Performance-oriented programming on multicore-based systems

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

NEC User Group Meeting
June 12, 2012
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

- A short history of multicore processing
- The LIKWID tool suite for multicore programming

**Data access on modern processors**
- Basic performance benchmarks and properties
- The balance metric: Bandwidth-based performance modeling
- Optimizing data access by code transformations

**Performance properties of parallel code on multicore processors and nodes**
- Exploration by microbenchmarks
- Sparse matrix-vector multiplication

**ccNUMA: Properties and efficient programming**

**Simultaneous Multithreading (SMT)**
- Principles
- Expected benefits
- Mythbusting
Multicore processor and system architecture

Basics
The x86 multicore evolution so far

Intel Single-Dual-/Quad-/Hexa-/Cores (one-socket view)

2005: “Fake” dual-core
- Chipset
- Memory

2006: True dual-core
- Chipset
- Memory

2008: Simultaneous Multi Threading (SMT)
- Chipset
- Memory

2010: 6-core chip
- Chipset
- Memory

2012: Wider SIMD units
- Chipset
- Memory

Nehalem EP
“Core i7”
45nm

Westmere EP
“Core i7”
32nm

Sandy Bridge EP
“Core i7”
32nm
There is no longer a single driving force for chip performance!

Floating Point (FP) Performance:

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

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

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

But: \( P=5 \, \text{GF/s (dp)} \) for serial, non-SIMD code
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 to 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)
- $P_{\text{max}} = (2.6 \times 8 \times 8)$ GF/s
  = 166.4 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
16-core dual socket “Interlagos” node

- **Two 8- (integer-) core chips per socket**
- **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
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
Performance tools for parallel multicore programming
Likwid Lightweight Performance Tools

- Lightweight command line tools for Linux
- Help to face the challenges without getting in the way
- Focus on X86 architecture

Philosophy:
- Simple
- Efficient
- Portable
- Extensible

Open source project (GPL v2):
http://code.google.com/p/likwid/
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

```
$ 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
```
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 Bulldozer/Interlagos (beta)
  - 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**

- **Sockets:** 2
- **Cores per socket:** 16
- **Threads per core:** 1

<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>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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**

- **Level:** 1
- **Size:** 16 kB
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
Graphical:

Socket 0:

Socket 1:

NUG Tutorial 2012 Performance on Multicore
Probing performance behavior

likwid-perfctr
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
  - Operating modes:
    - Wrapper
    - Stethoscope
    - Timeline
    - Marker API
  - Preconfigured and extensible metric groups, list with likwid-perfctr -a

  - 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

$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -t intel -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
likwid-perfctr
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
**likwid-perfctr**

*Identify load imbalance…*

- **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

---

### Event Counts

<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>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>2.10045e+10</td>
<td>1.90983e+10</td>
<td>1.729e+10</td>
<td>1.60898e+10</td>
<td>1.67958e+10</td>
<td>1.84689e+10</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>1.82569e+10</td>
<td>1.81203e+10</td>
<td>1.81802e+10</td>
<td>1.82084e+10</td>
<td>1.82334e+10</td>
<td>1.82484e+10</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>1.60953e+10</td>
<td>1.6438e+10</td>
<td>1.652/46e+10</td>
<td>1.60551e+10</td>
<td>1.65738e+10</td>
<td>1.65894e+10</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED</td>
<td>2.77016e+08</td>
<td>7.83476e+08</td>
<td>1.39355e+09</td>
<td>1.94356e+09</td>
<td>2.38559e+09</td>
<td>2.85981e+09</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR</td>
<td>1.70802e+08</td>
<td>2.64065e+08</td>
<td>2.23153e+08</td>
<td>2.60035e+08</td>
<td>2.30434e+08</td>
<td>2.07293e+08</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</td>
<td>19</td>
<td>0</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>4.47818e+08</td>
<td>1.04754e+09</td>
<td>1.61671e+09</td>
<td>2.30448e+09</td>
<td>2.61102e+09</td>
<td>3.0671e+09</td>
</tr>
</tbody>
</table>

