Chair of Computer Architecture and Parallel Systems
Department of Informatics
Technical University of Munich

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 Fürlinger, karl.fuerlinger@ifi.lmu.de

TUM

Bengisu Elis, bengisu.elis@tum.de

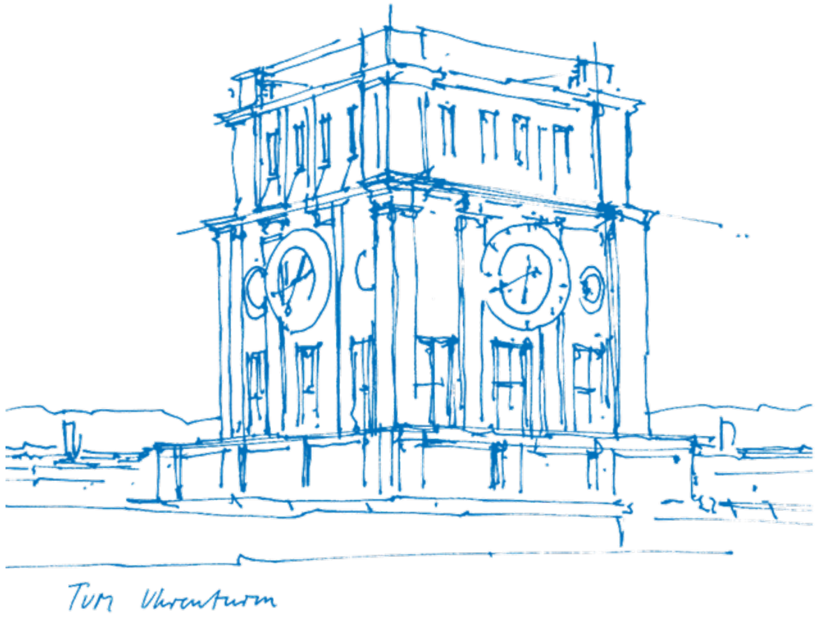


Table of Contents

Tentative Course Structure

- 12 meetings in total
  - Last meeting on January 25th 2024
- 7 Assignments (Assignment 0 is an informational handout)
  - 1 week each
- 2 Projects
  - 2 weeks each
- Student groups of 3 BA students and 2 MA students
- 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
  - About 15-20 minutes talk per group
  - Focus on the **most important and interesting** aspects of your report
  - You **don't** need to address every part of your report
  - Q/A after the student presentations (everyone is highly encouraged to ask questions)

Repository Structure

Gitlab main repository: `https://gitlab.lrz.de/beastlab23ws`

- 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, you current master branch state is automatically tagged and archived
  - Make sure your code and report is there by the deadline
- Machine account information will be sent via e-mail - Change passwords please !

- 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.

Infrastructure Usage

Two-Factor Authentification

- You will work on LRZ systems as part of the lab.
- LRZ requires 2FA.
- Today, We will make sure that everybody registers a 2nd factor.
- For this, we expect you to have a smart phone with an authenticator app.
  - privacyIDEA authenticator
  - or Microsoft Authenticator
- Instructions: `https://doku.lrz.de/two-factor-authentication-prerequisites-35882365.html`

Access to BEAST and each individual system

- see Assignment 0

Infrastructure Usage

Exclusive Resource Allocation

- We use SLURM this semester to schedule your jobs and measurements.
- For your development throughout the week, you can directly ssh into individual machines
- For final measurements we allocate 1 full day where no direct ssh can be done.
- **this semester we allocate Teusdays of every week for exclusive slurm jobs**
- In this timeframe you can only run code through slurm allocations
- We limit the job timelimit to 15 minutes to ensure every gorup gets a chance to schedule jobs during that day.
- here are some usefull links:
  `https://doku.lrz.de/example-parallel-job-scripts-on-the-linux-cluster-10746636.html`
- usefull commands: `sinfo, squeue, salloc, srun, and sbatch`

Up Next: Introduction to BEAST

BEAST Lab

Introduction to Systems

October 19, 2023 | Josef Weidendorfer

Collaboration among 3 institutions

LMU
TUM
LRZ

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

TUM – CAPS/Prof. Schulz
(Bengisu Elis)

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

Focus: Experimental Evaluation

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 / microbenchmark 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:

