Ingredients for good parallel performance on multicore-based systems

Georg Hager\textsuperscript{(a)} and Gerhard Wellein\textsuperscript{(a,b)}

\textsuperscript{(a)}HPC Services, Erlangen Regional Computing Center (RRZE)
\textsuperscript{(b)}Department for Computer Science
Friedrich-Alexander-University Erlangen-Nuremberg

SC10 Tutorial M16
Nov 15th, 2010, New Orleans, LA
Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- **New chances with multicore hardware**
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- **Summary**

- **Appendix**
Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- **New chances with multicore hardware**
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- **Summary**
- **Appendix**
Welcome to the multi-/manycore era
The free lunch is over: But Moore’s law continues

- In 1965 Gordon Moore claimed:
  #transistors on chip doubles every ≈24 months

Intel Nehalem EX: 2.3 Billion

- We are living in the multicore era → Is really everyone aware of that?
Welcome to the multi-/manycore era
The game is over: But Moore’s law continues

By courtesy of D. Vrsalovic, Intel

Power envelope:
Max. 95–130 W

Power consumption:
\[ P = f \times (V_{\text{core}})^2 \]

\[ V_{\text{core}} \sim 0.9–1.2 \text{ V} \]

Same process technology:
\[ P \sim f^3 \]
Welcome to the multi-/many-core era
The game is over: But Moore’s law continues

- Required relative frequency reduction to run \( m \) cores \((m \text{ times transistors})\) on a die at the same power envelope

![Graph showing reduction of clock speed with core count]

Year: 2007/08

8 cores running at half speed of a single core CPU = same energy

- 65 nm technology:
  - Sun T2 ("Niagara") 1.4 GHz → 8 cores
  - Intel Woodcrest 3.0 GHz → 2 cores
The x86 multicore evolution so far
Intel Single-Dual-/Quad-/Hexa-/Cores (one-socket view)

Hyperthreading/SMT is back!

Nehalem EP
"Core i7"

Westmere EP

2011:
"Sandy Bridge"
SSE → AVX
128 Bit → 256 Bit

45 nm
32 nm
Welcome to the multi-/many-core era
A new feature: shared on-chip resources

Shared outer-level cache

- Fast data transfer
- Fast thread synchronisation
- Data Coherency!
- Increased intra-cache traffic?
- Scalable bandwidth?
- MPI parallelization?

<table>
<thead>
<tr>
<th>AMD Opteron</th>
<th>Intel Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Istanbul</td>
<td>Westmere</td>
</tr>
<tr>
<td>6 cores @ 2.8 GHz</td>
<td>6 cores @ 2.93 GHz</td>
</tr>
<tr>
<td>L1: 64 KB</td>
<td>L1: 32 KB</td>
</tr>
<tr>
<td>L2: 512 KB</td>
<td>L2: 256 KB</td>
</tr>
<tr>
<td>L3: 6 MB</td>
<td>L3: 12 MB</td>
</tr>
<tr>
<td>2 X DDR2-800</td>
<td>3 X DDR3-1333</td>
</tr>
<tr>
<td>→ 12.8 GB/s</td>
<td>→ 31.8 GB/s</td>
</tr>
<tr>
<td>HT2000 → 8 GB/s/dir</td>
<td>2 X QPI6.4</td>
</tr>
<tr>
<td></td>
<td>→ 12.8 GB/s/dir</td>
</tr>
</tbody>
</table>
From UMA to ccNUMA
Basic architecture of commodity compute cluster nodes

Dual-socket Intel "Core2" node:

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

Shared Address Space within the node!

Dual-socket AMD (Istanbul) / Intel (Westmere) node:

Cache-coherent Non-Uniform Memory Architecture (ccNUMA)
HT / QPI provide scalable bandwidth at the expense of ccNUMA architectures:
Where does my data finally end up?
Back to the 2-chip-per-case age:
AMD Magny-Cours – a 2x6-core socket

- **AMD: “Magny-Cours”**
  - 12-core socket comprising two 6-core chips connected via 1.5 HT links
  - Main memory access: 2 DDR3-Channels per 6-core chip → 1/3 DDR3-Channel per core
  - 2 socket server → 4 memory locality domains → ccNUMA within a socket!
  - 4 socket server:

- **Network balance (QDR+2P Magny Cours) ~ 240 GF/s / 3 GB/s = 80 F/B**
  (2003: Intel Xeon DP 2.66 GHz + GBit ~ 10 GF/s / 0.12 GB/s = 80 B/F)
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!

Covered in detail in the hybrid MPI+OpenMP tutorial
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
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
Section summary: What to take home

- **Multicore is here to stay**
  - Shifting complexity from hardware back to software

- **Increasing core counts**
  - 4-12 today, 16-32 tomorrow?
  - x2 or x4 per cores node

- **Shared vs. separate caches**
  - Complex chip/node topologies

- **UMA is practically gone; ccNUMA will prevail**
  - “Easy” bandwidth scalability, but programming implications (see later)
  - Bandwidth bottleneck prevails on the socket

- **Programming models that take care of those changes are still in heavy flux**
  - We are left with MPI and OpenMP for now
  - This is complex enough, as we will see…
Tutorial outline

- Introduction
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- Multicore performance tools
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- Impact of processor/node topology on program performance
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- New chances with multicore hardware
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- Summary
- Appendix
Probing node topology