---

### Code Snippet

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

---

NUG Tutorial 2012

Performance on Multicore
likwid-perfctr
… and load-balanced codes

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

Higher CPI but better performance

```fortran
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```
likwid-perfctr

Stethoscope mode

- likwid-perfctr counts events on the specified cores
- No connection to running code / binary / process / thread

- Listen what currently happens without any overhead:

  likwid-perfctr -c N:0-11 -g FLOPS_DP -s 10

- It can be used as cluster/server monitoring tool
- A frequent use is to measure a certain part of a long running parallel application from outside
likwid-perfctr supports time-resolved measurements of full node:
likwid-perfctr –c N:0-11 -g MEM -d 50ms > out.txt
Marker API: **Manually instrument** parts of an application

The API only turns counters on/off. The configuration of the counters is still done by likwid-perfctr application.

Multiple named regions can be measured

Results on multiple calls are accumulated

Inclusive and overlapping Regions are allowed

```c
likwid_markerInit();  // must be called from serial region

likwid_markerStartRegion("Compute");
    ...
likwid_markerStopRegion("Compute");

likwid_markerStartRegion("postprocess");
    ...
likwid_markerStopRegion("postprocess");

likwid_markerClose();  // must be called from serial region
```
Groups are architecture-specific
- They are defined in simple text files
- The code is generated on recompile
- `likwid-perfctr -a` outputs list of groups
- A documentation text is available for every group
To know the performance properties of a machine is essential for any optimization effort

Microbenchmarking is an important method to gain this information

Extensible, flexible benchmarking framework

Rapid development of low level kernels

Already includes many ready to use threaded benchmark kernels

Benchmarking runtime cares for:
  - Thread management and placement
  - Data allocation and NUMA aware initialization
  - Timing and result presentation
likwid-bench
Benchmark example

- Implement micro benchmark in abstract assembly
- Add meta information
- The benchmark file is automatically converted, compiled and added to the benchmark application

$likwid-bench -t clcopy -g 1 -i 1000 -w S0:1MB:2
$likwid-bench -t load -g 2 -i 100 -w S1:1GB -w S0:1GB-0:S1,1:S0

STREAMS 2
TYPE DOUBLE
FLOPS 0
BYTES 16
LOOP 32
movaps FPR1, [STR0 + GPR1 * 8 ]
movaps FPR2, [STR0 + GPR1 * 8 + 64 ]
movaps FPR3, [STR0 + GPR1 * 8 + 128 ]
movaps FPR4, [STR0 + GPR1 * 8 + 192 ]
movaps [STR1 + GPR1 * 8 ], FPR1
movaps [STR1 + GPR1 * 8 + 64 ], FPR2
movaps [STR1 + GPR1 * 8 + 128 ], FPR3
movaps [STR1 + GPR1 * 8 + 192 ], FPR4
likwid-bench
Measure NUMA characteristics 1

- 2 thread group (2x4 threads) on socket 0/1
- Correct NUMA placement (first touch)

likwid-bench -g 2 -i 1000 -t copy -w S0:500MB:4 -w S1:500MB:4

23147 MB/s

14001 MB/s
13989 MB/s
likwid-bench
Measure NUMA characteristics 2

- 2 thread group (2x4 threads) on socket 0/1
- Common problem: memory placed on one socket only

likwid-bench -g 2 -i 1000 -t copy -w S0:500MB:4 -w S1:500MB:4-0:S0,1:S0

Bandwidth less than half

9839 MB/s

4479 MB/s

13402 MB/s
Bandwidth map data measured with likwid-bench. All cores used in one NUMA domain, memory is placed in a different NUMA domain. Test case: simple copy $A(:)=B(:)$, large arrays.
### AMD Magny Cours 4-socket system

**Topology at its best?**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **3.7 GB/s**
- **4.3 GB/s**
- **5.1 GB/s**
- **8.7 GB/s**

- **2.7 GB/s**
- **2.0 GB/s**

Diagram of network topology with data rates and connections.
Measuring energy consumption
likwid-powermeter

- Implements Intel RAPL interface (Sandy Bridge)
- RAPL (Running average power limit)

---

CPU name: Intel Core SandyBridge processor
CPU clock: 3.49 GHz

---

Base clock: 3500.00 MHz
Minimal clock: 1600.00 MHz
Turbo Boost Steps:
C1 3900.00 MHz
C2 3800.00 MHz
C3 3700.00 MHz
C4 3600.00 MHz

---

Thermal Spec Power: 95 Watts
Minimum Power: 20 Watts
Maximum Power: 95 Watts
Maximum Time Window: 0.15625 micro sec

---
### Energy consumption: Example

*A medical image reconstruction code on Sandy Bridge*

#### Sandy Bridge EP (8 cores, 2.7 GHz base freq.)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Runtime</th>
<th>Power</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 cores, plain C</td>
<td>90.43 s</td>
<td>89.69 Watt</td>
<td>8111 Joule</td>
</tr>
<tr>
<td>8 cores, SSE</td>
<td>29.63 s</td>
<td>92.62 Watt</td>
<td>2745 Joule</td>
</tr>
<tr>
<td>8 cores (SMT), SSE</td>
<td>22.61 s</td>
<td>102.07 Watt</td>
<td>2308 Joule</td>
</tr>
<tr>
<td>8 cores (SMT), AVX</td>
<td>18.42 s</td>
<td>110.80 Watt</td>
<td>2041 Joule</td>
</tr>
</tbody>
</table>

#### Sandy Bridge UP-Xeon E31280 (4 cores, 3.5 GHz base freq.)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Runtime</th>
<th>Power</th>
<th>Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 cores, plain C</td>
<td>154.72 s</td>
<td>55.61 Watt</td>
<td>8605 Joule</td>
</tr>
<tr>
<td>4 cores, SSE</td>
<td>49.99 s</td>
<td>58.01 Watt</td>
<td>2900 Joule</td>
</tr>
<tr>
<td>4 cores (SMT), SSE</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4 cores (SMT), AVX</td>
<td>36.73 s</td>
<td>66.43 Watt</td>
<td>2440 Joule</td>
</tr>
</tbody>
</table>
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
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

