cycle: } 2
  (1 MULT and 1 ADD)
- \( S \) \text{ FP ops / instruction: } 4 (dp) / 8 (sp)
  (256 Bit SIMD registers – “AVX”)
- \( \nu \) \text{ Clock speed: } 2.5 \text{ GHz}

\[ P = 160 \text{ GF/s (dp)} / 320 \text{ GF/s (sp)} \]

But: \( P=5 \text{ GF/s (dp)} \) for serial, non-SIMD code

Intel Xeon
“Sandy Bridge EP” socket
4,6,8 core variants available

TOP500 rank 1 (1996)
From UMA to ccNUMA

Basic architecture of commodity compute cluster nodes

Yesterday (2006): Dual-socket Intel “Core2” node:

- Uniform Memory Architecture (UMA)
- Flat memory; symmetric MPs
- But: system “anisotropy”

Today: Dual-socket Intel (Westmere) node:

- Cache-coherent Non-Uniform Memory Architecture (ccNUMA)
- HT / QPI provide scalable bandwidth at the price of ccNUMA architectures:
  Where does my data finally end up?

On AMD it is even more complicated → ccNUMA within a socket!
Back to the 2-chip-per-case age
12 core AMD Magny-Cours – a 2x6-core ccNUMA socket

- **AMD: single-socket ccNUMA since Magny Cours**
  - 1 socket: 12-core Magny-Cours built from two 6-core chips → 2 NUMA domains
  - 2 socket server → 4 NUMA domains
  - 4 socket server: → 8 NUMA domains

- WHY? → Shared resources are hard two scale: 2 x 2 memory channels vs. 1 x 4 memory channels per socket
Another flavor of “SMT”

AMD Interlagos / Bulldozer

- Up to 16 cores (8 Bulldozer modules) in a single socket
- Max. 2.6 GHz (+ Turbo Core)
- $P_{\text{max}} = (2.6 \times 8 \times 8) \text{ GF/s} = 166.4 \text{ GF/s}$

Each Bulldozer module:
- 2 “lightweight” cores
- 1 FPU: 4 MULT & 4 ADD (double precision) / cycle
- Supports AVX
- Supports FMA4

2 DDR3 (shared) memory channel > 15 GB/s
Cray XE6 (Hermit) “Interlagos” 16-core dual socket node

- Two 8- (integer-) core chips per socket @ 2.3 GHz (3.3 @ turbo)
- Separate DDR3 memory interface per chip
  - ccNUMA on the socket!
- Shared FP unit per pair of integer cores (“module”)
  - “256-bit” FP unit
  - SSE4.2, AVX, FMA4
- 16 kB L1 data cache per core
- 2 MB L2 cache per module
- 8 MB L3 cache per chip (6 MB usable)
Parallel programming models
on multicore multisocket nodes

- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 2.2)
  - OpenMP (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks
  - Cilk++, OpenCL, StarSs,… you name it

- **Distributed-memory (inter-node)**
  - MPI (current standard: 2.2)
  - PVM (gone)

- **Hybrid**
  - Pure MPI
  - MPI+OpenMP
  - MPI + any shared-memory model

All models require awareness of topology and affinity issues for getting best performance out of the machine!
Parallel programming models:

**Pure MPI**

- **Machine structure is invisible to user:**
  - → Very simple programming model
  - → MPI “knows what to do”!

- **Performance issues**
  - Intranode vs. internode MPI
  - Node/system topology

![Diagram of communication network and processor-memory connections]
Parallel programming models:
*Pure threading on the node*

- **Machine structure is invisible to user**
  - Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,...) should know about the details
- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology
Parallel programming models:
Hybrid MPI+OpenMP on a multicore multisocket cluster

One MPI process / node

One MPI process / socket: 
OpenMP threads on same socket: “blockwise”

OpenMP threads pinned “round robin” across cores in node

Two MPI processes / socket 
OpenMP threads on same socket

Cray XE6 Workshop  Performance for Multicore
Probing node topology

- Standard tools
- likwid-topology
How do we figure out the node topology?

- **Topology**
  - Where in the machine does core \( n \) reside? And do I have to remember this awkward numbering anyway?
  - Which cores share which cache levels?
  - Which hardware threads (“logical cores”) share a physical core?

- **Linux**
  - `cat /proc/cpuinfo` is of limited use
  - Core numbers may change across kernels and BIOSes even on identical hardware
  - `numactl --hardware` prints ccNUMA node information
  - Information on caches is harder to obtain

```bash
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 8192 MB
node 3 free: 7840 MB
```
How do we figure out the node topology?

- **LIKWID** tool suite:

  Like I Knew What I’m Doing

- Open source tool collection (developed at RRZE):

  http://code.google.com/p/likwid


http://arxiv.org/abs/1004.4431
Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **Current tools:**
  - `likwid-topology`: Print thread and cache topology
  - `likwid-pin`: Pin threaded application without touching code
  - `likwid-perfctr`: Measure performance counters
  - `likwid-mpirun`: mpirun wrapper script for easy LIKWID integration
  - `likwid-bench`: Low-level bandwidth benchmark generator tool
  - ... some more
likwid-topology – Topology information

- **Based on cpuid information**

- **Functionality:**
  - Measured clock frequency
  - Thread topology
  - Cache topology
  - Cache parameters (-c command line switch)
  - ASCII art output (-g command line switch)

- **Currently supported (more under development):**
  - Intel Core 2 (45nm + 65 nm)
  - Intel Nehalem + Westmere (Sandy Bridge in beta phase)
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8
  - Linux OS
Output of `likwid-topology -g`
on one node of Cray XE6 “Hermit”

---

**CPU type:** AMD Interlagos processor

**Hardware Thread Topology**

<table>
<thead>
<tr>
<th>Sockets</th>
<th>Cores per socket</th>
<th>Threads per core</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

...[

16       | 0      | 0    | 1      |
17       | 0      | 1    | 1      |
18       | 0      | 2    | 1      |
19       | 0      | 3    | 1      |
...

Socket 0: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 )
Socket 1: ( 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 )

---

**Cache Topology**

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Cache groups</th>
</tr>
</thead>
</table>
Output of likwid-topology continued

Level: 2
Size: 2 MB

Level: 3
Size: 6 MB
Cache groups: (0 1 2 3 4 5 6 7) (8 9 10 11 12 13 14 15) (16 17 18 19 20 21 22 23) (24 25 26 27 28 29 30 31)

*******************************************
NUMA Topology
*******************************************
NUMA domains: 4

Domain 0:
Processors: 0 1 2 3 4 5 6 7
Memory: 7837.25 MB free of total 8191.62 MB

Domain 1:
Processors: 8 9 10 11 12 13 14 15
Memory: 7860.02 MB free of total 8192 MB

Domain 2:
Processors: 16 17 18 19 20 21 22 23
Memory: 7847.39 MB free of total 8192 MB

Domain 3:
Processors: 24 25 26 27 28 29 30 31
Memory: 7785.02 MB free of total 8192 MB
### Output of likwid-topology continued

<table>
<thead>
<tr>
<th>Socket 0:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td>16kB</td>
</tr>
<tr>
<td>6MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Socket 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>16kB</td>
</tr>
<tr>
<td>6MB</td>
</tr>
</tbody>
</table>
Data access on modern processors

Characterization of memory hierarchies
Balance analysis and light speed estimates
Data access optimization
Latency and bandwidth in modern computer environments

We care about this region today

Avoiding slow data paths is the key to most performance optimizations!
Interlude: Data transfers in a memory hierarchy

- How does data travel from memory to the CPU and back?
- Example: Array copy $A(:)=C(:)$

![Diagram showing data transfer in a memory hierarchy with standard stores and nontemporal (NT) stores]

**Standard stores**

- LD $C(1)$
  - CPU registers
  - Cache
    - Miss
    - Evict (delayed)
    - Write allocate
    - 3 CL transfers

- ST $A(1)$
  - CPU registers
  - Cache
    - Miss
    - Hit

**Nontemporal (NT) stores**

- LD $C(2..N_{cl})$
  - CPU registers
  - Cache
    - Hit
    - NTST $A(2..N_{cl})$

- NTST $A(1)$
  - CPU registers
  - Cache
    - Hit

50% performance boost for COPY

Cray XE6 Workshop  Performance for Multicore  25
The parallel vector triad benchmark
A “swiss army knife” for microbenchmarking

Simple streaming benchmark:

```fortran
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

do j=1,NITER
    do i=1,N
        A(i) = B(i) + C(i) * D(i)
    enddo
    if (.something.that.is.never.true.) then
        call dummy(A,B,C,D)
    endif
endo
```

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!
A( : ) = B( : ) + C( : ) * D( : ) on one Interlagos core
STREAM benchmarks:
Memory bandwidth on Cray XE6 Interlagos node

- STREAM is the “standard” for memory BW comparisons
- NT store variants save write allocate on stores \(\rightarrow 50\%\) boost for copy, 33\% for TRIAD
- STREAM BW is practical limit for all codes
Balance metric: Machine balance

- The **machine balance** for data memory access of a specific computer is given by (architectural limitation):
  \[ B_m = \frac{b_s \text{ [words/s]}}{P_{\text{max}} \text{ [flops/s]}} \]

- **Bandwidth:**
  \[ 1 \text{ W} = 8 \text{ bytes} = 64 \text{ bits} \]
  \[ b_s = \text{achievable bandwidth over the slowest data path} \]

Floating point peak: \[ P_{\text{max}} \]

- **Machine Balance** = How many input operands can be delivered for each FP operation?

