

BEAST Lab Organization

LRZ

Dr. Josef Weidendorfer, josef.weidendorfer@lrz.de

Amir Raoofy, amir.raoofy@tum.de

LMU

Sergej Breiter, sergej.breiter@nm.ifi.lmu.de

Minh Thanh Chung, minh.thanh.chung@ifi.lmu.de

Dr. Karl Fűrlinger, karl.fuerlinger@ifi.lmu.de

TUM

Vincent Bode, vincent.bode@tum.de,

Dennis Florian Herr, herrod@tum.de,

Bengisu Elis, bengisu.elis@tum.de



TUM Uhrenturm

Table of Contents

Course Organization

Introduction to BEAST

Introduction to OpenMP

Assignment 1: Vector Triad

Tentative Course Structure

- 12 meetings in total (via Zoom and in class)
 - Last meeting on 28th July 2022
- 6 Assignments
 - 1 week each
- 2 Bigger Projects
 - 2 weeks each
- Student groups of 3
- Two groups will present their reports at the meeting right after deadline
 - Presentation notification will be sent 2 days before presentations
 - No slides, go through your report and talk about your findings

Repository Structure

Gitlab main repository: <https://gitlab.lrz.de/beastlab22ss>

- Each team has a repository, which includes:
 - Lecture slides
 - Assignment material
 - Code template
 - Your code submissions and report (once you place it there)
- Only solutions on the `main` branch will be graded !
 - At the due date, your current master branch state is automatically tagged and archived
 - Make sure your code and report is there by the deadline
- Groups are already set.
- Machine account information will be sent via e-mail - Change passwords please !

Infrastructure Usage

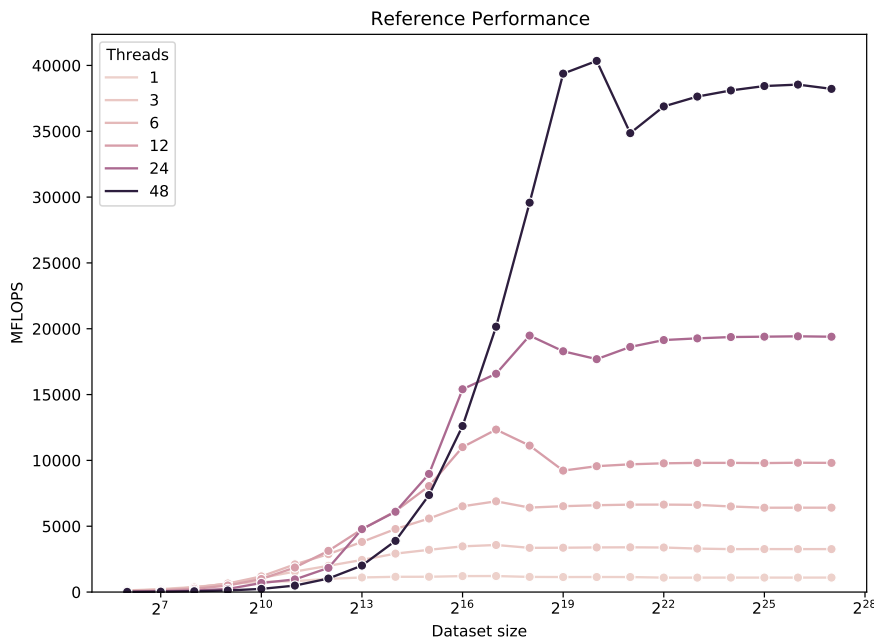
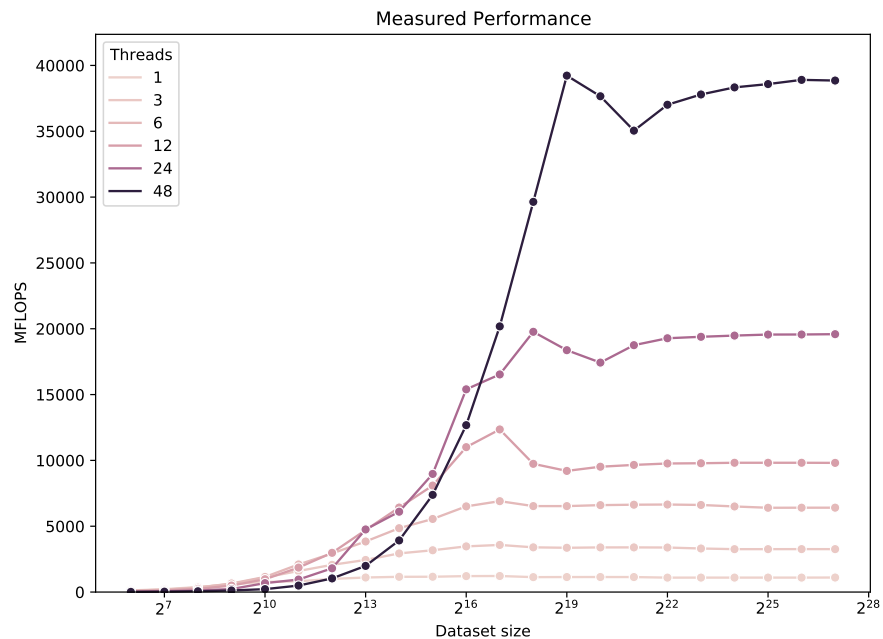
GitLab