CPU evaluation (Memory, Compute) ➔ GPU evaluation ➔ Tool

Focus: Experimental Evaluation

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 – Final Project

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

Evaluation of Single-Node Performance

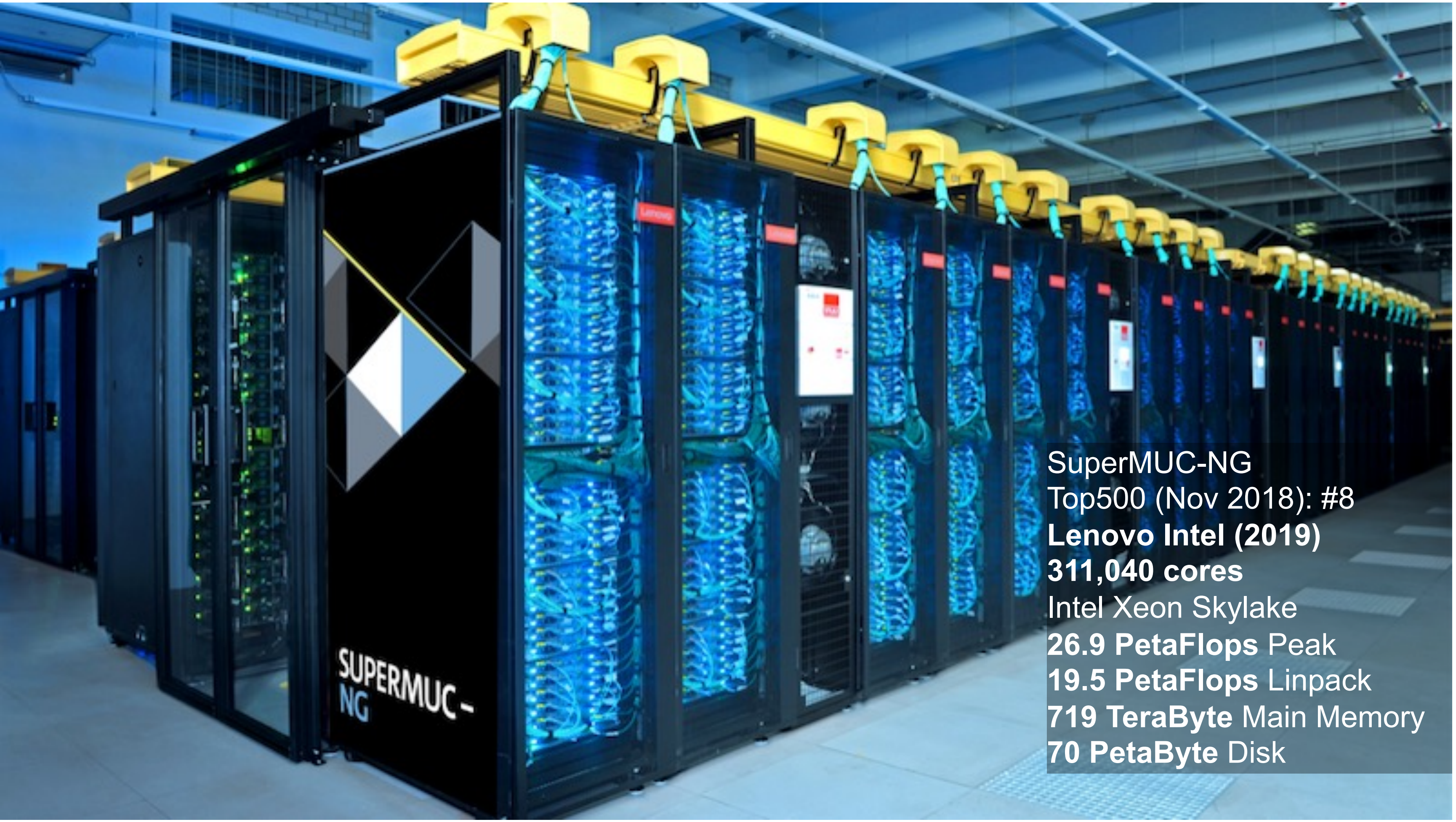
Target Architectures for the Lab

CPUs

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

GPUs

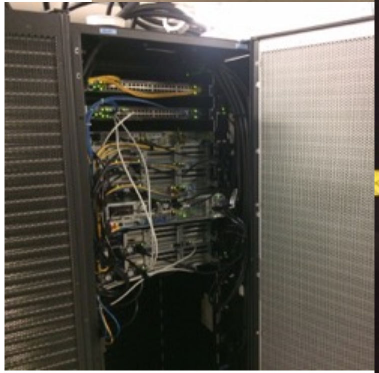
- NVidia V100 (eventually also A100)
- AMD MI-100 (eventually also MI-210)