```c
#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 `dpsole` (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!
Likwid-pin
Overview

- Pins processes and threads to specific cores without touching code
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- 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

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:

$ 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 ...]
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:
  
  OMP_NUM_THREADS=8 likwid-pin -c N:0-7 ./a.out

- Across the cores in each socket and across sockets in each node:

  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!*
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 \( A(\cdot) = C(\cdot) \)

![Diagram of data transfers in a memory hierarchy](image)

- Standard stores: 3 CL transfers
- Nontemporal (NT) stores: 2 CL transfers
- 50% performance boost for COPY
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
enddo
```

- **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!**
$$A(:) = B(:) + C(:) \times D(:)$$ on one Interlagos core

64 GB/s (no write allocate in L1)

L1D cache (16k)

L2 cache (2M)

L3 cache (8M)

Memory

6x bandwidth gap (1 core)

10 GB/s (incl. write allocate)

Is this the limit???
STREAM benchmarks:
Memory bandwidth on AMD Interlagos node

- STREAM is the “standard” for memory BW comparisons
- NT store variants save write allocate on stores → 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): \(B_m = \{17/8 \text{ GW/s}\} / \{4 \times 2.3 \times 8 \text{ GFlop/s}\} \approx 0.029 \text{ W/F}\)
  Intel Sandy Bridge EP (2.7 GHz): \(\approx 0.025 \text{ W/F}\)
  NEC SX9 (vector): \(\approx 0.3 \text{ W/F}\)
  nVIDIA GTX480: \(\approx 0.026 \text{ W/F}\)
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>

$1/B_M = \text{“Computational Intensity”: How many FP ops can be performed before FP performance becomes a bottleneck?}$
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:

\[
B_c = \frac{\text{data transfer (LD/ST) [words]}}{\text{arithmetic operations [flops]}}
\]

- Expected fraction of peak performance ("lightspeed"):
  \[ l = \min \left( 1, \frac{B_m}{B_c} \right) \]
  \[ l = 1 \rightarrow \text{code is not limited by bandwidth} \]

- Lightspeed for absolute performance: ($P_{\text{max}}$ : “applicable” peak performance)
  \[
P = l \cdot P_{\text{max}} = \min \left( P_{\text{max}}, \frac{b_S}{B_c} \right)
\]

- Example: Vector triad $A(:)=B(:)+C(:)*D(:)$ 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:

```plaintext
double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
integer :: t0,t1

t0 = 0 ; t1 = 1

do it = 1,imax ! 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
endo

! swap arrays
i = t0 ; t0=t1 ; t1=i
endo
```

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

- \( \phi(:,:,t0) : 3 \text{ LD} + \)
- \( \phi(:,:,t1) : 1 \text{ ST} + 1\text{LD} \)

\[ B_c = \frac{5}{4} \text{ FLOPs} = 1.25 \frac{W}{F} \]

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

**WRITE ALLOCATE:**
LD + ST \( \phi(i,k,t1) \)
Modern cache subsystems may further reduce memory traffic

If cache is large enough to hold at least 2 rows (shaded region): Each \( \phi(:,:,t_0) \) is loaded once from main memory and reused 3 times from cache:

\[
\phi(:,:,t_0): 1 \text{ LD} + \phi(:,:,t_1): 1 \text{ ST} + 1 \text{LD}
\]

\[B_C = \frac{3}{4} W / F = 0.75 W / F\]

If cache is large enough to hold at least one row \( \phi(:,k-1,t_0) \) needs to be reloaded:

\[
\phi(:,:,t_0): 2 \text{ LD} + \phi(:,:,t_1): 1 \text{ ST} + 1 \text{LD}
\]

\[B_C = \frac{4}{4} W / F = 1.0 W / F\]

Beyond that:

\[
\phi(:,:,t_0): 2 \text{ LD} + \phi(:,:,t_1): 1 \text{ ST} + 1 \text{LD}
\]

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

- Alternative implementation ("Macho FLOP version")

\[
\begin{align*}
\text{do } & k = 1, k_{\text{max}} \\
\text{do } & i = 1, i_{\text{max}} \\
\phi(i, k, t1) &= 0.25 \times \phi(i+1, k, t0) + 0.25 \times \phi(i-1, k, t0) + 0.25 \times \phi(i, k+1, t0) + 0.25 \times \phi(i, k-1, t0)
\end{align*}
\]

- 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

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

3D sweep:

\[
\begin{align*}
\text{do } & k=1,k_{\text{max}} \\
& \text{do } j=1,j_{\text{max}} \\
& \text{do } i=1,i_{\text{max}} \\
\phi(i,j,k,t_{1}) &= \text{oos} \ast (\phi(i-1,j,k,t_{0}) + \phi(i+1,j,k,t_{0}) & \\
& + \phi(i,j-1,k,t_{0}) + \phi(i,j+1,k,t_{0}) & \\
& + \phi(i,j,k-1,t_{0}) + \phi(i,j,k+1,t_{0})) \\
\end{align*}
\]

Best case balance: 1 LD
1 ST + 1 write allocate
6 flops

\[ B_{C} = 0.5 \text{ W/F} \]

If 2-layer condition does not hold but 2 rows fit: \( B_{C} = 5/6 \text{ W/F} \)

Worst case (2 rows do not fit): \( B_{C} = 7/6 \text{ W/F} \)
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
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
```