- We need you to sign your commits.
- Tutorial: https://docs.gitlab.com/ee/user/project/repository/gpg_signed_commits/
- If GitLab shows a **Verified** label on your commits you are good to go.
- We will tag your last commit before each deadline.

Continuous Integration

- Allows for rapid feedback to your code solutions, helps you get set up.
- Code runs on Fujitsu machines. Performance data is automatically collected and visualized.

Sample CI Visualization



Performance data is stored in the build job artifacts (right hand column when viewing build job).

CI Usage

Benefit: Shows you when your code fails to produce correct results or performance.

	#462227		P master 6e46d7b9 Fix artifact paths for assig...		00:01:07 2 hours ago	
	#462226		P master ad0dae44 Update artifact paths for s...		00:01:05 2 hours ago	
	#462197		P master 5392ec90 Add lecture slides		00:01:06 2 hours ago	

CI ensures you have the measuring basics correct. You still need to conduct most of the experiments and explain your findings.



Up Next: Introduction to BEAST

BEAST Lab

Introduction to Systems

April 28, 2022 | Josef Weidendorfer

Collaboration among 3 institutions

LMU
TUM
LRZ

LMU – MNM/Prof. Kranzlmüller
(Karl Förlinger, Minh Chung, Sergej Breiter)

TUM – CAPS/Prof. Schulz
(Bengisu Elis, Dennis-Florian Herr, Vincent Bode)

LRZ - Future Computing Group
(Josef Weidendorfer, Amir Raoofy)

We want you to learn about **performance properties of current architectures**

- Be able to understand and explain performance effects seen from measurements
- Get a deeper understanding of current system designs (CPU / GPU)

Part 1: get started with small codes across systems

- We show key hardware design concepts + a parallel programming model (OpenMP)
- We give you typical small HPC code examples
- You run measurements of different scenarios across systems, compare / discuss results
- We all discuss results in weekly meetings, starting with presentations of groups

Structure:

Memory on CPU (Triad / Traversal) → Compute on CPU (MM) → ... on GPU → Tools

We want you to learn about **performance properties of current architectures**

- Be able to understand and explain performance effects seen from measurements
- Get a deeper understanding of current system designs (CPU / GPU)

Part 2: make use of gained knowledge

- We assign randomly one system to each group
- We give you some larger typical HPC code examples
- You tune the code to get best single-node performance (3 week time)
- We discuss intermediate/final experiences/results in weekly meetings

Target Architectures for the Lab

CPUs

- Intel Icelake (ISA: x86-64 + AVX512)
- AMD Rome (ISA: x86-64 + AVX2)
- Marvell ThunderX2 (ISA: ARM AArch64 + Neon)
- Fujitsu A64FX (ISA: ARM AArch64 + SVE)