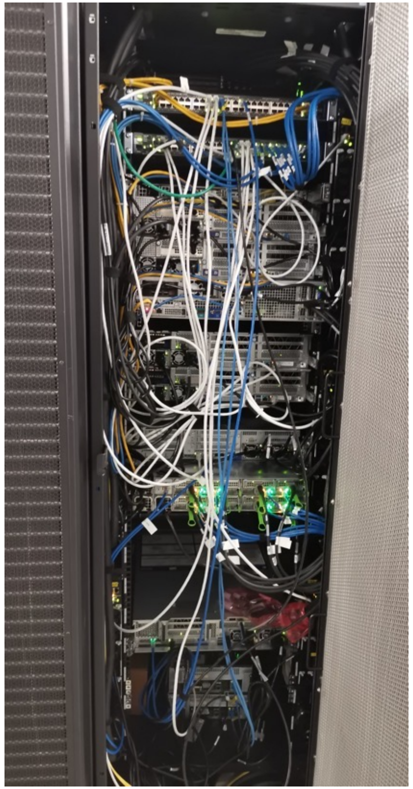


SuperMUC-NG
Top500 (Nov 2018): #8
Lenovo Intel (2019)
311,040 cores
Intel Xeon Skylake
26.9 PetaFlops Peak
19.5 PetaFlops Linpack
719 TeraByte Main Memory
70 PetaByte Disk

BEAST – Bavarian Energy Architecture and Software Testbed

The LRZ Future Computing Testbed





Testbed Objectives

- 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”**
  (to be upgraded: AMD MilanX + GPU**)**
- 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”**

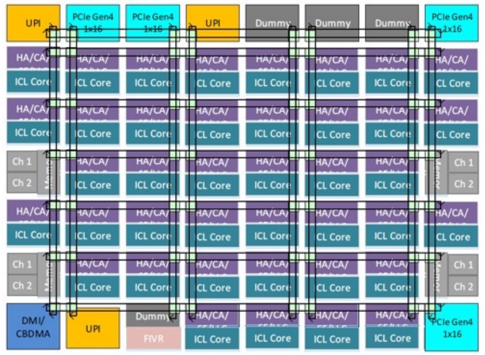
Intel Icelake

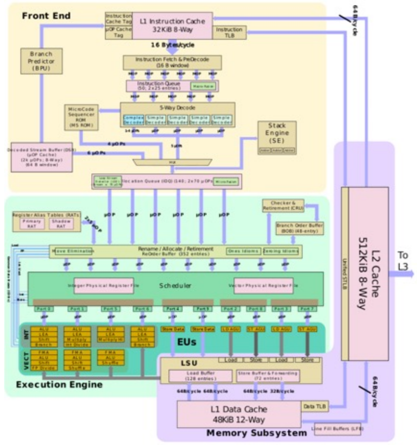
Two systems in BEAST

- 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/sunny_cove





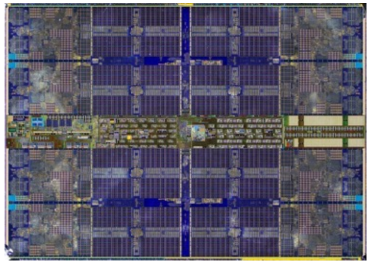
AMD Milan-X

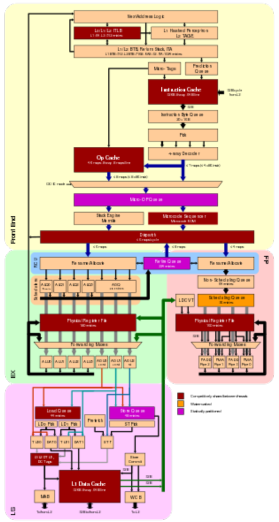
Two systems in BEAST

- 2 sockets with EPYC 7773X
- 2x 64 = 128 cores ("Zen3")
  - Chiplet design: IO-Die + 8x CCX-Dies (each 8-core)
  - 2x 256-bit vector units per core (4 x DP FMA)
  - 2 threads per core
  - 2.2 GHz base, TSMC 7nm
- 1 TB main memory
- 8x AMD Radeon MI-210 GPUs
  - 7nm, 64GB HBM, PCIe4