- Standard tools
- likwid-topology
- hwloc
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
```
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://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-features`: View and enable/disable hardware prefetchers
  - `likwid-bench`: Low-level bandwidth benchmark generator tool
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
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8
  - Linux OS
**Output of likwid-topology**

**CPU name:** Intel Core i7 processor  
**CPU clock:** 2666683826 Hz

**********************************************************************************

**Hardware Thread Topology**  
**********************************************************************************

<table>
<thead>
<tr>
<th>Sockets:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores per socket:</td>
<td>4</td>
</tr>
<tr>
<td>Threads per core:</td>
<td>2</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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Output of likwid-topology continued

Socket 0: ( 0 1 2 3 4 5 6 7 )
Socket 1: ( 8 9 10 11 12 13 14 15 )

Cache Topology

Level: 1
Size: 32 kB
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )

Level: 2
Size: 256 kB
Cache groups: ( 0 1 ) ( 2 3 ) ( 4 5 ) ( 6 7 ) ( 8 9 ) ( 10 11 ) ( 12 13 ) ( 14 15 )

Level: 3
Size: 8 MB
Cache groups: ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 )

NUMA Topology

NUMA domains: 2

Domain 0:
Processors: 0 1 2 3 4 5 6 7
Memory: 5182.37 MB free of total 6132.83 MB

Domain 1:
Processors: 8 9 10 11 12 13 14 15
Memory: 5568.5 MB free of total 6144 MB
Output of likwid-topology

- ... and also try the ultra-cool \( -g \) option!
**hwloc**

- **Successor to (and extension of) PLPA, part of OpenMPI development**
- **Comprehensive API and command line tool to extract topology info**
- **Supports several OSs and CPU types**
- **Pinning API available**
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
Generic thread/process-core affinity under Linux

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

- Binds processes/threads to a set of CPUs. Examples:
  - `taskset -c 0,2 mpirun -np 2 ./a.out` # doesn’t always work
  - `taskset 0x0006 ./a.out`
  - `taskset -c 4 33187`

- Processes can still move in 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
  - Caveat: Linux scheduler does not always use the full set
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**
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!
Explicit Process/Thread Binding With PLPA on Linux:
http://www.open-mpi.org/software/plpa/

- **Portable Linux Processor Affinity**
- **Wrapper library for** `sched_*affinity()` **functions**
  - Robust against changes in kernel API
- **Example for pure OpenMP:** Pinning of threads

```c
#include <plpa.h>
...
#pragma omp parallel
{
#pragma omp critical
{
if(PLPA_NAME(api_probe)() != PLPA_PROBE_OK) {
  cerr << "PLPA failed!" << endl; exit(1);
}
plpa_cpu_set_t msk;
PLPA_CPU_ZERO(&msk);
int cpu = omp_get_thread_num();
PLPA_CPU_SET(cpu, &msk);
PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &msk);
}
```

- **Similar for pure MPI and MPI+OpenMP hybrid code**

Care about correct core numbering! 0…N-1 is not always contiguous! If required, reorder by a map:

```c
cpu = map[cpu];
```

Which core to run on?

Pinning available?

Pin “me”
Trixi's Cluster Configurations Tutorial

1. 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
- 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:**

- `likwid-pin -t intel -c 0,2,4-6 ./myApp parameters`
- `mpirun likwid-pin -s 0x3 -c 0,3,5,6 ./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
[pthreads wrapper] PIN MASK: 0->1 1->4 2->5
[pthreads wrapper] SKIP MASK: 0x1
[pthreads wrapper 0] Notice: Using libpthread.so.0
  threadid 1073809728 -> SKIP
[pthreads wrapper 1] Notice: Using libpthread.so.0
  threadid 1078008128 -> core 1 - OK
[pthreads wrapper 2] Notice: Using libpthread.so.0
  threadid 1082206528 -> core 4 - OK
[pthreads wrapper 3] Notice: Using libpthread.so.0
  threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```
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:

```
likwid-pin -c N:0-7 ./a.out
```

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

```
likwid-pin -c S0:0-3@S1:0-3 ./a.out
```
More examples: Hybrid MPI+OpenMP

Using Intel MPI+compiler & home-grown mpirun

One MPI process per node (w/ explicit logical numbering)

One MPI process per socket (no pinning inside socket required)

OpenMP threads pinned “round robin” across cores (logical core numbers due to cpu set established by mpirun)

Two MPI processes per socket (dito)
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

No pinning

Pinning (physical cores first)
Monitoring the Binding

- How can we see whether the measures for binding are really effective?
  - `sched_getaffinity()`, ...

- `top`:

```
top - 16:05:03 up 24 days, 7:24, 32 users, load average: 5.47, 4.92, 3.52
Tasks: 419 total, 4 running, 415 sleeping, 0 stopped, 0 zombie
Cpu(s): 95.7% us, 1.1% sy, 1.6% ni, 0.0% id, 1.4% wa, 0.0% hi, 0.2% si
Mem: 8157028k total, 8131252k used, 25776k free, 2772k buffers
Swap: 8393848k total, 93168k used, 8300680k free, 7160040k cached

   PID USER      PR  VIRT  RES  SHR  NI P S %CPU %MEM   TIME COMMAND
 23914 unrz55    25  277m 223m 2660   0 2 R 99.9  2.8 23:42 dmrg_0.26_WOODY
 24284 unrz55    16  8580 1556  928   0 2 R  0.2  0.0   0:00 top
 4789 unrz55    15 40220 1452 1448   0 0 S  0.0  0.0 0:00 sshd
 4790 unrz55    15  7900  552  548   0 3 S  0.0  0.0 0:00 tcsh
```

- Press “H” for showing separate threads
Probing performance behavior

- **How do we find out about the performance 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`

```plaintext
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
```
likwid-perfCtr

Example usage with preconfigured metric group

$ env OMP_NUM_THREADS=4 likwid-perfCtr -c 0-3 -g FLOPS_DP likwid-pin -c 0-3 ./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>
Section summary: What to take home

- **Figuring out the node topology is usually the hardest part**
  - Virtual/physical cores, cache groups, cache parameters
  - This information is usually scattered across many sources

- **LIKWID-topology**
  - One tool for all topology parameters
  - Supports Intel and AMD processors under Linux (currently)

- **Generic affinity tools**
  - Taskset, numactl do not pin individual threads
  - Manual (explicit) pinning from within code

- **LIKWID-pin**
  - Binds threads/processes to cores
  - Optional abstraction of strange numbering schemes (logical numbering)

- **LIKWID-perfCtr**
  - End-to-end hardware performance metric measurement
  - Finds out about basic architectural requirements of a program
Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- **New chances with multicore hardware**
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- **Summary**
- **Appendix**
General remarks on the performance properties of multicore multisocket systems
The parallel vector triad benchmark
A “swiss army knife” for microbenchmarking

- Simple streaming benchmark:

```c
for(int j=0; j < NITER; j++){
    //pragma omp parallel for
    for(i=0; i < N; ++i)
        a[i]=b[i]+c[i]*d[i];
    if(OBSCURE)
        dummy(a,b,c,d);
}
```

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
The parallel vector triad benchmark

Optimal code on x86 machines

timing(&wct_start, &cput_start);
#pragma omp parallel private(j)
{
    for(j=0; j<niter; j++){
        if(size > CACHE_SIZE>>5) {
            #pragma omp parallel for
            #pragma vector always
            #pragma vector aligned
            #pragma vector nontemporal
            for(i=0; i<size; ++i)
                a[i]=b[i]+c[i]*d[i];
        } else {
            #pragma omp parallel for
            #pragma vector always
            #pragma vector aligned
            for(i=0; i<size; ++i)
                a[i]=b[i]+c[i]*d[i];
        }
        if(a[5]<0.0)
    }
}
timing(&wct_end, &cput_end);

// size = multiple of 8
int vector_size(int n){
    return int(pow(1.3,n))&(-8);
}

Large-N version (NT)

Small-N version (noNT)
The parallel vector triad benchmark

Performance results on Xeon 5160 node

- OMP overhead
- Cross-socket synch
- Team restart
- (small) L2 bottleneck
- Aggregate L2
- NT stores

The chart shows performance results for various parallel configurations on an Xeon 5160 node, highlighting different bottlenecks and overheads.
Bandwidth limitations: Memory
Some problems get even worse….

- System balance = \( \frac{\text{Peak Bandwidth [MByte/s]}}{\text{Peak Flops [MFlop/s]}} \)
  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 saturation effects in cache and memory
Bandwidth limitations: Memory and cache

Scalability of shared data paths on a socket

**Memory**

**L3 CACHE**

**P**

**C**

**L3 Load**

**L3 Load – L3 Store**

**L3 Cache**

**Memory**

**Intel Nehalem EP 2.66 GHz (DDR3-1333)**

**AMD Istanbul 2.6 GHz (DDR2-???)**

**# cores**

**Effective bandwidth [GB/s]**
Bandwidth limitations: Outer-level cache  
*L3 bandwidth may scale a bit better in future systems…*

- **Intel Nehalem EX**  
  - 8-core chip; 24 MB L3  
  - 4 DDR3-channels per socket  
  - 4 sockets EA system: 128 GB DDR3

- **Nehalem EX: New L3 design**  
  - 8 segments connected by ring  
  - Scalable bandwidth  
  - Lesson learned from “Larabee”  
  - Will show up in future generations, e.g., Sandy Bridge
Ameliorating bandwidth limitations by on-socket ccNUMA
AMD Magny-Cours – a ccNUMA 12-core socket

- **AMD “Magny-Cours”** available as 8-core or 12-core!
  - 12-core socket implemented as two 6-core chips connected via 1.5 HT links
  
  - Main memory access: 2 DDR3-Channels per 6-core chip → 1/3 DDR3-Channel per core

- 2 socket server → 4 memory locality domains → ccNUMA within a socket!

- 4 socket server:

  - Network balance (QDR+2P Magny Cours) ~ 240 GF/s / 3 GB/s = 80 F/B
    
    (2003: Intel Xeon DP 2.66 GHz + GBit ~ 10 GF/s / 0.12 GB/s = 80 B/F)
Ameliorating bandwidth limitations by on-socket ccNUMA

**AMD Magny-Cours – a ccNUMA 12-core socket**

- **AMD EA system – configuration:**
  - 2 x AMD Opteron 6172 (2x6 cores; 2x6MB L3; 2.1 GHz)
  - 64 GB DDR3-1333 MHz

- **Stream (triad w/ NT stores):**
  - 1 socket (12 cores): 24.8 GB/s
  - 2 sockets: 49.7 GB/s

- **Local vs. remote data access**

<table>
<thead>
<tr>
<th>Local / remote</th>
<th>Single thread (triad)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0 → LD0</td>
<td>7.8 GB/s</td>
</tr>
<tr>
<td>P0 → LD1</td>
<td>5.1 GB/s</td>
</tr>
<tr>
<td>P0 → LD2</td>
<td>5.1 GB/s</td>
</tr>
<tr>
<td>P0 → LD3</td>
<td>3.0 GB/s</td>
</tr>
</tbody>
</table>
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:

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

- Usually many spMVMs required to solve a problem

- Case study: Performance data on one 24-core AMD Magny Cours node
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

- **Case 1: Large matrix**
Case 2: Medium size

Application: Sparse matrix-vector multiply
Strong scaling on one Magny-Cours node

Intrasocket bandwidth bottleneck

Working set fits in aggregate cache
Application: Sparse matrix-vector multiply
Strong scaling on one Magny-Cours node

- **Case 3: Small size**

![Graph showing performance vs. threads]

- 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)
Example: HP DL585 G5
4-socket ccNUMA Opteron 8220 Server

- **CPU**
  - 64 kB L1 per core
  - 1 MB L2 per core
  - No shared caches
  - On-chip memory controller (MI)
  - 10.6 GB/s local memory bandwidth

- **HyperTransport 1000 network**
  - 4 GB/s per link per direction

- **3 distance categories for core-to-memory connections:**
  - same LD
  - 1 hop
  - 2 hops

- **Q1:** What are the real penalties for non-local accesses?
- **Q2:** What is the impact of contention?
Effect of non-local access on HP DL585 G5:

Serial vector triad $A(:)=B(:)+C(:)*D(:)$
Contestion vs. parallel access on HP DL585 G5:

OpenMP vector triad \( A(:) = B(:) + C(:) \times D(:) \)

- \( T = \# \) threads
- \( S = \# \) sockets

In-cache performance unharmed by ccNUMA

- Single LD saturated by 2 cores!
- Perfect scaling across LDs
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 Data Locality

- The programmer must ensure that memory pages get mapped locally in the first place (and then prevent migration)
  - Rigorously apply the "Golden Rule"
    - i.e. we have to take a closer look at initialization code
  - Some non-locality at domain boundaries may be unavoidable
  - Stack data may be another matter altogether:

    ```c
    void f(int s) { // called many times with different s
        double a[s];     // c99 feature
        // where are the physical pages of a[] now???
        ...
    }
    ```

- Fine-tuning is possible (see later)

- Prerequisite: Keep threads/processes where they are
  - Affinity enforcement (pinning) is key (see earlier section)
Simplest case: explicit initialization