- **Typical values (main memory):**
  - AMD Interlagos (2.3 GHz): \[ B_m = \{(17/8) \text{ GW/s}\} / \{4 \times 2.3 \times 8 \text{ GFlop/s}\} \sim 0.029 \text{ W/F} \]
  - Intel Sandy Bridge EP (2.7 GHz): \sim 0.025 \text{ W/F} \]
  - NEC SX9 (vector): \sim 0.3 \text{ W/F} \]
  - nVIDIA GTX480: \sim 0.026 \text{ W/F} \]
Machine Balance: Typical values beyond main memory

<table>
<thead>
<tr>
<th>Data path</th>
<th>Balance $B_M$ [W/F]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>0.5 – 1.0</td>
</tr>
<tr>
<td>Machine (main memory)</td>
<td>0.01 – 0.5</td>
</tr>
<tr>
<td>Interconnect (Infiniband)</td>
<td>0.001 – 0.002</td>
</tr>
<tr>
<td>Interconnect (GBit ethernet)</td>
<td>0.0001 – 0.0007</td>
</tr>
<tr>
<td>Disk (or disk subsystem)</td>
<td>0.0001 – 0.001</td>
</tr>
</tbody>
</table>

$$\frac{1}{B_M} = \text{“Computational Intensity”: How many FP ops can be performed before FP performance becomes a bottleneck?}$$
Balance metric: Code balance & lightspeed estimates

- $B_M$ tells us what the hardware can deliver at most
- Code balance ($B_C$) quantifies the requirements of the code:

  - Expected fraction of peak performance ("lightspeed"): $l = 1 \rightarrow$ code is not limited by bandwidth

  - Lightspeed for absolute performance: ($P_{\text{max}}$ : “applicable” peak performance)

  - Example: Vector triad $A(:)=B(:)+C(:)\ast D(:)$ on 2.3 GHz Interlagos
    - $B_C = (4+1)\text{ Words} / 2\text{ Flops} = 2.5\text{ W/F (including write allocate)}$
    - $B_m/B_C = 0.029/2.5 = 0.012$, i.e. 1.2% of peak performance (~1.7 GF/s)
Balance metric (a.k.a. the “roofline model”)

- The balance metric formalism is based on some (crucial) assumptions:
  - The code makes balanced use of MULT and ADD operation. For others (e.g. A=B+C) the peak performance input parameter $P_{\text{max}}$ has to be adjusted (e.g. $P_{\text{max}} \rightarrow P_{\text{max}}/2$)

- Attainable bandwidth of code = input parameter! Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications.

- Definition is based on 64-bit arithmetic but can easily be adjusted, e.g. for 32-bit

- Data transfer and arithmetic overlap perfectly!

- Slowest data path is modeled only; all others are assumed to be infinitely fast

- Latency effects are ignored, i.e. perfect streaming mode
Balance metric: 2D diffusion equation + Jacobi solver

- **Diffusion equation in 2D**
  \[
  \frac{\partial \Phi}{\partial t} = \Delta \Phi
  \]

- **Stationary solution** with Dirichlet boundary conditions using Jacobi iteration scheme can be obtained with:

  ```
  double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
  integer :: t0,t1
  t0 = 0 ; t1 = 1
  do it = 1,itmax ! choose suitable number of sweeps
    do k = 1,kmax
      do i = 1,imax
        ! four flops, one store, four loads
        phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                       + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
      enddo
    enddo
    ! swap arrays
    i = t0 ; t0=t1 ; t1=i
  enddo
  ```

  **Reuse when computing** \( \phi(i+2,k,t1) \)

  **Balance (crude estimate incl. write allocate):**

  \[ \begin{align*}
  \phi(:,:,t0) : & 3 \text{ LD} + \\
  \phi(:,:,t1) : & 1 \text{ ST} + 1\text{LD} \\
  \Rightarrow \ B_C = & \ 5 \text{ W} / 4 \text{ FLOPs} = 1.25 \text{ W} / \text{F} 
  \end{align*} \]

WRITE ALLOCATE:
LD + ST \( \phi(i,k,t1) \)
Balance metric: 2 D Jacobi

- Modern cache subsystems may further reduce memory traffic

If cache is large enough to hold at least 2 rows (shaded region): Each $\phi(:,:,t0)$ is loaded once from main memory and reused 3 times from cache:

$\phi(:,:,t0)$: 1 LD + $\phi(:,:,t1)$: 1 ST + 1 LD

$B_C = 3 \frac{W}{4 F} = 0.75 \frac{W}{F}$

If cache is large enough to hold at least one row $\phi(:,k-1,t0)$ needs to be reloaded:

$\phi(:,:,t0)$: 2 LD + $\phi(:,:,t1)$: 1 ST + 1 LD

$B_C = 4 \frac{W}{4 F} = 1.0 \frac{W}{F}$

Beyond that:

$\phi(:,:,t0)$: 2 LD + $\phi(:,:,t1)$: 1 ST + 1 LD

$B_C = 5 \frac{W}{4 F} = 1.25 \frac{W}{F}$
Performance metrics: 2D Jacobi

- **Alternative implementation** ("Macho FLOP version")

```
  do k = 1, kmax
    do i = 1, imax
      phi(i, k, t1) = 0.25 * phi(i+1, k, t0) + 0.25 * phi(i-1, k, t0)
      + 0.25 * phi(i, k+1, t0) + 0.25 * phi(i, k-1, t0)
    enddo
  enddo
```

- MFlops/sec increases by 7/4 but time to solution remains the same

- **Better metric** (for many iterative stencil schemes): Lattice Site Updates per Second (LUPs/sec)

2D Jacobi example: Compute LUPs/sec metric via

\[
P[LUPs / s] = \frac{it_{\text{max}} \cdot i_{\text{max}} \cdot k_{\text{max}}}{T_{\text{wall}}}
\]
Balance metric for 3D Jacobi

- **3D sweep:**

  ```
  do k=1,kmax
    do j=1,jmax
      do i=1,imax
        phi(i,j,k,t1) = oos * (phi(i-1,j,k,t0)+phi(i+1,j,k,t0) & 
                              + phi(i,j-1,k,t0)+phi(i,j+1,k,t0) & 
                              + phi(i,j,k-1,t0)+phi(i,j,k+1,t0))
      enddo
    enddo
  enddo
  ```

- **Best case balance:** 1 LD
  - 1 ST + 1 write allocate
  - 6 flops
  \[ B_C = 0.5 \text{ W/F (24 bytes/update)} \]

- **If 2-layer condition does not hold but 2 rows fit:**
  \[ B_C = 5/6 \text{ W/F (40 bytes/update)} \]

- **Worst case (2 rows do not fit):**
  \[ B_C = 7/6 \text{ W/F (56 bytes/update)} \]
3D Jacobi solver

Performance of vanilla code on one Interlagos chip (8 cores)

Problem size: $N^3$

- Cache
- Memory

2 layers of source array drop out of L2 cache

Performance model (mem.)