GPUs

- NVidia V100
- AMD MI-100



SuperMUC-NG
Top500 (Nov 2018): #8
Lenovo Intel (2019)
311,040 cores
Intel Xeon Skylake
26.9 PetaFlops Peak
19.5 PetaFlops Linpack
719 TeraByte Main Memory
70 PetaByte Disk

BEAST – Bavarian Energy Architecture and Software Testbed

The LRZ Future Computing Testbed



- Help decide about next large system
 - Get experience on benefits of various future architectures for LRZ codes
 - Find best configuration: how much money to spend on compute / memory / network?
 - Enable migration planning: educate own staff / port LRZ tools / prepare courses
 - Support vendor collaboration
- Enable research studies on new technologies
 - Forward looking: LRZ services around future platforms, novel usage models
 - more experimental: FPGAs, AI accelerators, integration of heterogeneity (QC)
 - In partnership with selected researchers from Munich universities

Lot of work to do! Engage students for student work (BA, MA): This Lab!

The Testbed – Available Hardware

2 racks, each with 6 PDUs (for power measurements)

- Max power consumption per rack: 35 kW

Top to bottom (picture from last year)

- 3 switches (Infiniband 200Gb/s HDR), 2x 48port 1Gb/s Ethernet
- Login 1U “testbed.cos.lrz.de”
- **2x AMD Rome** GPU server 2U: “rome1” / “**rome2**”
- Storage 2U with homes
- **2x Marvell ThunderX2** GPU server 2U: “thx1” / “**thx2**”

Not shown:

- HPC CS500 Management 2U + **8 nodes A64FX** “cs1” – “cs8”
- **2x Intel IceLake** GPU server 2U: “ice1” / “**ice2**”



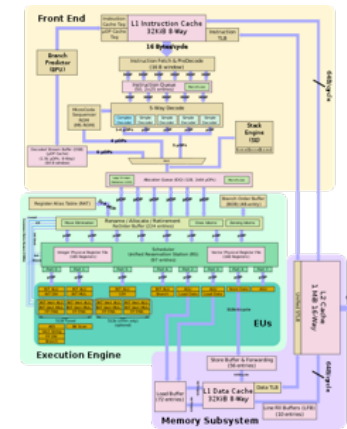
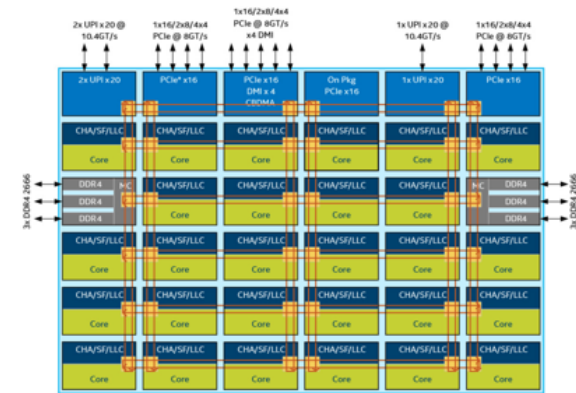
Intel Skylake (available as fallback)

SuperMUC-NG

- Here: only Single-Node experiments
- Node
 - 2 sockets with Intel Skylake Xeon Platinum 8174
 - 2x 24 = 48 cores
 - 2x 512bit vector units per core (8 x DP FMA)
 - 2 threads per core ("Hyper-Threading")
 - 2.3 GHz base (currently: 2.5 GHz), 14nm
 - 96 GB main memory

Links

- <https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG>
- [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server))



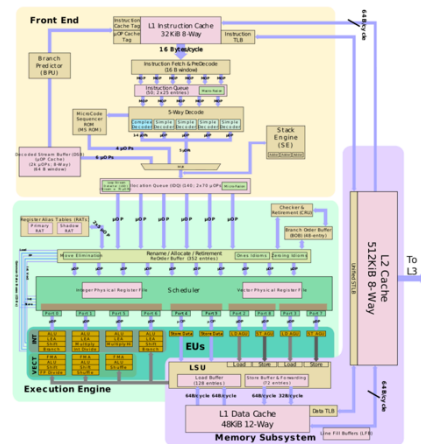
Intel Icelake

Two systems in BEAST

- 2 sockets Intel Xeon (Icelake) Platinum 8360Y
 - 2x 36 = 72 cores
 - 2x 512bit vector units per core (8 x DP FMA)
 - 2 threads per core ("Hyper-Threading")
 - 2.4 GHz base, Intel 10nm
- 512 GB main memory, 1.5 TB Optane NVRam