Link

- https://en.wikichip.org/wiki/amd/microarchitectures/zen_3





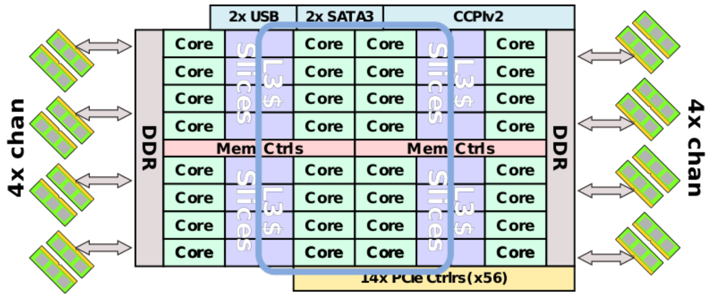
Marvell ThunderX2

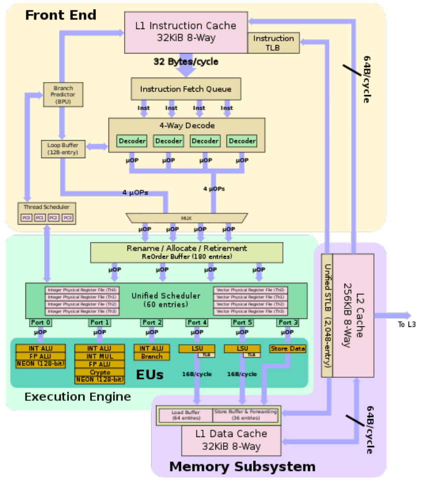
Two systems in BEAST

- 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

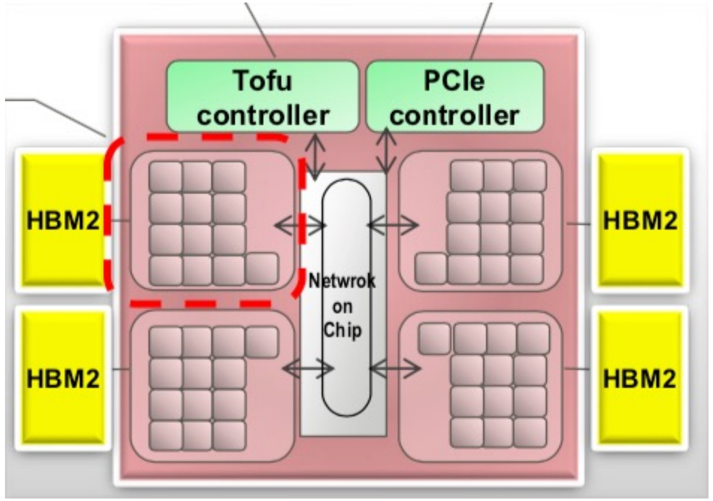




Fujitsu A64FX

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



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

Link

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

Access and Usage: BEAST Systems

Access via Linux Cluster login nodes

- ssh XXX@lxlogin1.lrz.de (or lxlogin2 / 3 / 4)
  - need to configure 2FA TOTP/PUSH via https://simmfa.sim.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@lxlogin1@lrz.de
- ssh testbed.cos.lrz.de
- ssh ice1

Compilers

- gcc, icc (Intel compiler)

Access and Usage: AMD Milan-X @ BEAST

Access

- ssh XXX@lxlogin1.lrz.de
- ssh testbed.cos.lrz.de
- ssh milan2

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: Fujitsu A64FX @ BEAST

Access

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

Compilers

- gcc (8)
- gcc 11 and 13 (via „module load gcc/11.0.0 or module load gcc-13.1.0-djjcsa … “)
- Cray compiler: “cc”, enable OpenMP: “-h omp”

lrz

Leibniz Supercomputing Centre

of the Bavarian Academy of Sciences and Humanities

Up Next: OpenMP Introduction

Sergej Breiter (presenting) and Dr. Karl Fürlinger
Institut für Informatik
Lehrstuhl für Kommunikationssysteme und Systemprogrammierung