```fortran
integer, parameter :: N = 1000000
real*8 A(N), B(N)

A = 0.d0

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

```fortran
integer, parameter :: N = 1000000
real*8 A(N), B(N)

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

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

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

```fortran
integer,parameter :: N=1000000
real*8 A(N), B(N)

READ(1000) A
!$OMP parallel do
do I = 1, N
    B(i) = function ( A(i) )
end do

integer,parameter :: N=1000000
real*8 A(N),B(N)

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

- **Required condition:** OpenMP loop schedule of initialization must be the same as in all computational loops
  - Best 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
    - 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
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 `new` operator or special function that places the memory before constructors are called (PAGE_BITS = base-2 log of pagesize)

```cpp
template <class T> T* pnew(size_t n) {
  size_t st = sizeof(T);
  int ofs, len = n*st;
  int i, pages = len >> PAGE_BITS;
  char *p = new char[len];
  #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;
  }
  #pragma omp parallel for schedule(static) private(ofs)
  for(ofs=0; ofs<n; ++ofs) {
    new(static_cast<void*>(p+ofs*st)) T;
  }
  return static_cast<T*>(p);
}
```

**parallel first touch**

**placement new!**
Coding for Data Locality:  
NUMA allocator for parallel first touch in \texttt{std::vector<>}

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:
\texttt{vector<double,NUMA_Allocator<double> > \ x(1000000)}
Memory Locality Problems

- **Locality of reference is key to scalable performance on ccNUMA**
  - Less of a problem with distributed memory (MPI) programming, but see below

- **What factors can destroy locality?**

- **MPI programming:**
  - Processes lose their association with the CPU the mapping took place on originally
  - OS kernel tries to maintain strong affinity, but sometimes fails

- **Shared Memory Programming (OpenMP, ...):**
  - Threads losing association with the CPU the mapping took place on originally
  - Improper initialization of distributed data

- **All cases:**
  - Other agents (e.g., OS kernel) may fill memory with data that prevents optimal placement of user data
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-perfCt can be used to measure nonlocal memory accesses
  - Example for Intel Nehalem (Core i7):

    ```bash
    env OMP_NUM_THREADS=8 likwid-perfCtr -g MEM -c 0-7 \
    likwid-pin -t intel -c 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>
</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>
</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>
</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>
</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>
</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
```

```bash
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
  - \(\rightarrow\) 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
  - 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 vs. UMA and the Linux buffer cache
- Compare two 4-way systems: AMD Opteron ccNUMA vs. Intel UMA, 4 GB main memory
- Run 4 concurrent triads (512 MB each) after writing a large file
- Report performance vs. file size
- Drop FS cache after each data point
ccNUMA placement and erratic access patterns

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

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

In both cases page placement cannot easily be fixed for perfect parallel access

Or you have to use tasking/dynamic scheduling:

!$OMP parallel
!$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
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
Performance impact of round-robin page placement with dynamic scheduling/tasking

- **OpenMP vector triad benchmark** \( A(:) = B(:) + C(:) \times D(:) \) with large array lengths on a 4-LD ccNUMA machine
- **Round-robin page placement** (see previous slide)
- **Static vs. dynamic loop scheduling**, varying chunk size

![Graph showing performance impact](image)

- Static loop schedule matches initialization, but no page alignment of arrays
- Asymptotic limit: 75% of all page accesses are nonlocal
- Full cache line transfer, partial access
OpenMP performance issues on multicore

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

**Synchronization of threads via shared caches**

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

Threads are synchronized at **explicit** AND **implicit** barriers.

On x86 systems there is no hardware support for synchronization.

- **Tested synchronization constructs**
  - **OpenMP** Barrier
  - **pthreads** Barrier
  - **Spin waiting loop** software solution

- **Test machines (Linux OS):**
  - Intel Core 2 Quad Q9550 (2.83 GHz)
  - Intel Core i7 920 (2.66 GHz)
### Thread synchronization overhead

**Barrier overhead in CPU cycles: pthreads vs. OpenMP vs. spin loop**

<table>
<thead>
<tr>
<th>Method</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q9550 (shared L2)</td>
<td>i7 920 (shared L3)</td>
</tr>
<tr>
<td>pthreads_barrier_wait</td>
<td>23739</td>
<td>6511</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>
<tr>
<td></td>
<td>42533</td>
<td>9820</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>977</td>
<td>814</td>
</tr>
<tr>
<td>Spin loop</td>
<td>1106</td>
<td>475</td>
</tr>
</tbody>
</table>

- pthreads → OS kernel call
- Spin loop does fine for shared cache sync
- OpenMP & Intel compiler
Thread synchronization overhead

Barrier overhead: OpenMP icc vs. gcc

<table>
<thead>
<tr>
<th></th>
<th>Q9550 (shared L2)</th>
<th>i7 920 (shared L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc 4.3.3</td>
<td>22603</td>
<td>7333</td>
</tr>
<tr>
<td>icc 11.0</td>
<td>399</td>
<td>469</td>
</tr>
<tr>
<td>4 Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc 4.3.3</td>
<td>64143</td>
<td>10901</td>
</tr>
<tr>
<td>icc 11.0</td>
<td>977</td>
<td>814</td>
</tr>
</tbody>
</table>

gcc obviously uses a pthreads barrier for the OpenMP barrier:

\[ P \quad P \quad P \quad P \]

→ Affinity enforcement is vital for getting small, reproducible sync overhead!
### Thread synchronization overhead