Links

- [https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(server))
- https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

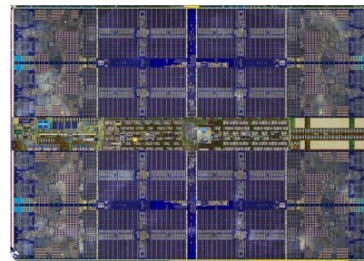


Two systems in BEAST

- 2 sockets with EPYC 7742
- 2x 64 = 128 cores (“Zen2”)
 - Chiplet design: IO-Die + 8x CCX-Dies (2x 4-core)
 - 2x 256-bit vector units per core (4 x DP FMA)
 - 2 threads per core
 - 2.25 GHz base, TSMC 7nm
- 512 GB main memory
- 2x AMD Radeon MI-100 GPUs
 - 7nm, 32GB HBM, PCIe4

Link

- https://en.wikichip.org/wiki/amd/microarchitectures/zen_2



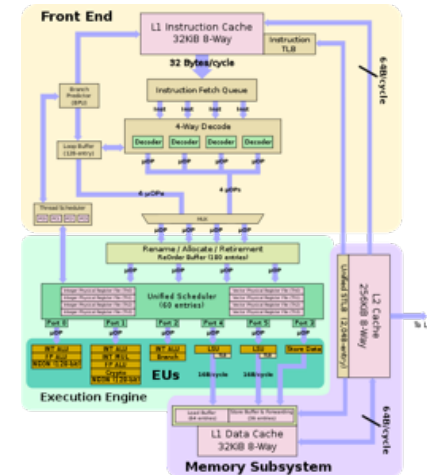
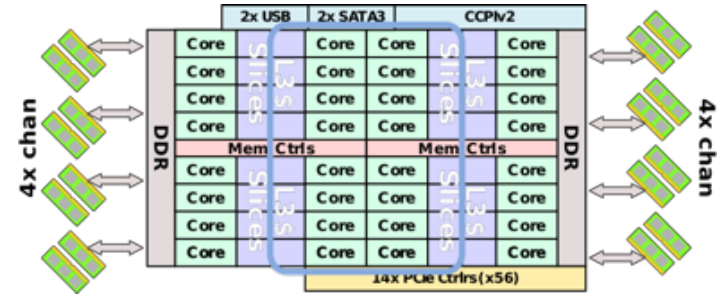
Marvell ThunderX2

Two systems in BEAST

- 2 sockets with ThunderX2 CN9980
- 2x 32 = 64 cores (“Vulcan”)
 - 128-bit vector units (2 x DP FMA)
 - 4 threads per core
 - 2.2 GHz base, 16nm
- 512 GB main memory
- 2x Nvidia V-100
 - Volta, 32GB HBM, PCIe3

Link

- <https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan>



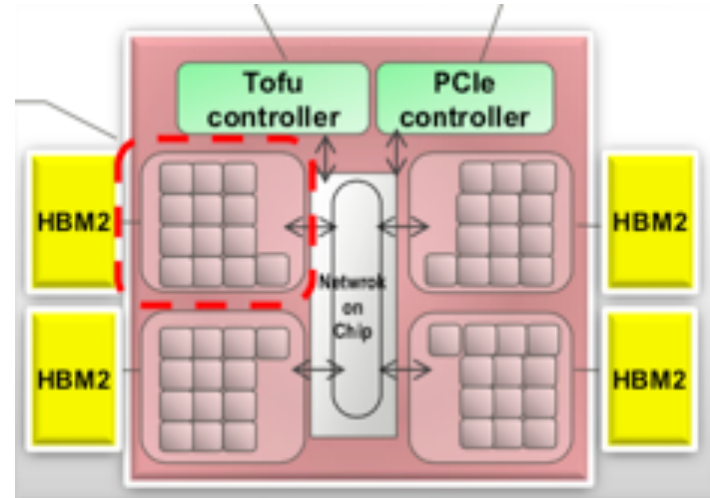
Fujitsu A64FX

HPE CS500 in BEAST

- 8 nodes with one A64FX CPU (“NSP1”)
- 48 cores per CPU
 - 2x 512bit vector units per core
 - 1.8 GHz, TSMC 7nm
 - 4 NUMA domains
- 32 GB HBM2