- $T=1$
- $T=2$
- $T=8$

Performance [MLup/s] vs Linear problem size $N$

24B/update

40B/update
Data Access Optimizations
General considerations
Case study: Optimizing a Jacobi solver
Data access – general considerations

- **Case 1: O(N)/O(N) Algorithms**
  - O(N) arithmetic operations vs. O(N) data access operations
  - Examples: Scalar product, vector addition, sparse MVM etc.
  - Performance limited by memory BW for large N (“memory bound”)
  - Limited optimization potential for single loops
    - …at most a constant factor for multi-loop operations
  - Example: successive vector additions

```
  do i=1,N
    a(i)=b(i)+c(i)
  enddo

  do i=1,N
    z(i)=b(i)+e(i)
  enddo
```

- Loop fusion

```
  do i=1,N
    a(i)=b(i)+c(i)
    z(i)=b(i)+e(i)
  enddo
```

- Fusing different loops allows O(N) data reuse from registers

```
  B_c = 7/2 W/F
```

```
  B_c = 4/1 W/F
  (incl. write-allocate)
```
Data access – general guidelines

- **Case 2: O(N²)/O(N²) algorithms**
  - Examples: dense matrix-vector multiply, matrix addition, dense matrix transposition etc.
    - Nested loops
  - Memory bound for large N
  - Some optimization potential (at most constant factor)
    - Can often enhance code balance by outer loop unrolling or spatial blocking
  - Example: dense matrix-vector multiplication

```plaintext
   do i=1,N
      do j=1,N
         c(i) = c(i) + a(i,j) * b(j)
      enddo
   enddo
```

Naïve version loads \( b[ ] \) \( N \) times!
Data access – general guidelines

- **O(N^2)/O(N^2) algorithms cont’d**
  - “Unroll & jam” optimization (or “outer loop unrolling”)
  
  ```
  do i=1,N
    do j=1,N
      c(i)=c(i)+a(i,j)*b(j)
    enddo
  enddo
  ```

  **Unroll**
  ```
  do i=1,N,2
    do j=1,N
      c(i)=c(i)+a(i,j)*b(j)
    enddo
  enddo
  ```

  **Jam**
  ```
  do i=1,N,2
    do j=1,N
      c(i)=c(i)+a(i,j)*b(j)
      c(i+1)=c(i+1)+a(i+1,j)*b(j)
    enddo
  enddo
  ```

  *b(j)* can be re-used once from register → save 1 LD operation
  
  Lowers *B_c* from 1 to \(\frac{3}{4}\) W/F
Data access – general guidelines

- **O(N^2)/O(N^2) algorithms cont’d**
  - Data access pattern for 2-way unrolled dense MVM:

  \[
  \begin{align*}
  \text{Vector } b[i] & \text{ now only loaded } N/2 \text{ times!} \\
  \text{Remainder loop handled separately}
  \end{align*}
  \]

  ![Diagram showing data access pattern]

  - Data transfers can further be reduced by more aggressive unrolling (i.e., m-way instead of 2-way)
  - Significant code bloat (try to use compiler directives if possible)
    - Main memory limit: \( b[i] \) only be loaded once from memory (\( B_c \approx ½ W/F \)) (can be achieved by high unrolling OR large outer level caches)
    - **Outer loop unrolling can also be beneficial to reduce traffic within caches!**
    - Beware: CPU registers are a limited resource
    - Excessive unrolling can cause register spills to memory
Optimizing data access for dense matrix transpose
Dense matrix transpose

- Simple example for data access problems in cache-based systems
- Naïve code:
  ```
  do i=1,N
    do j=1,N
      a(j,i) = b(i,j)
    enddo
  enddo
  ```

- Problem: Stride-1 access for a implies stride-N access for b
  - Access to a is perpendicular to cache lines (—)
  - Possibly bad cache efficiency (spatial locality)

- Three performance levels are expected:
  - C: Cache size; $L_C$: Cache line length; both are given in double words (8 byte)
  - $2 \times N^2 < C$: Both matrices stay in cache
  - $N \times L_C + N < C$: N cache lines of b and one row of a stays in cache
  - $N \times L_C + N > C$: Matrix b is reloaded from memory $L_C$ times

- Use outer loop unrolling blocking to reduce / avoid second drop
Dense matrix transpose: Base version

Second drop: cache lines of b are evicted before they can be reused

2\textsuperscript{nd} drop:
- 1 MB cache \( \Rightarrow N_c \sim 7.530 \)
- 4 MB cache \( \Rightarrow N_c \sim 30.000 \)

Rule of thumb: use C/2 as effective cache size
\( \Rightarrow N_C \rightarrow N_C/2 \)
Dense matrix transpose: Unrolling and blocking

\[
\begin{align*}
&\text{do } i=1,N \\
&\quad \text{do } j=1,N \\
&\quad \quad a(j,i) = b(i,j) \\
&\quad \text{enddo} \\
&\text{enddo} \\
\end{align*}
\]

\[
\begin{align*}
&\text{do } i=1,N,U \\
&\quad \text{do } j=1,N \\
&\quad \quad a(j,i) = b(i,j) \\
&\quad \quad a(j,i+1) = b(i+1,j) \\
&\quad \quad \ldots \\
&\quad \quad a(j,i+U-1) = b(i+U-1,j) \\
&\text{enddo} \\
&\text{enddo} \\
\end{align*}
\]

\[
\begin{align*}
&\text{do } ii=1,N,B \\
&\quad \text{istart}=ii; \ iend=ii+B-1 \\
&\text{do } jj=1,N,B \\
&\quad \jstart=jj; \ jend=jj+B-1 \\
&\text{do } i=\jstart,\iend,\U \\
&\quad \text{do } j=\jstart,\jend \\
&\quad \quad a(j,i) = b(i,j) \\
&\quad \quad a(j,i+1) = b(i+1,j) \\
&\quad \quad \ldots \\
&\quad \quad a(j,i+U-1) = b(i+U-1,j) \\
&\text{enddo} \\
&\text{enddo} \\
&\text{enddo} \\
&\text{enddo} \\
\end{align*}
\]

Blocking and unrolling factors \((B,U)\) can be determined experimentally; be guided by cache sizes and line lengths
Dense matrix transpose: Blocked/unrolled versions

- Intel Xeon/Netburst 3.2 GHz

Breakdown only eliminated by blocking!
Case study:
3D Jacobi solver

Spatial blocking for improved cache utilization
Remember the 3D Jacobi solver?

The graph shows performance in MLup/s against linear problem size N. The performance drops significantly when the number of layers of source array exceeds the L2 cache size (24B/update for T=1, 40B/update for T=2, T=8). To avoid this, spatial blocking can be used.

- 2 layers of source array drop out of L2 cache
- Avoid through spatial blocking!
Jacobi iteration (2D): No spatial Blocking

- **Assumptions:**
  - Cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array

This element is needed for three more updates; but 29 updates happen before this element is used for the last time
Jacobi iteration (2D): No spatial blocking

Assumptions:
- Cache can hold 32 elements (16 for each array)
- Cache line size is 4 elements
- Perfect eviction strategy for source array

This element is needed for three more updates but has been evicted
Jacobi iteration (2D): Spatial Blocking

- Divide system into blocks
- Update block after block
- Same performance as if three complete rows of the systems fit into cache
Spatial blocking reorders traversal of data to account for the data update rule of the code

→ Elements stay sufficiently long in cache to be fully reused

→ Spatial blocking improves temporal locality!
   (Continuous access in inner loop ensures spatial locality)

This element remains in cache until it is fully used (only 6 updates happen before last use of this element)
Jacobi iteration (2D): Spatial blocking

- **Implementation:**

  ```plaintext
do it=1,itmax
  do ioffset=1,imax,iblock
    do k=1,kmax
      do i=ioffset, min(imax,ioffset+iblock-1)
        phi(i, k, t1) = ( phi(i-1, k, t0) + phi(i+1, k, t0) + phi(i, k-1, t0) + phi(i, k+1, t0) )*0.25
      enddo; enddo; enddo; enddo
  enddo; enddo; enddo; enddo
```

- **Guidelines:**
  - Blocking of inner loop levels (traversing continuously through main memory)
  - Blocking size \texttt{iblock} large enough to keep elements sufficiently long in cache but cache size is a hard limit!
  - Blocking loops may have some impact on ccNUMA page placement (see later)
3D Jacobi solver (problem size $400^3$)

Blocking different loop levels (8 cores Interlagos)

- 3D vs. 2D?
- OpenMP parallelization?
- Optimal block size?
- k-loop blocking?