**Barrier overhead: Topology influence**

<table>
<thead>
<tr>
<th></th>
<th>Xeon E5420 2 Threads</th>
<th>shared L2</th>
<th>same socket</th>
<th>different socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthreads_barrier_wait</td>
<td>5863</td>
<td>27032</td>
<td>27647</td>
<td></td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>576</td>
<td>760</td>
<td>1269</td>
<td></td>
</tr>
<tr>
<td>Spin loop</td>
<td>259</td>
<td>485</td>
<td>11602</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>2761</td>
<td>479</td>
<td>1206</td>
<td></td>
</tr>
<tr>
<td>Spin loop</td>
<td>17388</td>
<td>267</td>
<td>787</td>
<td></td>
</tr>
</tbody>
</table>

- SMT can be a big performance problem for synchronizing threads
  - Well known for a long time → see below
  - Roll-your-own sync mechanism may be better sometimes, but good compilers do a good job, too
Work distribution overhead

Influence of thread-core affinity

- Overhead microbenchmark:

```c
!$OMP PARALLEL DO SCHEDULE(RUNTIME) REDUCTION(+ : s)
do i=1,N
    s = s + compute(i)
enddo
!$OMP END PARALLEL DO
```

- Choose N large so that synchronization overhead is negligible
- `compute()` implements purely computational workload
  - no bandwidth effects
- Run with 2 threads
Simultaneous multi-threading

Principles and performance impact
SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently

- SMT principle (2-way example):
SMT impact

- SMT adds another layer of topology (inside the physical core)

- 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

  Unrelated work in other thread can fill the pipeline bubbles

- 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
  ```
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: When it may help, and when not

<table>
<thead>
<tr>
<th>Ingredient</th>
<th>SMT Help?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional parallelization</td>
<td>✔️</td>
</tr>
<tr>
<td>FP-only parallel loop code</td>
<td>❌</td>
</tr>
<tr>
<td>Frequent thread synchronization</td>
<td>❌</td>
</tr>
<tr>
<td>Code sensitive to cache size</td>
<td>❌</td>
</tr>
<tr>
<td>Strongly memory-bound code</td>
<td>❌</td>
</tr>
<tr>
<td>Independent pipeline-unfriendly instruction streams</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Understanding MPI communication in multicore environments

Intra-node vs. inter-node MPI

MPI Cartesian topologies and rank-subdomain mapping
Intranode MPI

- **Common misconception**: Intranode MPI is infinitely fast compared to internode

- **Reality**
  - Intranode latency is much smaller than internode
  - Intranode asymptotic bandwidth is surprisingly comparable to internode
  - Difference in saturation behavior

- **Other issues**
  - Mapping between ranks, subdomains and cores with Cartesian MPI topologies
  - Overlapping intranode with internode communication
MPI and Multicores
Clusters: Unidirectional internode Ping-Pong bandwidth

QDR/GBit ~ 30X
MPI and Multicores
Clusters: Unidirectional intranode Ping-Pong bandwidth

Mapping problem for most efficient communication paths!?
“Best possible” MPI:
Minimizing cross-node communication

- **Example:** Stencil solver with halo exchange

![Diagram of halo exchange]

- **Goal:** Reduce inter-node halo traffic
- **Subdomains exchange halo with neighbors**
  - Populate a node's ranks with “maximum neighboring” subdomains
  - This minimizes a node's communication surface

- **Shouldn’t MPI_CART_CREATE (w/ reorder) take care of this?**
MPI rank-subdomain mapping in Cartesian topologies:
A 3D stencil solver and the growing number of cores per node

- Linear SD distribution
- Optimal SD distribution

- Common” MPI library behavior

- Nehalem EP 2-socket
- Istanbul 2-socket
- Shanghai 4-socket
- Magny Cours 2-socket
- Magny Cours 4-socket
- Nehalem EX 4-socket
- Sun Niagara 2

Node comm surface vs. # cores per node
MPI rank-subdomain mapping:
3D stencil solver – measurements for 8ppn and 4ppn GBE vs. IB

32 MPI processes

8 ppn QDR-IB

4 ppn SDR-IB

~ 1.5x
Section summary: What to take home

- **Bandwidth saturation** is a reality, in cache and memory
  - Use knowledge to choose the “right” number of threads/processes per node
  - You must know where those threads/processes should run
  - You must know the architectural requirements of your application

- **ccNUMA architecture** must be considered for bandwidth-bound code
  - Topology awareness, again
  - First touch page placement
  - Problems with dynamic scheduling and tasking: Round-robin placement is the “cheap way out”

- **OpenMP overhead**
  - Barrier (synchronization) often dominates the loop overhead
  - Work distribution and sync overhead is strongly topology-dependent
  - Strong influence of compiler
  - Synchronizing threads on “logical cores” (SMT threads) may be expensive

- **Intrane node MPI**
  - May not be as fast as you think…
  - Becomes more important as core counts increase
  - May not be handled optimally by your MPI library
Interlude:
What can software do for you?
Common Lore  
*Performance/Parallelization at the node level: Software does it*

Automatic parallelization for moderate processor counts is known for more than 15 years – simple testbed for modern multicores:

```fortran
allocate( x(0:N+1,0:N+1,0:N+1) )
allocate( y(0:N+1,0:N+1,0:N+1) )
x=0.d0
y=0.d0

