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

OpenMP Basics

BEAST Lab WS 2022/23

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: v5.1 (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**

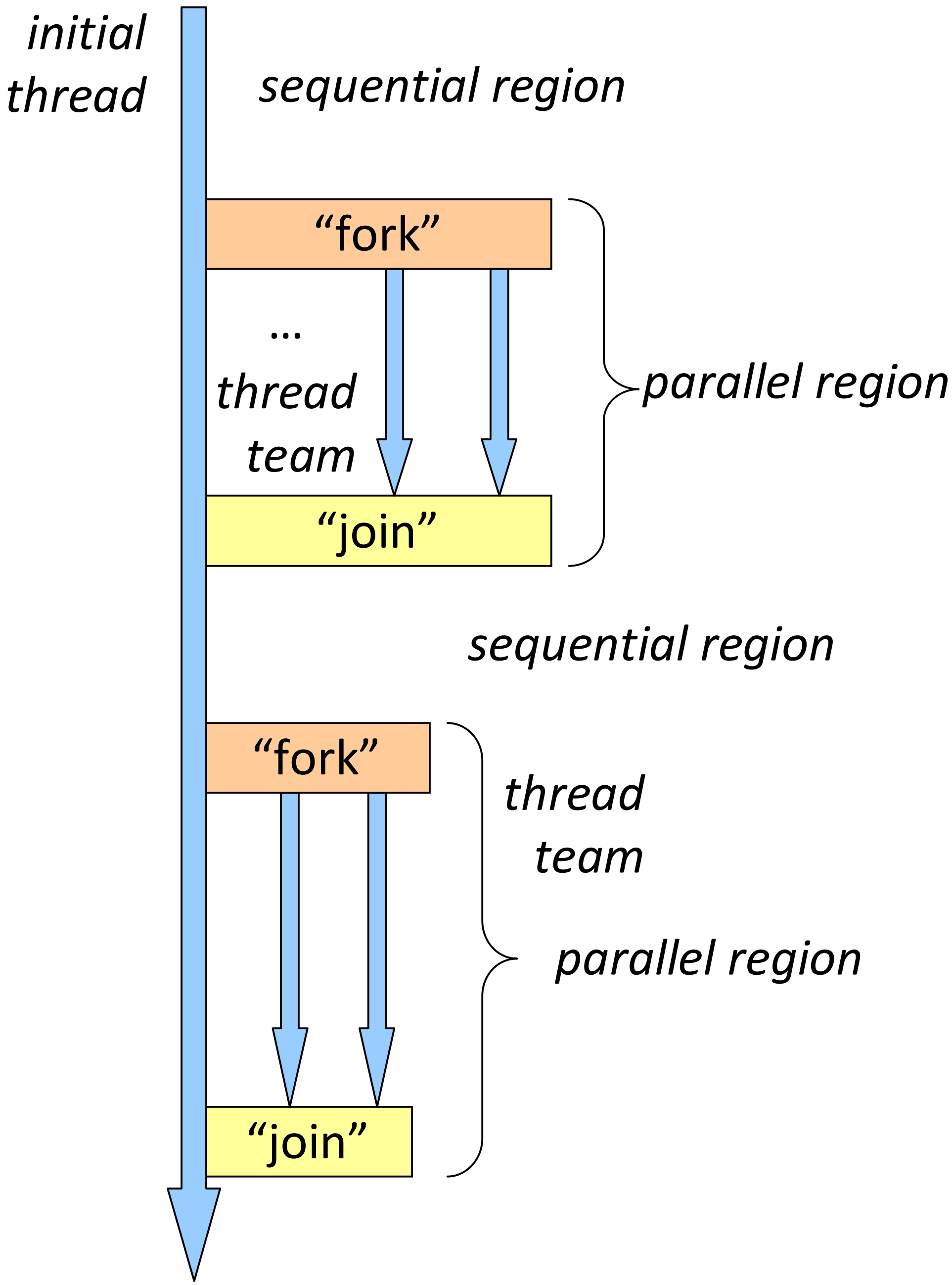
Execution:

```
>$ 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
  - 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