Link

- https://en.wikipedia.org/wiki/Fujitsu_A64FX



[Fujitsu: The 1st SVE Enabled Arm Processor: A64FX and Building up ARM HPC Ecosystem, 2019]

Access

- via “ssh rbgaccount@lxxhalle.informatik.tu-muenchen.de”
- ssh XXX@skx.supermuc.lrz.de

Compilers

- via “module”, see available packages with “module avail” / load with “module load”
- ICC (“icpc” for C++, “icc” for C, Intel compiler, enable OpenMP: “-openmp”)
- GCC (“g++” C++, “gcc” C, GNU Compiler Collection, enable OpenMP: “-fopenmp”)

Node allocation (test queue, 30 minutes, e.g. project h039y - see “groups”)

- salloc -Ah039y -ptest -t30

Access via Linux Cluster login nodes

- `ssh XXX@lxlogin8.lrz.de` (or `lxlogin1@lrz.de`)
- `ssh testbed.cos.lrz.de`
- `ssh <system>`

If `testbed.cos.lrz.de` is not reachable, retry after 1 hour

- probably just a reboot

Compilers

- system: “gcc”
- via modules: see “module avail”, then “module load <package>”

Access and Usage: Intel Icelake @ BEAST



Access

- `ssh XXX@lxlogin8.lrz.de` (or `lxlogin1@lrz.de`)
- `ssh testbed.cos.lrz.de`
- `ssh ice2`

Compilers

- `gcc`, `icc` (Intel compiler)

Access and Usage: AMD Rome @ BEAST



Access

- `ssh XXX@lxlogin8.lrz.de` (or `lxlogin1@lrz.de`)
- `ssh testbed.cos.lrz.de`
- `ssh rome2`

Compilers

- `gcc`, `clang` (from AMD RocM)

Access and Usage: ThunderX2 @ BEAST



Access

- `ssh XXX@lxlogin1.lrz.de`
- `ssh testbed.cos.lrz.de`
- `ssh thx2`

Compilers

- `gcc`
- (via „module load cuda/11.1.1 llvm“) `clang`

Access and Usage: AMD A64FX @ BEAST



Access

- `ssh XXX@lxlogin1.lrz.de`
- `ssh testbed.cos.lrz.de`
- `ssh cs1 / cs2`

Compilers

- `gcc (8)`
- `gcc 11` (via „`module load gcc/11.0.0`“)
- Cray compiler: “`cc`”, enable OpenMP: “`-h omp`”



Leibniz Supercomputing Centre
of the Bavarian Academy of Sciences and Humanities

Up Next: Introduction to OpenMP



Dr. Karl Furlinger
Institut für Informatik
Lehrstuhl für Kommunikationssysteme und
Systemprogrammierung

OpenMP Basics

BEAST Lab SS 2021

Praktikum
Quantitative Analyse von Hochleistungssystemen (LMU)
Evaluation of Modern Architectures and Accelerators (TUM)



OpenMP

- A method for portable programming of shared memory systems
 - **Open** specification for **Multi-Processing**
- Industry Standard
 - Guided by the OpenMP **Architecture Review Board** (ARB)
 - Major companies and research labs participate in the ARB
 - Current version: v5.1 (November 2020)
- Language extension for C/C++ and Fortran
 - Compiler **directives**
 - Library **routines**
 - Environment **variables**
- www.openmp.org
 - Current specification, tutorials, other resources



OpenMP Example: Hello World

Source Code:

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    #pragma omp parallel
    {
        printf(„Ahoi OpenMP world\n“);
    }
}
```

Compilation:

```
icc -openmp hello.c -o hello

gcc -fopenmp hello.c -o hello
```

The flag to enable OpenMP is
implementation-specific

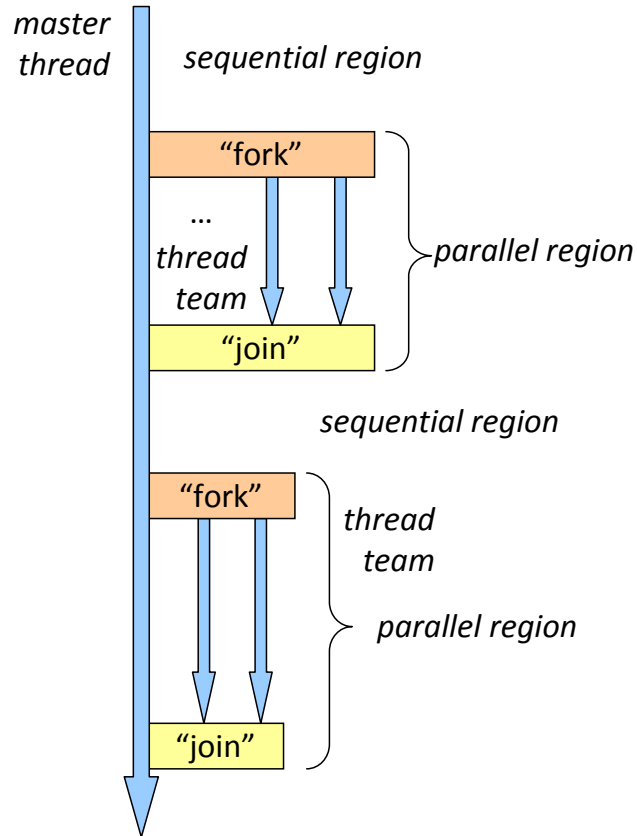
Execution:

```
>$ export OMP_NUM_THREADS=4
>$ ./hello
Ahoi OpenMP world
Ahoi OpenMP world
Ahoi OpenMP world
Ahoi OpenMP world
```

Execution with 2 threads:

```
>$ export OMP_NUM_THREADS=2
>$ ./hello
Ahoi OpenMP world
Ahoi OpenMP world
```

OpenMP Execution Model

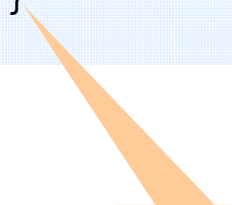


- This model is called the **fork-join** Model (but proper nesting is always enforced)
 - Program starts with a single thread (called the **initial thread**)
 - Parallel regions create additional threads (**team threads**)
 - Team threads disappear (logically) at the end of a parallel region
 - Implementations may keep team threads around in a thread pool for reasons of efficiency
 - There is an **implicit barrier** at the end of a parallel region
 - Number of threads **may change** between parallel regions

Creating Threads: #pragma omp parallel

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    #pragma omp parallel
    {
        printf(„Ahoi OpenMP world\n“);
    }
}
```



The structured block is
executed redundantly
(in parallel) by all
threads

■ **Worksharing** constructs are
used to distribute work
between threads

- for
- sections
- single
- workshare (Fortran only)

Shared and Private Variables

- Variables declared outside the parallel region are **shared** by default

```
#include <stdio.h>
#include <omp.h>

double alpha=1.23;

int main(int argc, char* argv[]) {
    double gamma=23.11;

    #pragma omp parallel
    {
        int mydelta;

        #pragma omp for
        for(int i=0; i<100; i++) {
            do_some_work(i, alpha);
        }
        mydelta = ...;
        gamma+=mydelta;
    }
}
```

shared by default

shared by default

private by default

OK! modifying private copy

!!!Warning: modifying shared variable!!!

- Shared variables
 - Are accessible by all threads (only one copy exists)
- Private variables
 - Accessible only by one thread (each thread has its own copy)
- Data sharing clauses can override defaults

Data Sharing Clauses (Parallel and Work Sharing Constructs)

- **private**(var-list)
 - Variables in var-list are **private**
- **shared**(var-list)
 - Variables in var-list are **shared**
- **default**(private | shared | none)
 - Sets the default for all variables in this region
 - Default **none** raises compiler error if sharing is not explicitly specified
- **firstprivate**(var-list)
 - Variables are private and are initialized with the value of the shared copy before the region
- **lastprivate**(var-list)
 - Variables are private and the value of the thread executing the last iteration of a parallel loop in sequential order is copied to the variable outside of the region.