... somewhere in a subroutine ...
do k = 1,N
  do j = 1,N
    do i = 1,N
      y(i,j,k) = \( b^* \left( x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1) \right) \)
    enddo
  enddo
enddo
```

Performance Metric: Million Lattice Site Updates per second (MLUPs)  
Equivalent MFLOPs: 6 FLOP/LUP * MLUPs  
Equivalent GByte/s: 24 Byte/LUP * MLUPs
Common Lore

Performance/Parallelization at the node level: Software does it

- **Intel Fortran compiler:**
  
  ```fortran
  ifort -O3 -xW -parallel -par-report2 ...
  ```

  - **Version 9.1.** (admittedly an older one…)
    - Innermost i-loop is SIMD vectorized, which prevents compiler from auto-parallelization: *serial loop: line 141: not a parallel candidate due to loop already vectorized*
    - No other loop is parallelized…

  - **Version 11.1.** (the latest one…)
    - Outermost k-loop is parallelized: *Jacobi_3D.F(139): (col. 10) remark: LOOP WAS AUTO-PARALLELIZED.*
    - Innermost i-loop is vectorized.
    - Most other loop structures are ignored by “parallelizer”, e.g. `x=0.d0` and `y=0.d0`: *Jacobi_3D.F(37): (col. 16) remark: loop was not parallelized: insufficient computational work*
Common Lore
Performance/Parallelization at the node level: Software does it

- **PGI compiler (V 10.6)**
  
  `pgf90 -tp nehalem-64 -fastsse -Mconcur -Minfo=par,vect`

  - Performs outer loop parallelization of k-loop
    139, Parallel code generated with block distribution if trip count is greater than or equal to 33
  
  - and vectorization of inner i-loop:
    141, Generated 4 alternate loops for the loop Generated vector sse code for the loop

  - Also the array instructions `(x=0.d0; y=0.d0)` used for initialization are parallelized:
    37, Parallel code generated with block distribution if trip count is greater than or equal to 50

  - Version 7.2. does the same job but some switches must be adapted

- **gfortran: No automatic parallelization feature so far (?!)**
Common Lore

Performance/Parallelization at the node level: Software does it

- 2-socket Intel Xeon 5550 (Nehalem; 2.66 GHz) node

![Graph]

- STREAM bandwidth:
  - Node: ~36-40 GB/s
  - Socket: ~17-20 GB/s

- Performance variations → Thread / core affinity?!
- Intel: No scalability 4→8 threads?

Cubic domain size: N=320 (blocking of j-loop)
Controlling thread affinity / binding
Intel / PGI compilers

- **Intel compiler controls thread-core affinity via **\texttt{KMP\_AFFINITY}** environment variable**
  - \texttt{KMP\_AFFINITY="granularity=fine,compact,1,0"} is packs the threads in a blockwise fashion ignoring the SMT threads. (equivalent to \texttt{likwid-pin -c 0-7})
  - Add \texttt{"verbose"} to get information at runtime
  - Cf. extensive Intel documentation
  - Disable when using other tools, e.g. likwid: \texttt{KMP\_AFFINITY=disabled}
  - Builtin affinity does not work on non-Intel hardware

- **PGI compiler offers compiler options:**
  - \texttt{Mconcur=bind} (binds threads to cores; link time option)
  - \texttt{Mconcur= numa} (prevents OS from process / thread migration; link time option)
  - No manual control about thread-core affinity
  - **Interaction likwid \leftrightarrow PGI ?!**
Thread binding and ccNUMA effects
7-point 3D stencil on 2-socket Intel Nehalem system

- Performance drops if 8 threads instead of 4 access a single memory domain:
  Remote access of 4 through QPI!
Thread binding and ccNUMA effects
7-point 3D stencil on 2-socket AMD Magny-Cours system

- 12-core Magny-Cours: A single socket holds two tightly HT-connected 6-core chips → 2-socket system has 4 data locality domains

Cubic domain size: $N=320$ (blocking of j-loop)

**OMP_SCHEDULE=** “static”

Performance [MLUPs]

<table>
<thead>
<tr>
<th>#threads</th>
<th>#L3 groups</th>
<th>#sockets</th>
<th>Serial Init.</th>
<th>Parallel Init.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>221</td>
<td>221</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1</td>
<td>347</td>
<td>1005</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>2</td>
<td>286</td>
<td>1860</td>
</tr>
</tbody>
</table>

3 levels of HT connections:
1.5x HT – 1x HT – 0.5x HT
Common Lore
Performance/Parallelization at the node level: Software does it

Based on Jacobi performance results one could claim victory, but increase complexity a bit, e.g. simple Gauß-Seidel instead of Jacobi

... somewhere in a subroutine ...

do k = 1,N
  do j = 1,N
    do i = 1,N
      \[ \mathbf{x}(i,j,k) = b \ast (\mathbf{x}(i-1,j,k) + \mathbf{x}(i+1,j,k) + \mathbf{x}(i,j-1,k) + \mathbf{x}(i,j+1,k) + \mathbf{x}(i,j,k-1) + \mathbf{x}(i,j,k+1)) \]
    enddo
  enddo
enddo

A bit more complex 3D 7-point stencil
update(„Gauß-Seidel“)

Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: \( 6 \text{ FLOP/LUP} \ast \text{MLUPs} \)
Equivalent GByte/s: \( 16 \text{ Byte/LUP} \ast \text{MLUPs} \)

Performance of Gauß-Seidel should be up to 1.5x faster than Jacobi if main memory bandwidth is the limitation
Common Lore

Performance/Parallelization at the node level: Software does it

- State of the art compilers do not parallelize Gauß-Seidel iteration scheme: loop was not parallelized: existence of parallel dependence
- That’s true but there are simple ways to remove the dependency even for the lexicographic Gauß-Seidel
- 10 yrs+ Hitachi’s compiler supported “pipeline parallel processing” (cf. later slides for more details on this technique)!

- There seem to be major problems to optimize even the serial code
  - 1 Intel Xeon X5550 (2.66 GHz) core
  - Reference: Jacobi
    - 430 MLUPs
  - Target Gauß-Seidel: 645 MLUPs

<table>
<thead>
<tr>
<th>Compiler</th>
<th>MLUPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel V9.1.</td>
<td>290</td>
</tr>
<tr>
<td>Intel V11.1.072</td>
<td>345</td>
</tr>
<tr>
<td>pgf90 V10.6.</td>
<td>149</td>
</tr>
<tr>
<td>pgf90 V7.2.1</td>
<td>149</td>
</tr>
</tbody>
</table>
Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- **New chances with multicore hardware**
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- **Summary**
- **Appendix**
Multicore awareness
Classical Approaches: Parallelize & Reduce memory pressure

Multicore processors are still mostly programmed the same way as classic n-way SMP single-core compute nodes!

Simple 3D Jacobi stencil update (sweep):

\[
do \ k = 1 \ , \ N_k \\
\quad do \ j = 1 \ , \ N_j \\
\quad \quad do \ i = 1 \ , \ N_i \\
\quad \quad \quad y(i,j,k) = a+x(i,j,k) + b \times (x(i-1,j,k)+x(i+1,j,k)+x(i,j-1,k)+x(i,j+1,k)+x(i,j,k-1)+x(i,j,k+1)) \\
\quad \quad enddo \\
\quad enddo \\
enddo
\]

Performance Metric: Million Lattice Site Updates per second (MLUPs)
Equivalent MFLOPs: 8 FLOP/LUP * MLUPs
Multicore awareness

*Standard sequential implementation*