OpenMP Basics

BEAST Lab WS 2023/24

Praktikum
Evaluierung moderner HPC-Architekturen und -Beschleuniger

OpenMP

- A method for portable programming of shared memory systems
  - **Open** specification for **M**ulti-**P**rocessing

- Industry Standard
  - Guided by the OpenMP **Architecture Review Board** (ARB)
  - Major companies and research labs participate in the ARB
  - Current version: v5.2 (November 2021)

- Language extension for C/C++ and Fortran
  - Compiler **directives**
  - Library **routines**
  - Environment **variables**

- www.openmp.org
  - Current specification, tutorials,
    other resources (such as examples)

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 -qopenmp hello.c -o hello

icx -fopenmp hello.c -o hello

gcc -fopenmp hello.c
```

The flag to enable OpenMP is **implementation-specific**

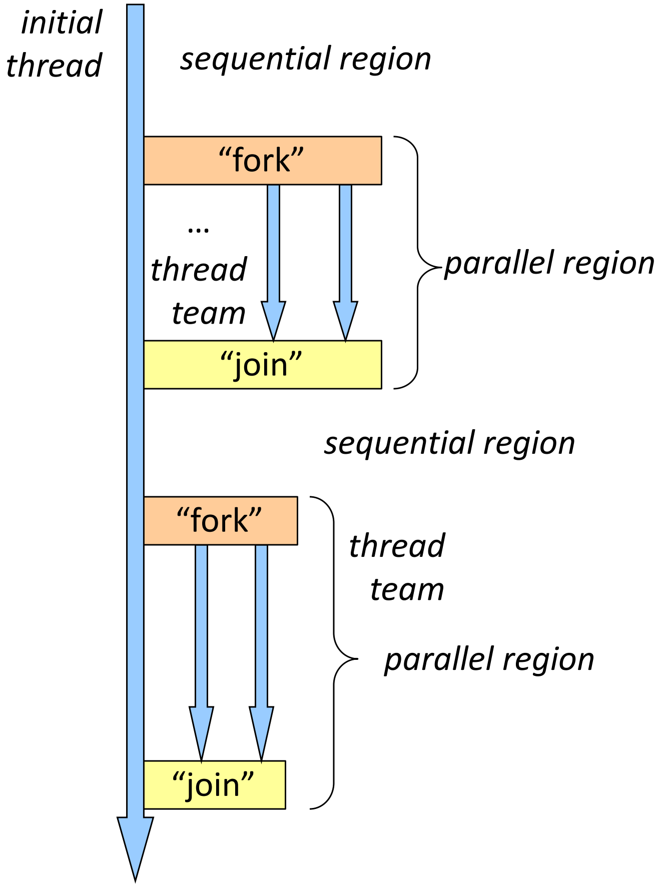
Execution:

```
>$ 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
  - Program starts with a single thread (called the **initial thread**)
  - Parallel regions create additional threads (**team threads**), initial thread becomes the **master thread** in the team
  - 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!!!
Needs some form of synchronization, e.g., *atomic*, *critical*, *locks*

- 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.

Initialization of Private Variables

```
int i, j;
i = 1;
j = 2;

#pragma omp parallel private(i) firstprivate(j)
{
  printf(„i=%d j=%d\n“, i, j);
}
```

Execution:

```
>$ export OMP_NUM_THREADS=4
>$ ./a.out
i=5456498 j=2
i=-732837541 j=2
i=788564 j=2
i=821656 j=2
```

- Private copies of i are **not initialized**!
- Firstprivate copies of j are **initialized to the outside value**

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:
  - The 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