- \( B_c = 3/1 \) W/F
- \( B_c = 5/2 \) W/F

- Loop fusion
- No optimization potential for either loop

- Fusing different loops allows O(N) data reuse from registers
Data access – general guidelines

- **Case 2: O(N^2)/O(N^2) 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

\[
do \ i=1,N \\
\ do \ j=1,N \\
\ \ \ c(i) = c(i) + a(i,j) \cdot b(j) \\
\ enddo \\
\ enddo
\]

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

- **O(N²)/O(N²) 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

  do i=1,N,2
    do j=1,N
      c(i)=c(i)+a(i,j)*b(j)
    enddo
    do j=1,N
      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 ¾ W/F**
Data access – general guidelines

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

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

  - 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: \( b[\cdot] \) only be loaded once from memory (\( B_c \approx \frac{1}{2} W/F \)) (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
Case study:
3D Jacobi solver

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

- 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
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
Jacobi iteration (2D): Spatial Blocking

- 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
Jacobi iteration (2D): Spatial blocking

**Implementation:**

```fortran
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
```

**Guidelines:**

- Blocking of inner loop levels (traversing continuously through main memory)
- Blocking size `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)

Optimum $j$ block size

24B/update performance model

Middle ($j$) loop blocking

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

Spatial blocking + nontemporal stores

Performance [Mlup/s]

Linear problem size N

T=8
T=8 bs=20
T=8 bs=20 NT stores

16 B/update perf. model
blocking
NT stores expected boost: 50%
General remarks on the performance properties of multicore multisocket systems

Shared vs. scalable resources

Synchronization overhead
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 x86 machines

call get_walltime(S)
!$OMP parallel private(j)
do j=1,R
   if(N.ge.CACHE_LIMIT) then
     !$DEC$ VECTOR NONTEMPORAL
     !$OMP parallel do
       do i=1,N
         A(i) = B(i) + C(i) * D(i)
       enddo
     !$OMP end parallel do
   else
     !$DEC$ VECTOR TEMPORAL
     !$OMP parallel do
       do i=1,N
         A(i) = B(i) + C(i) * D(i)
       enddo
     !$OMP end parallel do
   endif
   ! prevent loop interchange
   if(A(N2).lt.0) call dummy(A,B,C,D)
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 Interlagos node

OMP overhead and/or lower optimization w/ OpenMP active

Team restart is expensive!

⇒ use only outer parallel from now on!

L1 cache
L2 cache
L3 cache
memory

NUG Tutorial 2012
Performance on Multicore
The parallel vector triad benchmark

*Intra-chip scaling on Interlagos node*

- Per-module L2 caches
- L2 bottleneck
- Aggregate L2, exclusive L3
- Memory BW saturated @ 4 threads

**Performance vs Loop length (N)**

- OpenMP T=1
- OpenMP T=2
- OpenMP T=4
- OpenMP T=8

**Performance [MFlops]**
- 0 to 8000

**Loop length (N)**
- $10^1$ to $10^6$
The parallel vector triad benchmark

Nontemporal stores on 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 Interlagos node*

- OpenMP T=8
- OpenMP T=8 S=1 C=2
- OpenMP T=8 S=2 C=2

*more aggregate L3 with more chips*

*sync overhead nearly topology-independent @ constant thread count*

*bandwidth scalability across memory interfaces*
The parallel vector triad benchmark

*Inter-chip scaling on Interlagos node*

sync overhead grows with core/chip count

bandwidth scalability across memory interfaces

![Graph showing performance vs loop length for different OpenMP thread counts (T=8, T=16, T=24, T=32).](image)
Bandwidth saturation effects in cache and memory

Low-level benchmark results
Bandwidth limitations: Main Memory

Scalability of shared data paths inside NUMA domain \((A(:\) = B(:\))\)
Bandwidth limitations: Outer-level cache

Scalability of shared data paths in L3 cache

![Graph showing bandwidth vs. number of cores for different cache designs.]