```
   do t=1, t_{Max}
     do k=1, N
       do j=1, N
         do i=1, N
           y(i, j, k) = ...
         enddo
       enddo
     enddo
   enddo
```
Multicore awareness

Classical Approaches: Parallelize!

```
do t=1, t_{Max}
  !$OMP PARALLEL DO private(…)
    do k=1, N
      do j=1, N
        do i=1, N
          y(i, j, k) = ...
        enddo
      enddo
    enddo
  !$OMP END PARALLEL DO
endo
```
Multicore awareness

Parallelization – reuse data in cache between threads

Do not use domain decomposition!

Instead shift 2\textsuperscript{nd} thread by three i-j planes and proceed to the same domain → 2\textsuperscript{nd} thread loads input data from shared OL cache!

Sync threads/cores after each k-iteration!

“Wavefront Parallelization (WFP)”

\begin{align*}
\text{core0: } & \mathbf{x}(i, j, k-1:k+1)_{t} \rightarrow \mathbf{y}(i, j, k)_{t+1} \\
\text{core1: } & \mathbf{y}(i, j, (k-3):(k-1))_{t+1} \rightarrow \mathbf{x}(i, j, k-2)_{t+2}
\end{align*}
Use small ring buffer
\[ \text{tmp}( :, :, 0:3) \]
which fits into the cache

\[ \text{Save main memory data transfers for } y( :, :, :) ! \]

16 Byte / 2 LUP !

8 Byte / LUP !

**Compare with optimal baseline (nontemporal stores on y):**

**Maximum speedup of 2 can be expected**

(assuming infinitely fast cache and no overhead for OMP BARRIER after each k-iteration)
**Multicore awareness**

*WF parallelization – reuse data in cache between threads*

Thread 0: \( x(:,:,k-1:k+1)_t \) \( \rightarrow \) \( tmp(:,:,\text{mod}(k,4)) \)

Thread 1: \( tmp(:,:,\text{mod}(k-3,4):\text{mod}(k-1,4)) \) \( \rightarrow \) \( x(:,:,k-2)_t+2 \)

**Performance model including finite cache bandwidth \((B_C)\)**

Time for 2 LUP:

\[
T_{2\text{LUP}} = 16 \text{ Byte}/B_M + x \times 8 \text{ Byte} / B_C = T_0 \left( 1 + x/2 \times B_M/B_C \right)
\]

Minimum value: \( x = 2 \)

Speed-Up vs. baseline: \( S_W = 2T_0/T_{2\text{LUP}} = 2 / \left( 1 + B_M/B_C \right) \)

\( B_C \) and \( B_M \) are measured in saturation runs:

Clovertown: \( B_M/B_C = 1/12 \) \( \Rightarrow S_W = 1.85 \)

Nehalem: \( B_M/B_C = 1/4 \) \( \Rightarrow S_W = 1.6 \)
Jacobi solver

WFP: Propagating four wavefronts on native quadcores (1x4)

Running $\mathbf{tb}$ wavefronts requires $\mathbf{tb}-1$ temporary arrays $\mathbf{tmp}$ to be held in cache!

Max. performance gain (vs. optimal baseline): $\mathbf{tb} = 4$

Extensive use of cache bandwidth!

1 x 4 distribution

$\mathbf{x}(\vdash, \vdash, \vdash)$
Jacobi solver

**WF parallelization: New choices on native quad-cores**

Thread 0: \( x(:, :, k-1:k+1)_t \)

\[ \Rightarrow \text{tmp1}(mod(k, 4)) \]

Thread 1: \( \text{tmp1}(mod(k-3, 4):mod(k-1, 4)) \)

\[ \Rightarrow \text{tmp2}(mod(k-2, 4)) \]

Thread 2: \( \text{tmp2}(mod(k-5, 4):mod(k-3, 4)) \)

\[ \Rightarrow \text{tmp3}(mod(k-4, 4)) \]

Thread 3: \( \text{tmp3}(mod(k-7, 4):mod(k-5, 4)) \)

\[ \Rightarrow x(:, :, k-6)_{t+4} \]

---

**1 x 4 distribution**

- **core0**
  - \( \text{tmp1}(0:3) \)
- **core1**
  - \( \text{tmp2}(0:3) \)
- **core2**
  - \( \text{tmp3}(0:3) \)
- **core3**
  - \( x(:, :, :) \)

**2 x 2 distribution**

- **core0**
  - \( x(:, 1:N/2, :) \)
- **core1**
  - \( x(:, N/2+1:N, :) \)
- **core2**
  - \( \text{tmp0}(\cdot, :, 0:3) \)
- **core3**
  - \( x(:, N/2+1:N, :) \)
Jacobi solver

Wavefront parallelization: L3 group Nehalem

Performance model indicates some potential gain → new compiler tested.

Only marginal benefit when using 4 wavefronts → A single copy stream does not achieve full bandwidth
Multicore-aware parallelization
Wavefront – Jacobi on state-of-the-art multicores

Compare against optimal baseline!

Performance gain $\sim B_{olc} = \frac{L3 \text{ bandwidth}}{\text{memory bandwidth}}$

$B_{olc} \sim 10$

$B_{olc} \sim 2-3$

$B_{olc} \sim 10$
Section summary: What to take home

- **Shared caches are the interesting new feature on current multicore chips**
  - Shared caches provide opportunities for fast synchronization (see sections on OpenMP and intra-node MPI performance)
  - Parallel software should leverage shared caches for performance
  - One approach: Shared cache reuse by WFP
  - In addition fast synchronization (pref. within a socket) allows to exploit parallel structures at finer granularity (shorter loops, frequent synchronisation)

- **WFP technique can easily be extended to many regular stencil based iterative methods, e.g.**
  - Gauß-Seidel  \(\rightarrow\) done
  - Lattice-Boltzmann flow solvers  \(\rightarrow\) work in progress
  - Multigrid-smoother  \(\rightarrow\) work in progress

- **WFP can be extended to hybrid MPI+OpenMP parallelization**
  - See references
Tutorial outline

- **Introduction**
  - Architecture of multisocket multicore systems
  - Nomenclature
  - Current developments
  - Programming models

- **Multicore performance tools**
  - Finding out about system topology
  - Affinity enforcement
  - Performance counter measurements

- **Impact of processor/node topology on program performance**
  - Bandwidth saturation effects
  - Programming for ccNUMA
  - OpenMP performance
  - Simultaneous multithreading (SMT)
  - Intranode vs. internode MPI

- **New chances with multicore hardware**
  - Pipeline parallel processing
    - Case study: Wavefront parallelization of stencil codes

- **Summary**

- **Appendix**
Summary & Conclusions

- **Multicore/multisocket topology needs to be considered:**
  - OpenMP performance
  - MPI communication parameters
  - Shared resources

- **Be aware of the architectural requirements of your code**
  - Bandwidth vs. compute
  - Synchronization
  - Communication

- **Use appropriate tools**
  - Node topology: likwid-pin, hwloc
  - Affinity enforcement: likwid-pin
  - Simple profiling: likwid-perfCtr

- **Try to leverage the new architectural feature of modern multicore chips**
  - Shared caches!
Appendix: References

Books:


Papers:

- M. Wittmann, G. Hager and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP), IPDPS 2010, April 23rd, 2010, Atlanta, GA.
References

Papers continued:


Advanced OpenMP: Pipeline parallel processing → Eliminating recursion

Parallelizing a 3D Gauss-Seidel solver
The Gauss-Seidel algorithm in 3D

double precision, parameter :: osth=1/6.d0
do it=1,itmax    ! number of iterations (sweeps)
  ! not parallelizable right away
  do k=1,kmax
    do j=1,jmax
      do i=1,imax
        phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k)
                       + phi(i,j-1,k) + phi(i,j+1,k)
                       + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
      enddo
    enddo
  enddo
endo
3D Gauss-Seidel parallelized

- **Pipeline parallel principle: Wind-up phase**
  - Parallelize middle j-loop and shift thread execution in k-direction to account for data dependencies
  - Each diagonal ($W_t$) is executed by $t$ threads concurrently
  - Threads sync after each k-update
3D Gauss-Seidel parallelized

- Full pipeline: All threads execute
3D Gauss-Seidel parallelized: The code

