Multicore Performance and Tools

Probing node topology

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

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

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

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

- **LIKWID** tool suite:

  Like
  I
  Knew
  What
  I’m
  Doing

- Open source tool collection (developed at RRZE):

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

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

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

- **Current tools:**
  - `likwid-topology`: Print thread and cache topology
  - `likwid-powermeter`: Measure energy consumption
  - `likwid-pin`: Pin threaded application without touching code
  - `likwid-perfctr`: Measure performance counters
  - `likwid-bench`: Low-level bandwidth benchmark generator tool
  - … some more
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

(c) RRZE 2014
Output of likwid-topology continued

Graphical:

Socket 0:

Socket 1:
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
Example: STREAM benchmark on 16-core Sandy Bridge: Anarchy vs. thread pinning

There are several reasons for caring about affinity:
- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention
Overview

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

- **taskset** restricts processes/threads to a set of CPUs. Examples:

  ```
  taskset 0x0006 ./a.out
  taskset -c 4 33187
  ```

- Processes/threads can still move within the set!

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

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

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

- Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA
More thread/Process-core affinity ("pinning") options

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`
    - Windows: `SetThreadAffinityMask()`
    - ...

- Support for "semi-automatic" pinning in some compilers/environments
  - Intel compilers > V9.1 (KMP_AFFINITY environment variable)
  - PGI, Pathscale, GNU
  - Generic Linux: `taskset, numactl, likwid-pin` (see below)
  - hwloc tools and API
  - OpenMP 4.0

- Affinity awareness in MPI libraries
  - OpenMPI
  - Intel MPI
  - ...

(c) RRZE 2014
Intel **KMP_AFFINITY** environment variable

- **KMP_AFFINITY**=
  - `<modifier>,...<type>[],<permute>][,<offset>]`

**modifier**
- `granularity=<specifier>` takes the following specifiers: fine, thread, and core
- `norespect`
- `noverbose`
- `proclist={<proc-list>}`
- `respect`
- `verbose`

**type (required)**
- `compact`
- `disabled`
- `explicit (GOMP_CPU_AFFINITY)`
- `none`
- `scatter`

- Default: `noverbose,respect,granularity=core`

- **KMP_AFFINITY**=`verbose,none` to list machine topology map

OS processor IDs

Respect an OS affinity mask in place

(c) RRZE 2014  LIKWID: Topology/Affinity
Intel KMP_AFFINITY examples

- **KMP_AFFINITY=granularity=fine,compact**

- **KMP_AFFINITY=granularity=fine,scatter**
Intel KMP_AFFINITY permute example

- **KMP_AFFINITY=granularity=fine,compact,1,0**

  ```plaintext
  KMP_AFFINITY=granularity=core,compact
  ```

(c) Intel

Threads may float within core
GNU GOMP_AFFINITY

- GOMP_AFFINITY=3, 0-2 used with 6 threads

- Always operates with OS processor IDs

(c) Intel

Roundrobin oversubscription

(c) RRZE 2014 LIKWID: Topology/Affinity
Likwid-pin

Overview

- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Works on any processor type
- Explicit command line interface
- Based on combination of wrapper tool together with overloaded pthread library → binary must be dynamically linked!
- Supports **logical core numbering** using a thread group syntax

Usage examples:

- Physical numbering (as given by likwid-topology):
  ```
  likwid-pin -c 0,2,4-6 .:/myApp parameters
  ```

- Logical numbering by topological entities:
  ```
  likwid-pin -c S0:0-3 .:/myApp parameters
  ```

**OMP_NUM_THREADS is set if not present**
Running the STREAM benchmark with likwid-pin:

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -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 ...]
```
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:

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

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

```bash
likwid-pin -c S0:0-3@S1:0-3 ./a.out
```
Pitkhid-pin
Using logical core numbering with thread groups

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

*Default if -c is not specified!*

(c) RRZE 2014

LIKWID: Topology/Affinity
Advanced options for pinning: Expressions

- Expressions are more powerful in situations where the pin mask would be very long or clumsy

Compact pinning:
likwid-pin -c E:<thread domain>:\<number of threads>\[<chunk size>:<stride>] ...

Scattered pinning across all domains of the designated type:
likwid-pin -c <domaintype>:scatter

- Examples:

  likwid-pin -c E:N:8 ... # equivalent to N:0-7

  likwid-pin -c E:N:120:2:4 ... # Phi: 120 threads, 2 per core

- Scatter across all NUMA domains:
  likwid-pin -c M:scatter

(c) RRZE 2014
LIKWID: Topology/Affinity
likwid-mpirun
MPI startup and Hybrid pinning

- How do you manage affinity with MPI or hybrid MPI/threading?
- In the long run a unified standard is needed
- Till then, likwid-mpirun provides a portable/flexible solution
- The examples here are for Intel MPI/OpenMP programs, but are also applicable to other threading models

Pure MPI:
likwid-mpirun -np 16 -nperdomain S:2 ./a.out

Hybrid:
likwid-mpirun -np 16 -pin S0:0,1_S1:0,1 ./a.out
likwid-mpirun
1 MPI process per node

likwid-mpirun -np 2 -pin N:0-11 ./a.out

Intel MPI+compiler:
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out

(c) RRZE 2014
likwid-mpirun
1 MPI process per socket

likwid-mpirun -np 4 -pin S0:0-5_S1:0-5 ./a.out

Intel MPI+compiler:
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \n    -env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out