- Intel SB: New scalable L3 design
- AMD: Optimize for L2 cache!

- 1 NUMA domain
- 2 NUMA domains
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>
### Thread synchronization overhead on Intel CPUs

**pthread vs. OpenMP vs. Spin loop**

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Q9550 (shared L2)</th>
<th>i7 920 (shared L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthreads_barrier_wait</td>
<td>23739</td>
<td>6511</td>
</tr>
<tr>
<td>omp barrier gcc 4.3.3</td>
<td>22603</td>
<td>7333</td>
</tr>
<tr>
<td>omp barrier icc 11.0</td>
<td>399</td>
<td>469</td>
</tr>
<tr>
<td>Spin loop</td>
<td>231</td>
<td>270</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nehalem 2 Threads</th>
<th>Shared SMT threads</th>
<th>shared L3</th>
<th>different socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthreads_barrier_wait</td>
<td>23352</td>
<td>4796</td>
<td>49237</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>2761</td>
<td>479</td>
<td>1206</td>
</tr>
<tr>
<td>Spin loop</td>
<td>17388</td>
<td>267</td>
<td>787</td>
</tr>
</tbody>
</table>

Pthreads → OS kernel call 😞

Spin loop does fine for shared cache sync 😊

OpenMP & Intel compiler 😊
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
!$OMP parallel do
do i = 1,Nr
   do j = row_ptr(i), row_ptr(i+1) - 1
      c(i) = c(i) + val(j) * b(col_idx(j))
   enddo
endo
dono
!$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:**

\[
\begin{align*}
&\text{do } i = 1, N_r \\
&\quad \text{do } j = \text{row_ptr}(i), \text{row_ptr}(i+1) - 1 \\
&\quad \quad C(i) = C(i) + \text{val}(j) \times B(\text{col_idx}(j)) \\
&\end{align*}
\]

**DP CRS code balance**

- \( \kappa \) quantifies extra traffic for loading RHS more than once
- Predicted Performance = \( \text{streamBW}/B_{CRS} \)
- Determine \( \kappa \) by measuring performance and actual memory BW

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

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

Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

- **Case 1: Large matrix**

Intrasocket bandwidth bottleneck

Good scaling across sockets

- CRS-magnycours

cant, 62451x62451, non-zero: 4007383
Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

- **Case 2: Medium size**

  Working set fits in aggregate cache

  Intrasocket bandwidth bottleneck

  mc2depi, 525825x525825, non-zero: 2100225
Application: Sparse matrix-vector multiply
Strong scaling on one Magny-Cours node

- **Case 3: Small size**

No bandwidth bottleneck

Parallelization overhead dominates
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?**
"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));
```

```c
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

$OMP parallel do
   do i = 1, N
      A(i) = 0.d0
   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 parallel do
   do i = 1, N
      B(i) = function ( A(i) )
   end do
$OMP end parallel do
```

NUG Tutorial 2012
Performance on Multicore
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 parallel do
do i = 1, N
    B(i) = function ( A(i) )
end do
!$OMP end parallel do

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

NUG Tutorial 2012 Performance on Multicore 106
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)
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
```
D* array = new D[1000000];
```
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”
template <class T> class NUMA_Allocator {
public:
    T* allocate(size_type numObjects, const void *
localityHint=0) {
        size_type ofs,len = numObjects * sizeof(T);
        void *m = malloc(len);
        char *p = static_cast<char*>(m);
        int i,pages = len >> PAGE_BITS;
        #pragma omp parallel for schedule(static) private(ofs)
        for(i=0; i<pages; ++i) {
            ofs = static_cast<size_t>(i) << PAGE_BITS;
            p[ofs]=0;
        }
        return static_cast<pointer>(m);
    }
};

Application:
vector<double,NUMA_Allocator<double> > x(10000000)
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 \ 
        -t intel ./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>
</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>
</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>
</tr>
<tr>
<td>UNC_QMC_NORMAL_READS_ANY</td>
<td>8.17606e+07</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>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>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>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>10513.4</td>
<td>0</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>5269.43</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