Worksharing in Parallel Regions

- Goal: distribute work among threads in a parallel region

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char* argv[]) {
    int i;
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0; i<100; i++) {
            do_some_work(i);
        }
    }
}
```

```
// shorthand notation for the above
// combined parallel-workshare
#pragma omp parallel for
```

- **The omp for construct**

- Specifies that the work (loop iterations) should be distributed to the available threads
- **Asserts** that the loop iterations are independent and can be parallelized

Parallel Loop (C/C++)

```
#pragma omp for [clause[,] clause]  
for(i=0; i<..; i++..) { .. }
```

- Loop iterations are distributed between the threads of the team
 - A **loop scheduling clause** specifies exactly **how**
 - Loop scheduling options: **static**, **dynamic**, **guided**, **auto**, and **runtime**
 - E.g., *schedule(static)*
- Characteristics:
 - There is **no synchronization (i.e., barrier) at the entry** of the loop
 - There is an **implicit barrier** at the end of the loop unless a *nowait* clause is specified
 - The loop iteration variable is **private by default**
 - Only **simple** (so-called **canonical forms**) of loops are supported
 - Integer iteration variable, only modified in the increment expression
 - Iteration count can be computed before executing the loop

Loop Scheduling Strategies

```
#pragma omp for schedule(type[, size])
```

- Scheduling type is one of:
 - **static**: chunks of iterations of the specified size are distributed among threads in a **round-robin** fashion
 - **dynamic**: Threads **request chunks** of the specified size from the runtime; when finished executing, a thread requests a new chunk
 - **guided**: like dynamic, but the chunk size is proportional to remaining work; size parameter specifies the minimal chunk size
 - **auto**: decision is delegated to the compiler and/or runtime system
 - **runtime**: defer scheduling decision to runtime selection (via environment variable **OMP_SCHEDULE**); note that it is only possible to specify one schedule for all loops via an environment variable

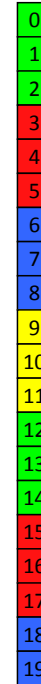
Loop Scheduling Example