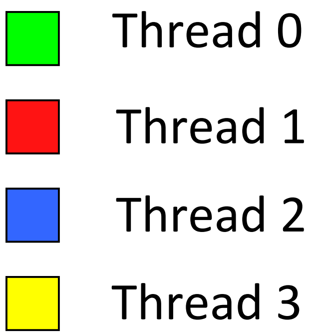
Loop Scheduling Strategies

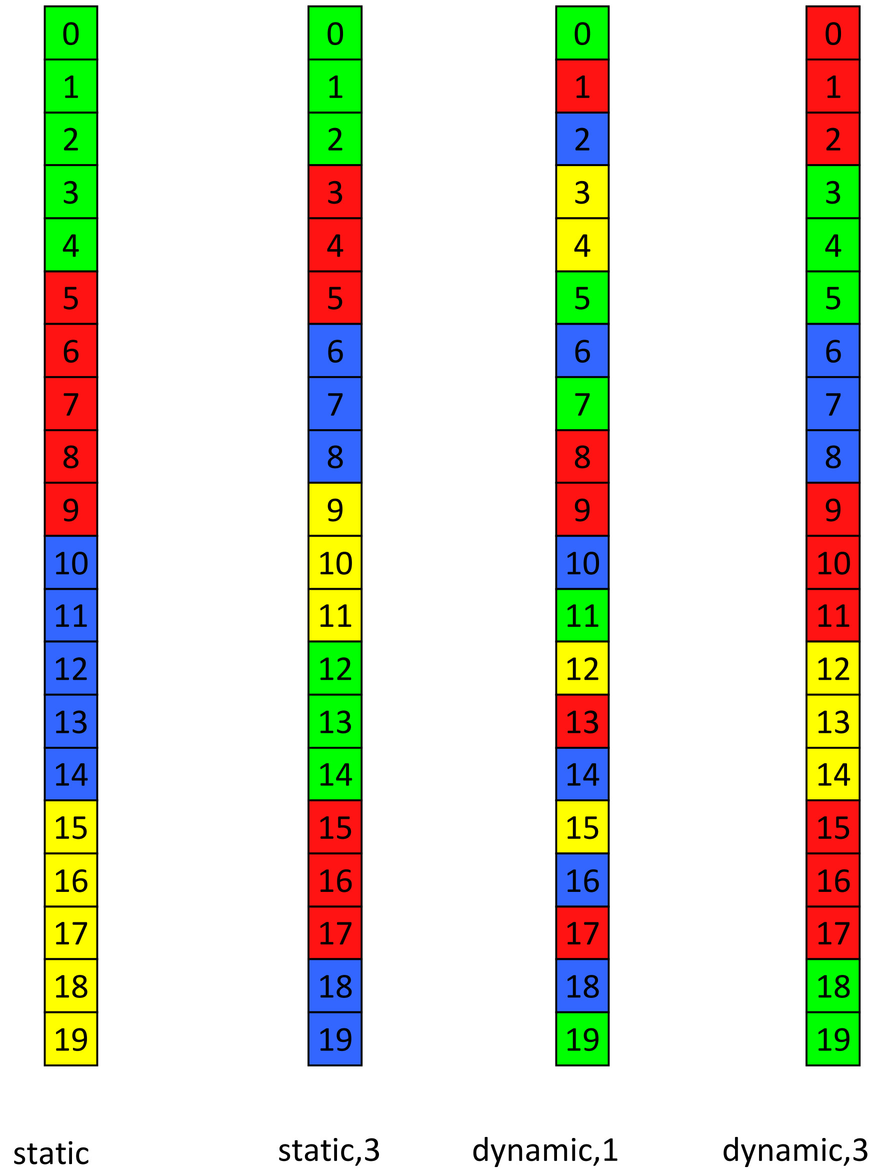
```
#pragma omp for schedule(type[, size])
```

- 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

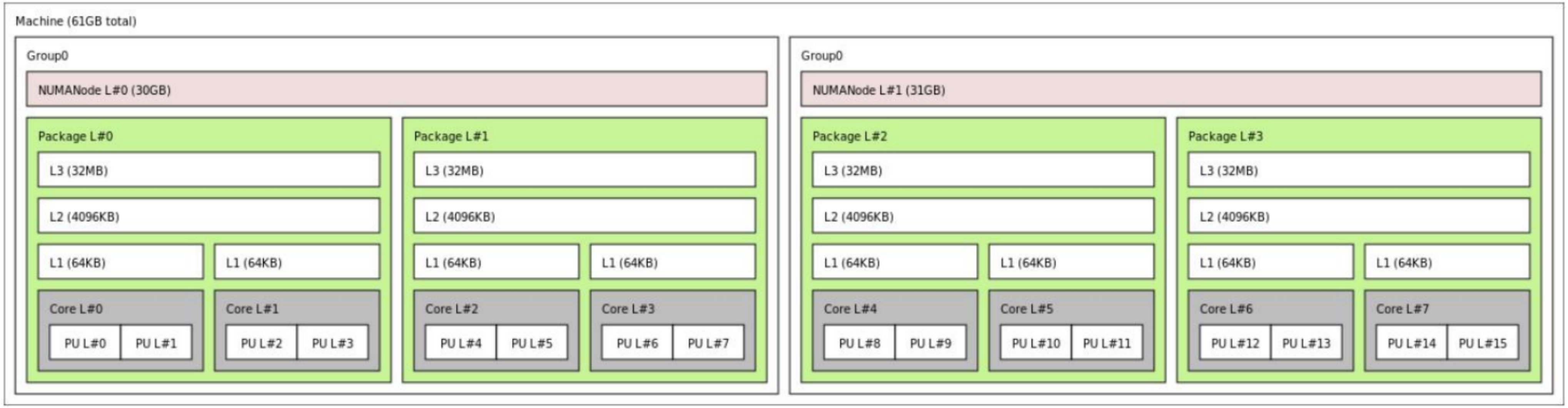
```
#pragma omp parallel
  {
//#pragma omp for schedule(static)
//#pragma omp for schedule(static,3)
//#pragma omp for schedule(dynamic,1)