$\rightarrow$ see Exercise!

inner (i) loop blocking

middle (j) loop blocking

optimum j block size

24B/update performance model

Cray XE6 Workshop

Performance for Multicore
3D Jacobi solver
Spatial blocking + nontemporal stores

![Graph showing performance vs linear problem size N for T=8 and T=8 bs=20 with nontemporal stores. The graph indicates a 50% expected performance boost with nontemporal stores.](image)

- **Performance** vs **Linear problem size N**
- **Graphs** for:
  - T=8
  - T=8 bs=20
  - T=8 bs=20 NT stores

**Legend**:
- **16 B/update perf. model**
- **Blocking**
- **NT stores expected boost: 50%**
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
- aprun (Cray)
Example: STREAM benchmark on 12-core Intel Westmere: Anarchy vs. thread pinning

There are several reasons for caring about affinity:
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention
Generic thread/process-core affinity under Linux

**Overview**

- `taskset [OPTIONS] [MASK | -c LIST] \ [PID | command [args]...]`

- `taskset` binds processes/threads to a set of CPUs. Examples:

  ```
  taskset 0x0006 ./a.out
  taskset -c 4 33187
  mpirun -np 2 taskset -c 0,2 ./a.out # doesn’t always work
  ```

- Processes/threads can still move within the set!

- **Alternative:** let process/thread bind itself by executing syscall

  ```
  #include <sched.h>
  int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask);
  ```

- **Disadvantage:** which CPUs should you bind to on a non-exclusive machine?

- Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA
Generic thread/process-core affinity under Linux

- Complementary tool: `numactl`

**Example:** `numactl --physcpubind=0,1,2,3` command [args]
Bind process to specified physical core numbers

**Example:** `numactl --cpunodebind=1` command [args]
Bind process to specified ccNUMA node(s)

- Many more options (e.g., interleave memory across nodes)
  - → see section on ccNUMA optimization

- Diagnostic command (see earlier):
  `numactl --hardware`

- Again, this is not suitable for a shared machine
More thread/Process-core affinity (“pinning”) options

- **Highly OS-dependent system calls**
  - But available on all systems
    - Linux: `sched_setaffinity()`, PLPA (see below) → hwloc
    - Solaris: `processor_bind()`
    - Windows: `SetThreadAffinityMask()`
    - ...

- **Support for “semi-automatic” pinning in some compilers/environments**
  - Intel compilers > V9.1 (**KMP_AFFINITY** environment variable)
  - PGI, Pathscale, GNU
  - SGI Altix `dplace` (works with logical CPU numbers!)
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)

- **Affinity awareness in MPI libraries**
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - ...

Example for program-controlled affinity: Using PLPA under Linux!

**SKIPPED**
Likwid-pin
Overview

- Inspired by and based on ptoverride (Michael Meier, RRZE) and taskset
- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Allows user to specify skip mask (shepherd threads should not be pinned)
- Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!
- Can also be used as a superior replacement for taskset
- Supports logical core numbering within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system
- Configurable colored output
- Usage examples:
  - likwid-pin -t intel -c 0,2,4-6 ./myApp parameters
  - likwid-pin -s 3 -c S0:0-3 ./myApp parameters
Running the STREAM benchmark with likwid-pin:

```bash
$ export OMP_NUM_THREADS=4
$ likwid-pin -s 0x1 -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK

Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word

[… some STREAM output omitted …]
The *best* time for each test is used
*EXCLUDING* the first and last iterations

[pthread wrapper] PIN_MASK: 0->1 1->4 2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
threadid 1086404928 -> core 5 - OK
[… rest of STREAM output omitted …]

Main PID always pinned
Skip shepherd thread
Pin all spawned threads in turn
Likwid-pin
Using logical core numbering

- Core numbering may vary from system to system even with identical hardware
  - Likwid-topology delivers this information, which can then be fed into likwid-pin
- Alternatively, likwid-pin can abstract this variation and provide a purely logical numbering (physical cores first)

- Across all cores in the node:
  \texttt{OMP\_NUM\_THREADS=8 likwid-pin -c N:0-7 ./a.out}
- Across the cores in each socket and across sockets in each node:
  \texttt{OMP\_NUM\_THREADS=8 likwid-pin -c S0:0-3@S1:0-3 ./a.out}
Likwid-pin

Using logical core numbering

- Possible unit prefixes
  - **N** node
  - **S** socket
  - **M** NUMA domain
  - **C** outer level cache group

Default if –c is not specified!
Likwid-pin

Using logical core numbering

- ... and: Logical numbering inside a pre-existing cpuset:

- `OMP_NUM_THREADS=4` likwid-pin -c L:0-3 ./a.out
aprun on Cray

- **See Cray workshop slides 28ff**

- **aprun supports only physical core numbering**
  - This is OK since the cores are always numbered consecutively on Crays
  - Use `-ss` switch to restrict allocation to local NUMA domain (see later for more on ccNUMA)
  - Use `-d $OMP_NUM_THREADS` or similar for MPI+OMP hybrid code

- **See later on how using multiple cores per module/chip/socket affects performance**
General remarks on the performance properties of multicore multisocket systems
Parallelism in modern computer systems

- Parallel and shared resources within a shared-memory node

Parallel resources:
- Execution/SIMD units
- Cores
- Inner cache levels
- Sockets / memory domains
- Multiple accelerators

Shared resources:
- Outer cache level per socket
- Memory bus per socket
- Intersocket link
- PCIe bus(es)
- Other I/O resources

How does your application react to all of those details?
The parallel vector triad benchmark
(Near-)Optimal code on Cray x86 machines

call get_walltime(S)
!$OMP parallel private(j)
do j=1,R
  if(N.ge.CACHE_LIMIT) then
    !DIR$ LOOP_INFO cache_nt(A)
    !$OMP parallel do
      do i=1,N
        A(i) = B(i) + C(i) * D(i)
      enddo
    !$OMP end parallel do
  else
    !DIR$ LOOP_INFO cache(A)
    !$OMP parallel do
      do i=1,N
        A(i) = B(i) + C(i) * D(i)
      enddo
    !$OMP end parallel do
  endif
!$OMP end parallel
enddo
!$OMP end parallel

call get_walltime(E)

“outer parallel”: Avoid thread team restart at every workshared loop

Large-N version (nontemporal stores)

Small-N version (standard stores)
The parallel vector triad benchmark

Single thread on Cray XE6 Interlagos node

![Graph showing performance metrics](image)

- **OMP overhead and/or lower optimization w/ OpenMP active**
- **Team restart is expensive!**

→ use only outer parallel from now on!
The parallel vector triad benchmark

*Intra-chip scaling on Cray XE6 Interlagos node*

- **Per-module L2 caches**
- **L2 bottleneck**
- **Aggregate L2, exclusive L3**
- **Sync overhead**
- **Memory BW saturated @ 4 threads**
The parallel vector triad benchmark

Nontemporal stores on Cray XE6 Interlagos node

NT stores hazardous if data in cache

25% speedup for vector triad in memory via NT stores
The parallel vector triad benchmark

*Topology dependence on Cray XE6 Interlagos node*

- sync overhead nearly topology-independent @ constant thread count
- more aggregate L3 with more chips
- bandwidth scalability across memory interfaces
The parallel vector triad benchmark

*Inter-chip scaling on Cray XE6 Interlagos node*

- **Sync overhead grows with core/chip count**
- **Bandwidth scalability across memory interfaces**
Bandwidth saturation effects in cache and memory

Low-level benchmark results
Bandwidth limitations: Memory
Some problems get even worse….

- System balance = PeakBandwidth [MByte/s] / PeakFlops [MFlop/s]
  Typical balance ~ 0.25 Byte / Flop → 4 Flop/Byte → 32 Flop/double

Balance values:
Scalar product: 1 Flop/double
→ 1/32 Peak

Dense Matrix·Vector: 2 Flop/double
→ 1/16 Peak

Large MatrixMatrix (BLAS3)
Bandwidth limitations: Main Memory

Scalability of shared data paths inside a NUMA domain (V-Triad)

- Main Memory
- Scalability of shared data paths inside a NUMA domain (V-Triad)

Saturation with 3 threads
Saturation with 2 threads
Saturation with 4 threads

1 thread cannot saturate bandwidth

1 NUMA domain 2 NUMA domains
Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache (V-Triad)
OpenMP performance issues on multicore

Synchronization (barrier) overhead

Work distribution overhead
Welcome to the multi-/many-core era
Synchronization of threads may be expensive!

!$OMP PARALLEL ...
...
!$OMP BARRIER
!$OMP DO
...
!$OMP ENDDO
!$OMP END PARALLEL

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

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

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 Interlagos

*Barrier overhead in CPU cycles*

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Threads</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L2</td>
<td>258</td>
<td>3995</td>
<td>1503</td>
<td>128623</td>
</tr>
<tr>
<td>Shared L3</td>
<td>698</td>
<td>2853</td>
<td>1076</td>
<td>128611</td>
</tr>
<tr>
<td>Same socket</td>
<td>879</td>
<td>2785</td>
<td>1297</td>
<td>128695</td>
</tr>
<tr>
<td>Other socket</td>
<td>940</td>
<td>2740 / 4222</td>
<td>1284 / 1325</td>
<td>128718</td>
</tr>
</tbody>
</table>

Sad face: Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler happy face

<table>
<thead>
<tr>
<th></th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full domain</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L3</td>
<td>2272</td>
<td>27916</td>
<td>5981</td>
<td>151939</td>
</tr>
<tr>
<td>Socket</td>
<td>3783</td>
<td>49947</td>
<td>7479</td>
<td>163561</td>
</tr>
<tr>
<td>Node</td>
<td>7663</td>
<td>167646</td>
<td>9526</td>
<td>178892</td>
</tr>
</tbody>
</table>
Case study: OpenMP-parallel sparse matrix-vector multiplication

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory
Case study: Sparse matrix-vector multiply

- Important kernel in many applications (matrix diagonalization, solving linear systems)
- Strongly memory-bound for large data sets
  - Streaming, with partially indirect access:

```c
 !$OMP parallel do
do i = 1,Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    c(i) = c(i) + val(j) * b(col_idx(j))
  enddo
endo
endo
!$OMP end parallel do
```

- Usually many spMVMs required to solve a problem

- Following slides: Performance data on one 24-core AMD Magny Cours node
Bandwidth-bound parallel algorithms: Sparse MVM

- Data storage format is crucial for performance properties
  - Most useful general format: Compressed Row Storage (CRS)
  - SpMVM is easily parallelizable in shared and distributed memory

- For large problems, spMVM is inevitably memory-bound
  - Intra-LD saturation effect on modern multicores

- MPI-parallel spMVM is often communication-bound
  - See hybrid part for what we can do about this…
**SpMVM node performance model**

- **Double precision CRS:**
  
  ```
  do i = 1, N_r
      do j = row_ptr(i), row_ptr(i+1) - 1
          C(i) = C(i) + val(j) * B(col_idx(j))
      enddo
  enddo
  ```

- **DP CRS code balance**
  
  - $\kappa$ quantifies extra traffic for loading RHS more than once
  - Predicted Performance = $\frac{\text{streamBW}}{B_{\text{CRS}}}$
  - Determine $\kappa$ by measuring performance and actual memory BW
  - $\rightarrow$ Even though the model has a “fudge factor” it is still useful!

\[ B_{\text{CRS}} = \left( \frac{12 + 24/N_{\text{nzr}} + \kappa}{2} \right) \frac{\text{bytes}}{\text{flop}} \]

\[ = \left( \frac{6 + 12/N_{\text{nzr}} + \kappa/2}{2} \right) \frac{\text{bytes}}{\text{flop}}. \]

---

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 1: Large matrix**

Intrasocket bandwidth bottleneck

Good scaling across sockets
Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

Case 2: Medium size

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache

mc2depi, 525825x525825, non-zero: 2100225

 CRS-magnycours
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- Case 3: Small size

No bandwidth bottleneck

Parallelization overhead dominates
Probing performance behavior

likwid-perfctr
Basic approach to performance analysis

1. Runtime profile / Call graph (gprof)
2. Instrument parts which consume significant part of runtime
3. Find performance signatures

Possible signatures:
- Bandwidth saturation
- Instruction throughput limited (real or language induced)
- Latency bound (irregular data access, high branch ratio)
- Load inbalance
- ccNUMA issues
- Pathologic cases (false cacheline sharing, expensive operations)
Probing performance behavior

- **How do we find out about the performance properties and requirements of a parallel code?**
  - Profiling via advanced tools is often overkill

- **A coarse overview is often sufficient**
  - likwid-perfctr (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - Simple end-to-end measurement of hardware performance metrics
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with likwid-perfctr -a

- **Metrics:**
  - BRANCH: Branch prediction miss rate/ratio
  - CACHE: Data cache miss rate/ratio
  - CLOCK: Clock of cores
  - DATA: Load to store ratio
  - FLOPS_DP: Double Precision MFlops/s
  - FLOPS_SP: Single Precision MFlops/s
  - FLOPS_X87: X87 MFlops/s
  - L2: L2 cache bandwidth in MBytes/s
  - L2CACHE: L2 cache miss rate/ratio
  - L3: L3 cache bandwidth in MBytes/s
  - L3CACHE: L3 cache miss rate/ratio
  - MEM: Main memory bandwidth in MBytes/s
  - TLB: TLB miss rate/ratio
Example usage with preconfigured metric group

```bash
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe
```

CPU type: Intel Core Lynnfield processor
CPU clock: 2.93 GHz

Measuring group FLOPS_DP

YOUR PROGRAM OUTPUT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>1.97463e+08</td>
<td>2.31001e+08</td>
<td>2.30963e+08</td>
<td>2.31885e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>9.56999e+08</td>
<td>9.58401e+08</td>
<td>9.58637e+08</td>
<td>9.57338e+08</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED</td>
<td>4.00294e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR</td>
<td>882</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION</td>
<td>4.00303e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.326242</td>
<td>0.32672</td>
<td>0.326801</td>
<td>0.326358</td>
</tr>
<tr>
<td>CPI</td>
<td>4.84647</td>
<td>4.14891</td>
<td>4.15061</td>
<td>4.12849</td>
</tr>
<tr>
<td>DP MFlops/s (DP assumed)</td>
<td>245.399</td>
<td>189.108</td>
<td>189.024</td>
<td>189.304</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>122.698</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.00270351</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP MUOPS/s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DP MUOPS/s</td>
<td>122.701</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
</tbody>
</table>

Always measured

Configured metrics (this group)

Derived metrics
Best practices for runtime counter analysis

Things to look at (in roughly this order)

- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Shared cache BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions, branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are overrated
  - If I really know my code, I can often calculate the misses
  - Runtime and resource utilization is much more important
- Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes….
- Floating Point Operations Executed is often a better indicator
- Waiting / “Spinning” in barrier generates a high instruction count