```
# 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)
  - 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 installed in LD0

Result: By default, Buffer cache is given priority over local page placement → restrict to local domain if possible!
Sometimes access patterns are just not nicely grouped into contiguous chunks:

```c
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:

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

     Observe page alignment of array to get proper placement!

  2. Using global control via `numactl`:

     ```
     numactl --interleave=0-3 ./a.out
     ```

     This is for all memory, not just the problematic arrays!

- **Fine-grained program-controlled placement via `libnuma` (Linux) using**, e.g., `numa_alloc_interleaved_subset()`, `numa_alloc_interleaved()` and others
The curse and blessing of interleaved placement: 
OpenMP STREAM on an AMD Interlagos node

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

Principles and performance impact
SMT vs. independent instruction streams
Facts and fiction
SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently

- SMT principle (2-way example):

  ![Diagram showing SMT principle for 2-way example]
SMT impact

- **SMT is primarily suited for increasing processor throughput**
  - With multiple threads/processes running concurrently

- **Scientific codes tend to utilize chip resources quite well**
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- **SMT is an important topology issue**
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.
SMT impact

- SMT adds another layer of topology (inside the physical core)
- Caveat: SMT threads share all caches!
- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread’s executing instructions:

```
Thread 0:
do i=1,N
    a(i) = a(i-1)*c
  enddo

Thread 1:
do i=1,N
    b(i) = func(i)*d
  enddo
```

Dependency → pipeline stalls until previous MULT is over

- Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
    a(i) = a(i-1)*c
    b(i) = func(i)*d
  enddo
```
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
MULT Pipeline depth: 5 stages \(\rightarrow\) 1 F / 5 cycles for recursive update

Fill bubbles via:
- SMT
- Multiple streams

\[
\text{Thread 0:} \quad \begin{array}{l}
\text{do } i=1,N \\
A(i)=A(i-1)*c \\
B(i)=B(i-1)*d \\
\text{enddo}
\end{array}
\]

\[
\text{Thread 1:} \quad \begin{array}{l}
\text{do } i=1,N \\
A(i)=A(i-1)*c \\
B(i)=B(i-1)*d \\
\text{enddo}
\end{array}
\]
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update

Thread 0:
\[
\text{do } i=1,N \\
A(i)=A(i-1)*s \\
B(i)=B(i-1)*s \\
C(i)=C(i-1)*s \\
D(i)=D(i-1)*s \\
E(i)=E(i-1)*s \\
\text{enddo}
\]

5 independent updates on a single thread do the same job!
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
Pure update benchmark can be vectorized → 2 F / cycle (store limited)

Recursive update:
- SMT can fill pipeline bubbles
- A single thread can do so as well
- Bandwidth does not increase through SMT
- SMT can not replace SIMD!
Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”

Truth

1. A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.

2. If a pipeline is already full, SMT will not improve its utilization.
Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

Truth:
1. If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.
2. If the maximum memory bandwidth is not reached, SMT may help since it can fill bubbles in the LOAD pipeline.
Myth: “SMT can help bridge the latency to memory (more outstanding references).”

Truth: Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets “wasted” in the cache hierarchy.
<table>
<thead>
<tr>
<th>SMT: When it may help, and when not</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional parallelization</strong></td>
</tr>
<tr>
<td><strong>FP-only parallel loop code</strong></td>
</tr>
<tr>
<td><strong>Frequent thread synchronization</strong></td>
</tr>
<tr>
<td><strong>Code sensitive to cache size</strong></td>
</tr>
<tr>
<td><strong>Strongly memory-bound code</strong></td>
</tr>
<tr>
<td><strong>Independent pipeline-unfriendly instruction streams</strong></td>
</tr>
</tbody>
</table>