#pragma omp for schedule(dynamic,3)
       for(i=0; i<20; i++)  {
         do_some_work(i);
       }
  }
```





Thread 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

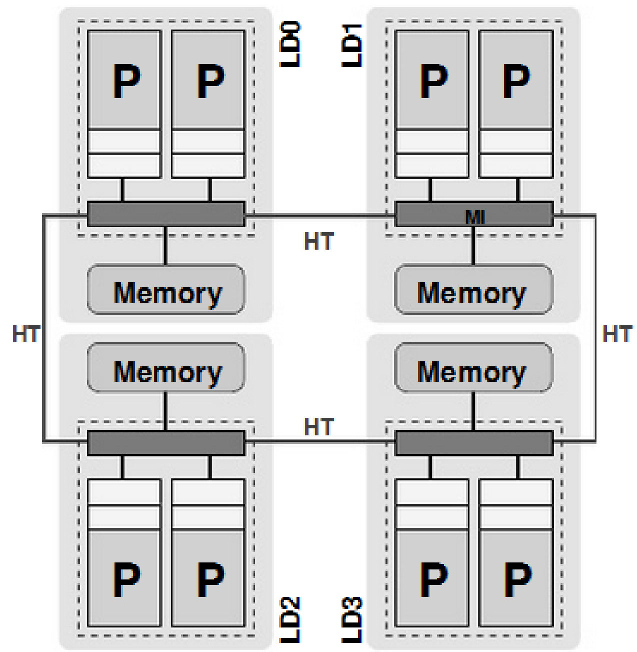
```
// initialize data strucutres
for(i=0; i<N; i++ ) {
  B[i]= . . .
  C[i]= . . .
  D[i]= . . .
}

#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"