```c
!$OMP PARALLEL DO
DO I = 1, N
    DO J = 1, I
        x(I) = x(I) + A(J,I) * y(J)
    ENDDO
ENDDO
!$OMP END PARALLEL DO
```
likwid-perfctr
... and load-balanced codes

env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out

Higher CPI but better performance

!$OMP PARALLEL DO
DO I = 1, N
   DO J = 1, N
      x(I) = x(I) + A(J,I) * y(J)
   ENDDO
ENDDO
!$OMP END PARALLEL DO
Detecting latency-bound codes
… often with graph and tree data structures

<table>
<thead>
<tr>
<th>Metric</th>
<th>Red-Black tree</th>
<th>Optimized data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions retired</td>
<td>1.34268e+11</td>
<td>1.28581e+11</td>
</tr>
<tr>
<td>CPI</td>
<td>9.0176</td>
<td>0.71887</td>
</tr>
<tr>
<td>L3-MEM data volume [GB]</td>
<td>301</td>
<td>3.22</td>
</tr>
<tr>
<td>TLB misses</td>
<td>3.71447e+09</td>
<td>4077</td>
</tr>
<tr>
<td>Branch rate</td>
<td>36%</td>
<td>8.5%</td>
</tr>
<tr>
<td>Branch mispredicted ratio</td>
<td>7.8%</td>
<td>0.00000013%</td>
</tr>
<tr>
<td>Memory bandwidth [GB/s]</td>
<td>10.5</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Useful likwid-perfctr groups: L3, L3CACHE, MEM, TLB, BRANCH

High CPI, near perfect scaling if using SMT threads (Intel).
Note: Latency bound code can still produce significant aggregated bandwidth.
Programming language induced problems

- The **object-oriented programming** paradigm implements functionality resulting in many calls to small functions
- The ability of the compiler to inline functions *(and still generate the best possible machine code)* is limited

**Symptoms:**
- Low ("good") CPI
- Low resource utilization (Flops/s, bandwidth)
- Orders of magnitude more general purpose than arithmetic floating point instructions
- High branch rate

**Solution:**
- Use **basic data types and plain arrays** in compute intensive loops
- Use **plain C-like code**
- Keep things simple – do not obstruct the compiler’s view on the code
Microarchitectural features of modern processors

Hardware-software interaction
SIMD parallelism
A closer look at the cache hierarchy
Performance modeling on the microarchitecture level
Where do we come from?
Stored program design

Flexible, but optimization is hard!

Architect’s view:
Make the common case fast!

Instruction Level Parallelism
Pipelining
Superscalar execution

Data Access Locality
Memory Hierarchy

Hardware Prefetcher

Data Parallelism
SIMD execution

MIMD Parallelism
SMT
Multicore
Multisocket
Cluster

EDSAC 1949
ENIAC 1948

Cray XE6 Workshop
Performance for Multicore
First Assumption: ILP

**Assumption:** Every sequential instruction stream implies potential parallelism on instruction level (ILP)

Techniques to exploit assumption:
- Pipelining (Overlap the execution of instructions)
- Superscalar design (more than 1 ALU)
- Out of order (OoO) execution

Problems:
- Makes hardware implementation complex
- Benefit is often not worth the effort
- Real-world benefit is limited (3-6 ops/cycle, 1 or less on average)
CPI: A Measure for ILP

CPI: Cycles per Instruction

Ideal CPI for pipelined (non-superscalar) processor: 1
CPI for superscalar processor: < 1

Connection to Runtime:

\[ \text{time} = \text{cycles} \times \text{clock rate} \]

Cycles can be calculated as:

\[ \text{cycles} = \text{CPI} \times \text{number of instructions} \]
Second Assumption: Locality of Data Access

**Assumption:** If a data item is loaded it is likely that it is loaded again in the near future (*temporal locality*). If a data item is loaded it is likely that a data item in close distance is also loaded (*spatial locality*).

**Techniques to exploit assumption:**
- Use **caches** to make repeated data accesses faster
- Use **cache lines** to reduce latency impact

**Problems:**
- Does not work for **unstructured data accesses**
- There are many **algorithms with no or weak locality**
Hardware-Software Co-Design?  
*From algorithm to execution*

The machine view:

ISA (Machine code)

Reality:

Algorithm

Programming language

Compiler tries

Hardware = Black Box
How to achieve Performance
(for data intensive floating point codes on commodity chips)

<table>
<thead>
<tr>
<th>Year</th>
<th>Technology</th>
<th>Performance Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>Caches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIMD</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>Prefetching</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUMA</td>
<td></td>
</tr>
<tr>
<td>2003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2005</td>
<td>Multicore</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Explicit Performance Factor**
  - Thread level parallelism: 4-40x
  - SIMD: DP 2-4x, SP 4-8x
  - Distributed memory parallelism: unlimited, 1000x

- **Implicit Performance Factor**
  - Instruction level parallelism: Pipelining 3-4x, Superscalar 2x, SMT 30%
  - Caches: 4-6x
  - NUMA: 2-4x

**Node Performance**: 1TFlops/s, 50-100 GB/s memory bandwidth
IA-32 Architecture Basics: Floating Point Operations and SIMD

- “Sensible SIMD” came with SSE (Pentium III) and SSE2 (Pentium 4) – Streaming SIMD Extensions
- With AVX a new SIMD instruction set with 256 bit register width was introduced
- AVX will be the relevant instruction set for the near future
- An extension to 512 bit register width is already in planning

- Each register can be partitioned into several integer or FP data types
  - 8 to 128-bit integers
  - single or double precision floating point
- SIMD instructions can operate on the lowest or all partitions of a register at once
- Possible data types in an SIMD register

![Diagram showing possible data types in an SIMD register]

- SSE and AVX-128 types:
  - 4x float
  - 2x double
  - 16x byte
  - 8x 16-bit word
  - 4x 32-bit doubleword
  - 2x 64-bit quadword
  - 1x 128-bit doublequadword

- AVX-256 types:
  - 8x float
  - 4x double
IA-32 Architecture Basics: Floating Point Operations and SIMD

- **Example:** Single precision FP packed vector addition

  ![SIMD Mode vs Scalar Mode Diagram](image)

- **Multiple operations are done in one single instruction**
- **Nehalem:** 1-cycle throughput for double precision SSE2 MULT & ADD leading to a **peak performance of 4 (DP) FLOPs/cycle**
- **Sandy Bridge & Interlagos:** Peak performance of **8 (DP) FLOPs/cycle**
  - **Interlagos:** Only achievable with FMA instruction
Everything on a processor happens in terms of cycles!

All efforts are focused on increasing the average instruction throughput:
Metric CPI (cycles per instruction)

Important for us:
• Arithmetic instruction throughput
• Load/Store instruction throughput
• Overall instruction throughput

Runtime Contributions:
1. Instruction execution
2. Data transfers
   • Cache transfers
   • Memory transfers
AMD Interlagos
Microarchitecture of Compute Unit (CU)

FP units 128bit wide

FP Co-processor supports: AVX and FMA4

CU can sustain two 128bit and one 128 bit store
AMD Interlagos

Node topology

Provide competitive node memory bandwidth for the price of a higher node complexity.

Target cache (i.e., the level that gets filled from memory) is the L2 cache.

Visible L3 cache size is 6 MB per chip (12 MB per socket).
### Comparison chart

#### SIMD instruction throughput (instr/cycle)

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>SandyBridge</th>
<th>Westmere</th>
<th>MagnyCours</th>
<th>Interlagos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add SSE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Mul SSE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Mul/Add SSE</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Load SSE</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Store SSE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2</td>
<td>2</td>
<td>2 ?</td>
<td>2</td>
</tr>
<tr>
<td>Add AVX</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Mul AVX</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Mul/Add AVX</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>1 (FMA 2)</td>
</tr>
<tr>
<td>Load AVX</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Store AVX</td>
<td>0.5</td>
<td>-</td>
<td>-</td>
<td>0.5</td>
</tr>
<tr>
<td>Load/Store AVX</td>
<td>0.5</td>
<td>-</td>
<td>-</td>
<td>0.5</td>
</tr>
<tr>
<td>Max Overall</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td><strong>Comparison chart</strong></td>
<td><strong>Memory Hierarchies</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intel SandyBridge EP</strong></td>
<td><strong>AMD Magny Cours</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 cores, 8 FP Units</td>
<td>6 cores, 6 FP Units</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| L1D: 32kB, 8-way, write back |
| L2: 256kB, 8-way, inclusive |
| L3: 20MB, 20-way, inclusive, shared 8C |
| **Memory:** 4-channel DDR3-1600 |
| Aggregated 40MB node cache size. |

<table>
<thead>
<tr>
<th><strong>AMD Interlagos</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>8 cores, 4 FP Units</td>
</tr>
</tbody>
</table>

| L1D: 64kB, 2-way, write back |
| L2: 512kB, 16-way, exclusive |
| L3: 5 MB, 32-way, exclusive, shared 6C |
| **Memory:** 2-channel DDR3-1333 |

| L1D: 16kB, 4-way, **write through** |
| L2: 2MB, 16-way, inclusive, shared 2C |
| L3: 6 MB, 48-way, exclusive, shared 8C |
| **Memory:** 2-channel DDR3-1866 |

Aggregated 56MB node cache size.
Exclusive cache means that there is only one copy of a cache line in the cache hierarchy! Often called **victim cache**

**Motivation:** Visible cache size for application is larger

**BUT:** More cache traffic necessary

The aggregated L3 bandwidth is low

For HPC applications the L3 cache is not attractive

**Stream benchmark:**

**L3:** IL 40 GB/s, SNB 193 GB/s

5MB (fits in aggr. L2):
IL 108 GB/s, SNB 215GB/s
Interlagos design feature
*Write through L1 cache*

<table>
<thead>
<tr>
<th>Cycles/CL</th>
<th>load</th>
<th>store</th>
<th>copy</th>
<th>stream triad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores/ CU</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>L1</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>20</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>L2</td>
<td>5.43</td>
<td>5.83</td>
<td>11.21</td>
<td>22.21</td>
</tr>
<tr>
<td></td>
<td>13.47</td>
<td>25.21</td>
<td>17.63</td>
<td>30.40</td>
</tr>
<tr>
<td>L2 (prefetch)</td>
<td>3.64</td>
<td>5.72</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>12.92</td>
<td>25.53</td>
<td>16.22</td>
<td>30.21</td>
</tr>
</tbody>
</table>

**Consequences:**
- Stores involve a large penalty
- L2 cache store bandwidth does not scale
- Prefetching to L1 only pays off with one core

**Write through motivation:**
- Simpler to implement (cache coherence)
- Can save overhead for shared L2 access
- No write allocate
- But higher cost for stores in L1 cache

Try to avoid stores as far as possible! ☺️
Reading x86 assembly code
Introduction to Assembly

To read or write assembly code you have to know about:

- Instruction Set Architecture (ISA)
- Application Binary Interface (ABI)
- Object Code Format (ELF on Linux)
- Assembler specific directives (gas, masm)

Useful tools:

- GNU binutils (objdump, readelf)
- Debugger (gdb)
- Compiler option –S (Intel/GCC)
Basic approach to check the instruction code

- Get the assembler code (Intel compiler):
  \texttt{icc -S -O3 -xHost triad.c -o triad.s}

- Disassemble Executable:
  \texttt{objdump -d ./cacheBench | less}

- Things to check for:
  - Is the code vectorized? Search for pd/ps suffix.
    \texttt{mulpd, addpd, vaddpd, vmulpd}
  - Is the data loaded with 16 byte moves?
    \texttt{movapd, movaps, vmovupd}
  - For memory-bound code: Search for nontemporal stores:
    \texttt{movntpd, movntps}

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B
Basics of the x86-64 ISA

- Instructions have 0 to 2 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: A[i] equivalent to *(A+i) (a pointer has a type: A+i*8)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps %xmm3, 0x30(%rdi,%rax,8)
addq $8, %rax
js ..B1.4
```

```
401b9f: 0f 29 5c c7 30    movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08    add $0x8,%rax
401ba8: 78 a6    js 401b50 <triad_asm+0x4b>
```
Basics of the x86-64 ISA II