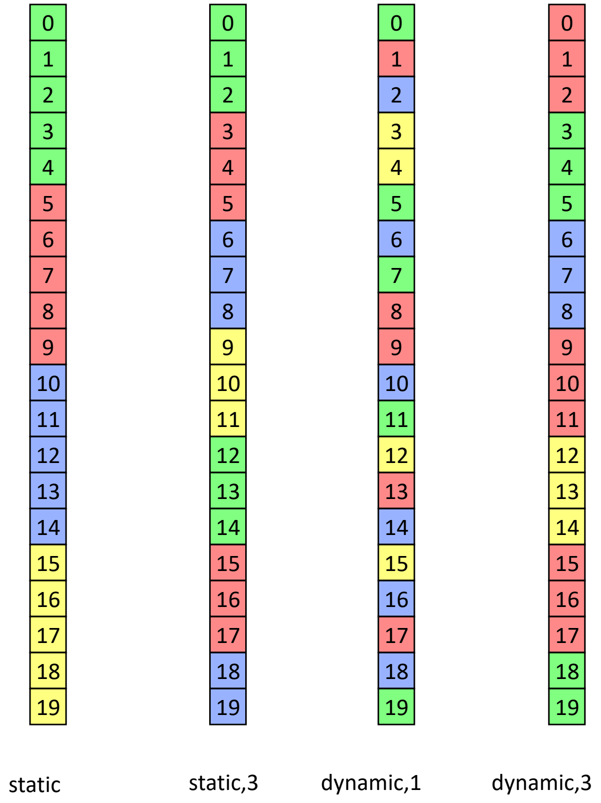
```
#pragma omp for schedule(type[, size])
```

- 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

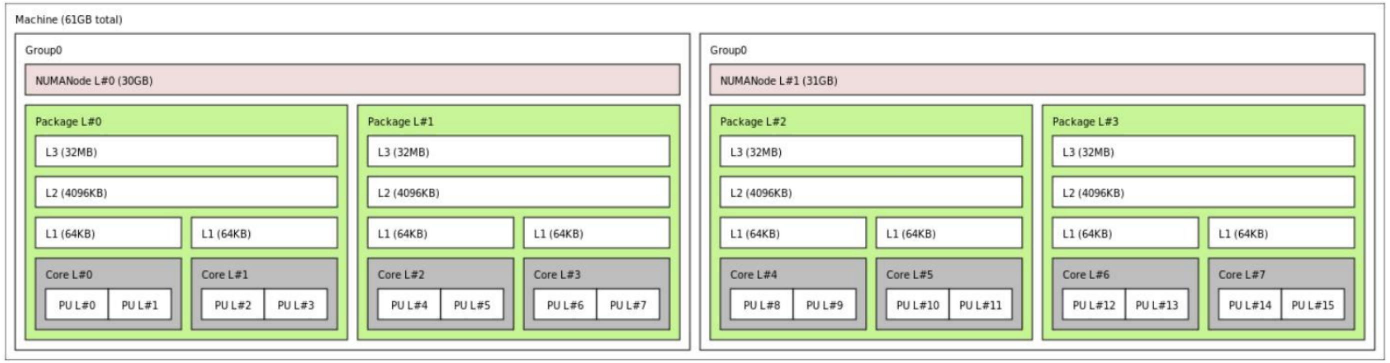
```
#pragma omp parallel
  {
//#pragma omp for schedule(static)
//#pragma omp for schedule(static,3)
//#pragma omp for schedule(dynamic,1)
#pragma omp for schedule(dynamic,3)
       for(i=0; i<20; i++)  {
         do_some_work(i);
       }
  }
```

🟩 Thread 0
🟥 Thread 1
🟦 Thread 2
🟨 Thread 3



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

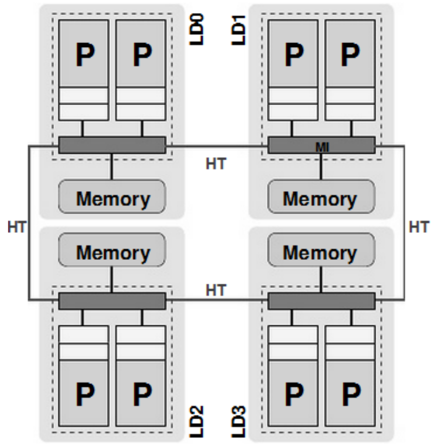
```
// initialize data strucutres
for(i=0; i<N; i++ ) {
  B[i]= . . .
  C[i]= . . .
  D[i]= . . .
}

#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 strucutres 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"