```
#pragma omp parallel
{
  // #pragma omp for schedule(static)
  // #pragma omp for schedule(static,3)
  // #pragma omp for schedule(dynamic,1)
  #pragma omp for schedule(dynamic,3)
  for(i=0; i<20; i++) {
    do_some_work(i);
  }
}
```

 Thread 0
 Thread 1
 Thread 2
 Thread 3



static



static,3



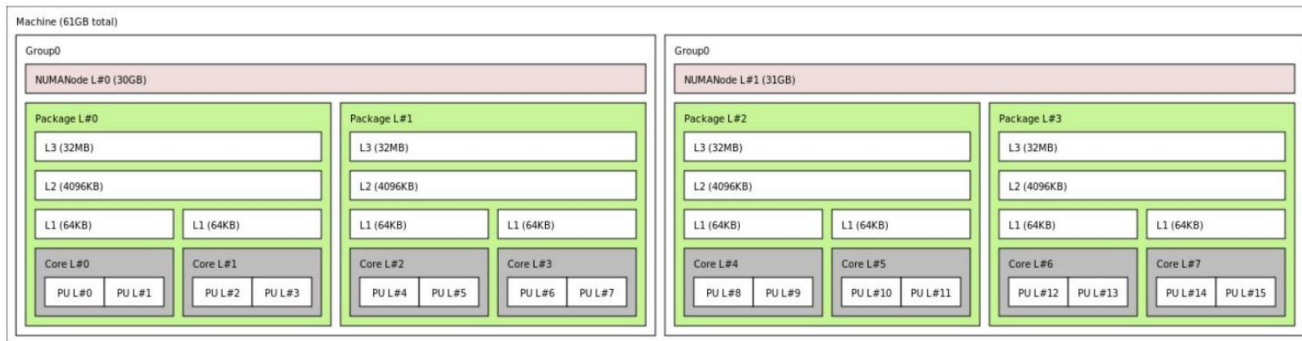
dynamic,1



dynamic,3

Thread Affinity

- How threads are mapped to hardware may influence performance
 - E.g., placement of threads to optimize cache sharing vs. memory bandwidth
 - HWLoc output example



- OpenMP allows the specification of
 - What we consider the unit of locality
 - OMP_PLACES** env. variable = **threads** | **cores** | **sockets**
 - How to distribute threads to places
 - **OMP_PROC_BIND** env. variable and **proc_bind** clause
 - master** | **spread** | **close**

OMP_PLACES Env. Variable

- **OMP_PLACES** specifies a list of places where threads should be executed
 - **sockets** – each place corresponds to a single socket, a socket can have multiple cores
 - **cores** – each place corresponds to a single core, each core can have multiple hardware threads
 - **threads** – each place corresponds to a single hardware thread

export OMP_PLACES=cores

- Places and place lists can also be specified numerically
 - Meaning of numeric IDs depends on the system (/proc/cpuinfo, lscpu)

export OMP_PLACES={0,1,2,3}, {4,5,6,7}, ...

OMP_PROC_BIND Env. Variable and proc_bind clause

- **OMP_PROC_BIND(policy)** or **proc_bind(policy)** clause specify how threads are mapped onto places
 - **master** – each thread in the team is assigned to the same place as the master thread
 - **close** – threads in the team are placed close to the master thread
 - **spread** – threads are spread evenly over the places

- **Examples (HW as in Hwloc example)**

Parallel region with two threads, one per socket

OMP_PLACES=sockets

`#pragma omp parallel num_threads(2) proc_bind(spread)`

Parallel region with four threads, all on one socket

OMP_PLACES=cores

`#pragma omp parallel num_threads(4) proc_bind(close)`

Optimizing for NUMA (1)

- NUMA=Non-Uniform-Memory Access
 - Accessing local data is beneficial for performance
 - Virtual memory is mapped to physical memory in the granularity of pages (typically 4KB)
 - Usually where a memory page gets allocated is determined by a **first touch policy** (i.e., local to the core that first uses a memory page)
 - This implies that the initialization of data structures should reflect the intended later access patterns
 - Bad: Serial initialization and parallel access
 - Bad: Different parallel initialization and parallel access
 - Good: Parallel initialization and parallel access in same way
- Other options:
 - Explicit control using OS mechanisms, e.g., **numactl**

Optimizing for NUMA (2)

- Bad: serialized initialization leads to allocation of B,C,D all in one locality domain

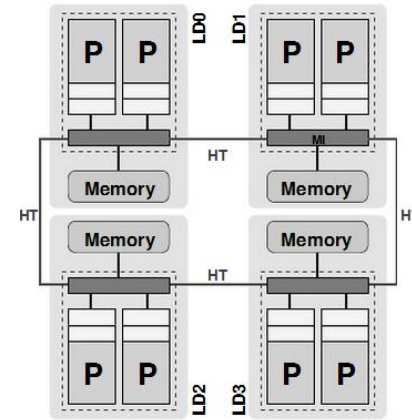
```
// initialize data structures
for(i=0; i<N; i++) {
    B[i]= . . .
    C[i]= . . .
    D[i]= . . .
}
```

```
#pragma omp parallel for
for( i=0; i<N; i++ ) {
    A[i] = B[i]+C[i]*D[i];
}
```

- Good: parallel initialization in the same way it is later accessed (distributed across locality domains)

```
// initialize data structures in parallel
#pragma omp parallel for
for(i=0; i<N; i++) {
    B[i]= . . .
    C[i]= . . .
    D[i]= . . .
}
```

```
#pragma omp parallel for
for( i=0; i<N; i++ ) {
    A[i] = B[i]+C[i]*D[i];
}
```



ccNUMA system with
four locality domains

Image source: Hager, Wellein:
"Introduction to High Performance
Computing for Scientists and
Engineers"



Up Next: Assignment 1: Vector Triad