16 general Purpose Registers (64bit):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp

Floating Point SIMD Registers:
xmm0–xmm15  SSE (128bit)  alias with 256bit registers
ymm0–ymm15  AVX (256bit)

SIMD instructions are distinguished by:
AVX (VEX) prefix: v
Operation: mul, add, mov
Modifier: non temporal (nt), unaligned (u), aligned (a), high (h)
Data type: single (s), double (d)
Basics of x86-64 ABI

- Regulations how functions are called on binary level
- Differs between 32 bit / 64 bit and Operating Systems

x86-64 on Linux:

- Integer or address parameters are passed in the order: 
  rdi, rsi, rdx, rcx, r8, r9

- Floating Point parameters are passed in the order xmm0–xmm7

- Registers which must be preserved across function calls: 
  rbx, rbp, r12–r15

- Return values are passed in rax/rdx and xmm0/xmm1
Case Study: summation

```c
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

Instruction code:

<table>
<thead>
<tr>
<th>Instruction address</th>
<th>Opcodes</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>401d08:</td>
<td>f3 0f 58 04 82</td>
<td>addss (%rdx,%rax,4),%xmm0</td>
</tr>
<tr>
<td>401d0d:</td>
<td>48 83 c0 01</td>
<td>add $0x1,%rax</td>
</tr>
<tr>
<td>401d11:</td>
<td>39 c7</td>
<td>cmp %eax,%edi</td>
</tr>
<tr>
<td>401d13:</td>
<td>77 f3</td>
<td>ja 401d08</td>
</tr>
</tbody>
</table>

To get code use `objdump -d` on object file or executable.
How to leverage SIMD

- The compiler does it for you (aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available. To enable instruction set often additional flags are necessary:

- xmmmintrin.h (SSE)
- pmmintrin.h (SSE2)
- immintrin.h (AVX)
- x86intrin.h (all instruction set extensions)
Case Study: summation using intrinsics

```c
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = __mm_setzero_ps();
sum1 = __mm_setzero_ps();
sum2 = __mm_setzero_ps();
sum3 = __mm_setzero_ps();

for (int j=0; j<size; j+=16){
    t0 = __mm_loadu_ps(data+j);
    t1 = __mm_loadu_ps(data+j+4);
    t2 = __mm_loadu_ps(data+j+8);
    t3 = __mm_loadu_ps(data+j+12);
    sum0 = __mm_add_ps(sum0, t0);
    sum1 = __mm_add_ps(sum1, t1);
    sum2 = __mm_add_ps(sum2, t2);
    sum3 = __mm_add_ps(sum3, t3);
}

sum0 = __mm_add_ps(sum0, sum1);
sum0 = __mm_add_ps(sum0, sum2);
sum0 = __mm_add_ps(sum0, sum3);
sum0 = __mm_hadd_ps(sum0, sum0);
sum0 = __mm_hadd_ps(sum0, sum0);
_mm_store_ss(&scalar_sum, sum0);
```
### Case Study: summation, instruction code

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0f 57 c9</td>
<td>xorps %xmm1,%xmm1</td>
</tr>
<tr>
<td>17</td>
<td>31 c0</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>19</td>
<td>0f 28 d1</td>
<td>movaps %xmm1,%xmm2</td>
</tr>
<tr>
<td>1c</td>
<td>0f 28 c1</td>
<td>movaps %xmm1,%xmm0</td>
</tr>
<tr>
<td>1f</td>
<td>0f 28 d9</td>
<td>movaps %xmm1,%xmm3</td>
</tr>
<tr>
<td>22</td>
<td>66 0f 1f 44 00 00</td>
<td>nopw 0x0(%rax,%rax,1)</td>
</tr>
<tr>
<td>28</td>
<td>0f 10 3e</td>
<td>movups (%rsi),%xmm7</td>
</tr>
<tr>
<td>2b</td>
<td>0f 10 76 10</td>
<td>movups 0x10(%rsi),%xmm6</td>
</tr>
<tr>
<td>2f</td>
<td>0f 10 6e 20</td>
<td>movups 0x20(%rsi),%xmm5</td>
</tr>
<tr>
<td>33</td>
<td>0f 10 66 30</td>
<td>movups 0x30(%rsi),%xmm4</td>
</tr>
<tr>
<td>37</td>
<td>83 c0 10</td>
<td>add $0x10,%eax</td>
</tr>
<tr>
<td>3a</td>
<td>48 83 c6 40</td>
<td>add $0x40,%rsi</td>
</tr>
<tr>
<td>3e</td>
<td>0f 58 df</td>
<td>addps %xmm7,%xmm3</td>
</tr>
<tr>
<td>41</td>
<td>0f 58 c6</td>
<td>addps %xmm6,%xmm0</td>
</tr>
<tr>
<td>44</td>
<td>0f 58 d5</td>
<td>addps %xmm5,%xmm2</td>
</tr>
<tr>
<td>47</td>
<td>0f 58 cc</td>
<td>addps %xmm4,%xmm1</td>
</tr>
<tr>
<td>4a</td>
<td>39 c7</td>
<td>cmp %eax,%edi</td>
</tr>
<tr>
<td>4c</td>
<td>77 da</td>
<td>ja 28 &lt;compute_sum_SSE+0x18&gt;</td>
</tr>
<tr>
<td>4e</td>
<td>0f 58 c3</td>
<td>addps %xmm3,%xmm0</td>
</tr>
<tr>
<td>51</td>
<td>0f 58 c2</td>
<td>addps %xmm2,%xmm0</td>
</tr>
<tr>
<td>54</td>
<td>0f 58 c1</td>
<td>addps %xmm1,%xmm0</td>
</tr>
<tr>
<td>57</td>
<td>f2 0f 7c c0</td>
<td>haddps %xmm0,%xmm0</td>
</tr>
<tr>
<td>5b</td>
<td>f2 0f 7c c0</td>
<td>haddps %xmm0,%xmm0</td>
</tr>
<tr>
<td>5f</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

#### Loop body
Improving Memory Performance
Streaming Stores on Interlagos

```c
#pragma vector aligned
#pragma vector always
#pragma vector nontemporal
for (i=0; i< size; i++){
    A[i] = B[i] + alpha* C[i];
}
```

617 GFlop/s vs. 854 GFlop/s

```c
..B1.4:
movaps (%rdx,%rax,8),%xmm1
mulpd %xmm0, %xmm4
addpd (%rsi,%rax,8),%xmm1
movntpd %xmm1, (%rdi,%rax,8)
addq 1,%rax
cmpq %rcx, %rax
js ..B1.4
```

On Interlagos NT stores circumvent both write-through stores and the L3 cache. This makes them even attractive for smaller data sets which could fit into L3 cache. triad (3MB): 783 Gflop/s, NT 1156 Gflop/s
Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes
First touch placement policy
C++ issues
ccNUMA locality and dynamic scheduling
ccNUMA locality beyond first touch
ccNUMA performance problems
“The other affinity” to care about

- **ccNUMA:**
  - Whole memory is *transparently accessible* by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)

- **How do we make sure that memory access is always as "local" and "distributed" as possible?**

- Page placement is implemented in units of OS pages (often 4kB, possibly more)
Cray XE6 Interlagos node
4 chips, two sockets, 8 threads per ccNUMA domain

- **ccNUMA map:** Bandwidth penalties for remote access
  - Run 8 threads per ccNUMA domain (1 chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM triad benchmark using nontemporal stores
ccNUMA locality tool numactl: How do we enforce some locality of access?

- **numactl** can influence the way a binary maps its memory pages:

  ```
  numactl --membind=<nodes> a.out  # map pages only on <nodes>
  --preferred=<node> a.out  # map pages on <node>
  # and others if <node> is full
  --interleave=<nodes> a.out  # map pages round robin across
  # all <nodes>
  ```

- **Examples:**

  ```
  env OMP_NUM_THREADS=2 numactl --membind=0 -cpunodebind=1 ./stream
  ```

  ```
  env OMP_NUM_THREADS=4 numactl --interleave=0-3 \
  likwid-pin -c N:0,4,8,12 ./stream
  ```

- **But what is the default without numactl?**
ccNUMA default memory locality

- "Golden Rule" of ccNUMA:

  A memory page gets mapped into the local memory of the processor that first touches it!

  - Except if there is not enough local memory available
  - This might be a problem, see later

- **Caveat**: "touch" means "write", not "allocate"

- **Example**:

  ```c
  double *huge = (double*)malloc(N*sizeof(double));
  for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;
  ```

- It is sufficient to touch a single item to map the entire page
Coding for ccNUMA data locality

- Most simple case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

A = 0.d0

 !$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i) = 0.d0
end do
!$OMP end do
!$OMP end parallel
...```

```
integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel do
!$OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

Most simple case: explicit initialization
Coding for ccNUMA data locality

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```fortran
integer,parameter :: N=10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   A(i)=0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O.

- Initialization:

  ```fortran
  integer,parameter :: N=10000000
double precision A(N), B(N)
  !$OMP parallel
  !$OMP do schedule(static)
do i = 1, N
     A(i)=0.d0
  end do
  !$OMP end do
  !$OMP single
  READ(1000) A
  !$OMP end single
  !$OMP do schedule(static)
do i = 1, N
     B(i) = function ( A(i) )
  end do
  !$OMP end parallel do
  ```
Coding for Data Locality

- **Required condition:** OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: *static*! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
    - Guaranteed by OpenMP 3.0 only for loops in the same enclosing parallel region and static schedule
    - In practice, it works with any compiler even across regions
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order

- **How about global objects?**
  - Better not use them
  - If communication vs. computation is favorable, might consider properly placed copies of global data
  - In C++, STL allocators provide an elegant solution (see hidden slides)
Coding for Data Locality:
Placement of static arrays or arrays of objects

- Speaking of C++: Don't forget that constructors tend to touch the data members of an object. Example:

```cpp
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  inline D operator*(const D& o) throw() {
    return D(d*o.d);
  }
...}
```

→ placement problem with

```cpp
D* array = new D[1000000];
```
Coding for Data Locality:
Parallel first touch for arrays of objects

- **Solution:** Provide overloaded `D::operator new[]`

    ```cpp
    void* D::operator new[](size_t n) {
        char *p = new char[n]; // allocate
        size_t i,j;
        #pragma omp parallel for private(j) schedule(...)  
        for(i=0; i<n; i += sizeof(D))
            for(j=0; j<sizeof(D); ++j)
                p[i+j] = 0;
        return p;
    }

    void D::operator delete[](void* p) throw() {
        delete [] static_cast<char*>(p);
    }
    ```

- **Placement of objects is then done automatically by the C++ runtime via “placement new”**
Coding for Data Locality:
NUMA allocator for parallel first touch in std::vector<>
Diagnosing Bad Locality

- If your code is cache-bound, you might not notice any locality problems.

- Otherwise, bad locality **limits scalability at very low CPU numbers** (whenever a node boundary is crossed):
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…

- **Consider using performance counters**
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel Nehalem (Core i7):

    ```
    env OMP_NUM_THREADS=8 likwid-perfctr -g MEM -C N:0-7 ./a.out
    ```
Using performance counters for diagnosing bad ccNUMA access locality

- **Intel Nehalem EP node:**

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
<th>core 5</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>5.20725e+08</td>
<td>5.24793e+08</td>
<td>5.21547e+08</td>
<td>5.23717e+08</td>
<td>5.28269e+08</td>
<td>5.29083e+08</td>
<td>5.30103e+08</td>
<td>5.29479e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>1.90447e+09</td>
<td>1.90599e+09</td>
<td>1.90619e+09</td>
<td>1.90673e+09</td>
<td>1.90583e+09</td>
<td>1.90746e+09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNC_QMC_NORMAL_READS_ANY</td>
<td>8.17606e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.07797e+07</td>
<td>0</td>
</tr>
<tr>
<td>UNC_QMC_WRITES_FULL_ANY</td>
<td>5.53837e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.51052e+07</td>
<td>0</td>
</tr>
<tr>
<td>UNC_QHL_REQUESTS_REMOTE_READS</td>
<td>6.84504e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.8107e+07</td>
<td>0</td>
</tr>
<tr>
<td>UNC_QHL_REQUESTS_LOCAL_READS</td>
<td>6.82751e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.76274e+07</td>
<td>0</td>
</tr>
</tbody>
</table>

  RDTSC timing: 0.827196 s

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
<th>core 5</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.714167</td>
<td>0.714733</td>
<td>0.71481</td>
<td>0.715013</td>
<td>0.714673</td>
<td>0.715286</td>
<td>0.71486</td>
<td>0.71515</td>
</tr>
<tr>
<td>Memory bandwidth [MBytes/s]</td>
<td>10610.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10513.4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Remote Read BW [MBytes/s]</td>
<td>5296</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5269.43</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Uncore events only counted once per socket

Half of read BW comes from other socket!
If all fails…

- Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?
  - Program has erratic access patterns → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```bash
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
top - 14:18:25 up 92 days, 6:07, 2 users, load average: 0.00, 0.02, 0.00
Mem: 4065564k total, 1149400k used, 2716164k free, 43388k buffers
Swap: 2104504k total, 2656k used, 2101848k free, 1038412k cached
```
ccNUMA problems beyond first touch:  
Buffer cache

- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - “sync” is not sufficient to drop buffer cache blocks

- **Remedies**
  - Drop FS cache pages after user job has run (admin’s job)
    - seems to be automatic after aprun has finished on Crays
  - User can run “sweeper” code that allocates and touches all physical memory before starting the real application
  - **numactl** tool or **aprun** can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels
ccNUMA problems beyond first touch: 
*Buffer cache*

**Real-world example: ccNUMA and the Linux buffer cache**

**Benchmark:**

1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory available in LD0

**Result:** By default, Buffer cache is given priority over local page placement

→ restrict to local domain if possible!
ccNUMA placement and erratic access patterns

- Sometimes access patterns are just not nicely grouped into contiguous chunks:

```fortran
double precision :: r, a(M)
 !$OMP parallel do private(r)
do i=1,N
  call RANDOM_NUMBER(r)
  ind = int(r * M) + 1
  res(i) = res(i) + a(ind)
enddo
!$OMP end parallel do
```

- Or you have to use tasking/dynamic scheduling:

```fortran
!$OMP parallel
!$OMP single
do i=1,N
  call RANDOM_NUMBER(r)
  if(r.le.0.5d0) then
    !$OMP task
    call do_work_with(p(i))
    !$OMP end task
  endif
enddo
!$OMP end single
!$OMP end parallel
```

- In both cases page placement cannot easily be fixed for perfect parallel access
ccNUMA placement and erratic access patterns

- **Worth a try:** Interleave memory across ccNUMA domains to get at least some parallel access
  1. **Explicit placement:**
     ```
     !$OMP parallel do schedule(static,512)
     do i=1,M
       a(i) = ...
     enddo
     !$OMP end parallel do
     ```
  2. **Using global control via `numactl`:**
     ```
     numactl --interleave=0-3 ./a.out
     ```

- **Fine-grained program-controlled placement via `libnuma` (Linux) using, e.g., `numa_alloc_interleaved_subset()`**, `numa_alloc_interleaved()` **and others**

  Observe page alignment of array to get proper placement!

  This is for all memory, not just the problematic arrays!
The curse and blessing of interleaved placement: OpenMP STREAM on a Cray XE6 Interlagos node

- **Parallel init**: Correct parallel initialization
- **LD0**: Force data into LD0 via `numactl -m 0`
- **Interleaved**: `numactl --interleave <LD range>`

![Graph showing performance comparison between Parallel placement, LD0 placement, and Interleaved placement for different numbers of locality domains.](image)
There is no alternative to knowing what is going on between your code and the hardware

Without performance modeling, optimizing code is like stumbling in the dark

Performance x Flexibility = constant

a.k.a. Abstraction is the natural enemy of performance