```Fortran
!$OMP PARALLEL PRIVATE(k,j,i,jStart,jEnd,threadID)
   threadID=OMP_GET_THREAD_NUM()
!$OMP SINGLE
   numThreads=OMP_GET_NUM_THREADS()
!$OMP END SINGLE
   jStart=jmax/numThreads*threadID
   jEnd=jStart+jmax/numThreads ! jmax is a multiple of numThreads
   do l=1,kmax+numThreads-1
      k=l-threadID
      if((k.ge.1).and.(k.le.kmax)) then
         do j=jStart,jEnd
            do i=1,iMax
               phi(i,j,k) = ( phi(i-1,j,k) + phi(i+1,j,k) 
                             + phi(i,j-1,k) + phi(i,j+1,k) 
                             + phi(i,j,k-1) + phi(i,j,k+1) ) * osth
            enddo
         enddo
      endif
      !$OMP BARRIER
      enddo
!$OMP END PARALLEL
```
Parallel 3D Gauß-Seidel

- Gauß-Seidel can also be parallelized using a red-black (2D) or ??? (3D) scheme

- But data dependency is representative for several linear (sparse) solvers \( Ax=b \) arising from regular discretization, e.g. Stone’s Strong Implicit (SIP) solver based on incomplete \( A \sim \) LU factorization
  - Still used in many CFD FV codes (→ RRZE report)
  - \( L \) & \( U \): Each contains 3 non-zero off-diagonals only!
  - Solving \( Lx=b \) or \( Ux=c \) has loop carried data dependencies similar to GS → PPP
Presenter Biographies

- **Georg Hager** holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. See his blog at [http://blogs.fau.de/hager](http://blogs.fau.de/hager) for current activities, publications, and talks.

- **Gerhard Wellein** holds a PhD in solid state physics from the University of Bayreuth and is a professor at the Department for Computer Science at the University of Erlangen. He leads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than ten years of experience in teaching HPC techniques to students and scientists from computational science and engineering programs. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.
Abstract

- Tutorial M16: Ingredients for Good Parallel Performance on Multicore-based systems
- Presenter(s): Georg Hager, Gerhard Wellein

ABSTRACT:

This tutorial covers program optimization techniques for multi-core processors and the systems they are used in. It concentrates on the dominating parallel programming paradigms, MPI and OpenMP. We start by giving an architectural overview of multicore processors. Peculiarities like shared vs. separate caches, bandwidth bottlenecks, and ccNUMA characteristics are pointed out. We show typical performance features like synchronization overhead, intranode MPI bandwidths and latencies, ccNUMA locality, and bandwidth saturation (in cache and memory) in order to pinpoint the influence of system topology and thread affinity on the performance of typical parallel programming constructs. Multiple ways of probing system topology and establishing affinity, either by explicit coding or separate tools, are demonstrated. Finally we elaborate on programming techniques that help establish optimal parallel memory access patterns and/or cache reuse, with an emphasis on leveraging shared caches for improving performance.