


Introduction to HPC Programming

Shared Memory Programming, Part I

Survey Outcome

- Programming languages:

- 6x C/C++ and Python
- 2x C/++ only
- 1x Python only



today:
Simple Python example
+ extend C exercises

- Programming skills:

- 3x beginner
- 5x advanced
- 1x expert

- Parallel programming skills:

- 8x beginner
- 1x advanced

Good Practice: Optimal Usage of Resources

Plan to speed up existing code?

→ Upgrade “main branch”, never copy

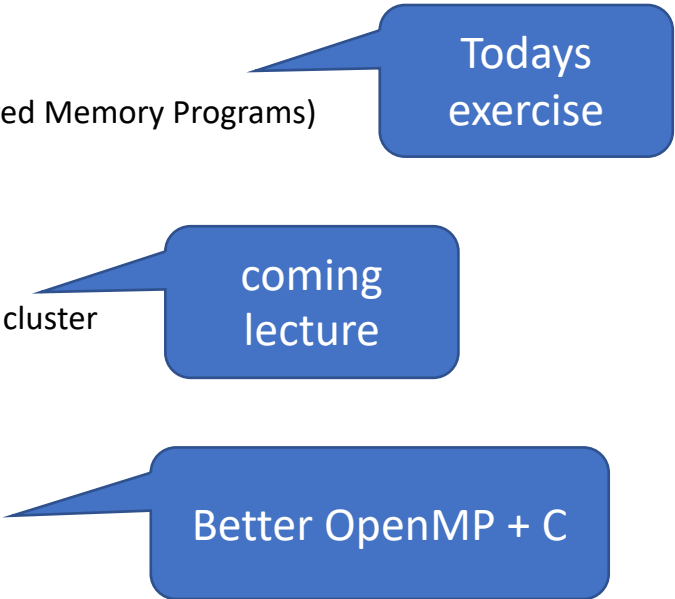
Order of optimisation:

1. Identify the bottlenecks, only tackle them.
2. Check for good/better algorithm
3. Check for already existing implementations (less coding time, better tested and documented)
4. Develop on single core
5. Extend your code to use multicore, but retain single core capability
6. Extend for multiple computers



Python!

Parallelisation in Python

- Multithreading package from Python
 - Numba
 - JIT compiler with parallelisation support (SMP, Shared Memory Programs)
 - Dask
 - Scale your Python code
 - Code rewrite necessary, scales from workstation to cluster
 - <https://dask.org>
 - Cython
 - optimising static compiler
 - OpenMP (for SMP) available
 - Usergroup: PyHEP – Python in HEP working group
 - The PyHEP working group brings together a community of developers and users of Python in Particle Physics, with the aim of improving the sharing of knowledge and expertise.
 - <https://hepsoftwarefoundation.org/workinggroups/pyhep.html>
- 
- The image contains three blue callout boxes with white text, each pointing to a specific item in the list:
- A box labeled "Todays exercise" points to the Numba sub-bullet "JIT compiler with parallelisation support (SMP, Shared Memory Programs)".
 - A box labeled "coming lecture" points to the Dask sub-bullet "Code rewrite necessary, scales from workstation to cluster".
 - A box labeled "Better OpenMP + C" points to the Cython sub-bullet "OpenMP (for SMP) available".

Numba in a nutshell

- Supports:
 - NumPy arrays, functions and loops
- Supports GPU (Nvidia CUDA, experimental: AMD ROC)
- <http://numba.pydata.org>
- **Unbeatable ratio: your time vs speedup**
- Function is compiled to machine code when called the first (~0.3 sec per call)
- Usage: „Numba decorated function“

```
from numba import jit
@jit(nopython=True)
def go_fast(a):
```
- Extra options:
 - `parallel = True` - enable the automatic parallelization of the function.
 - `fastmath = True` - enable fast-math behaviour for the function.



Needed in
exercise!

Introduction OpenMP



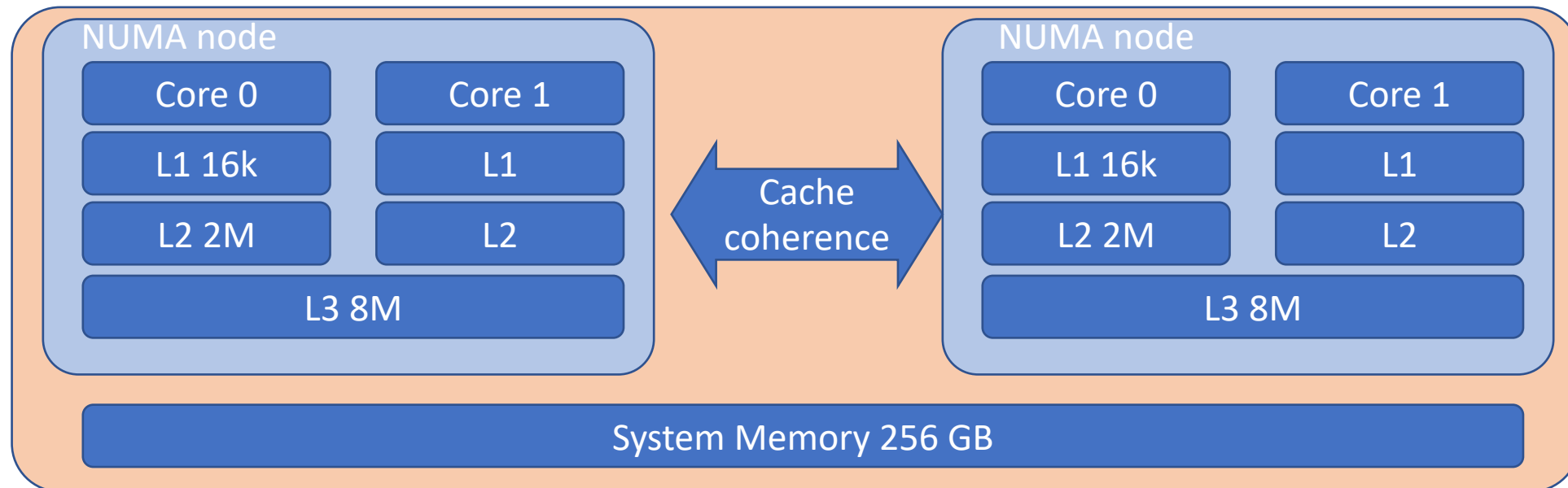
Introduction OpenMP



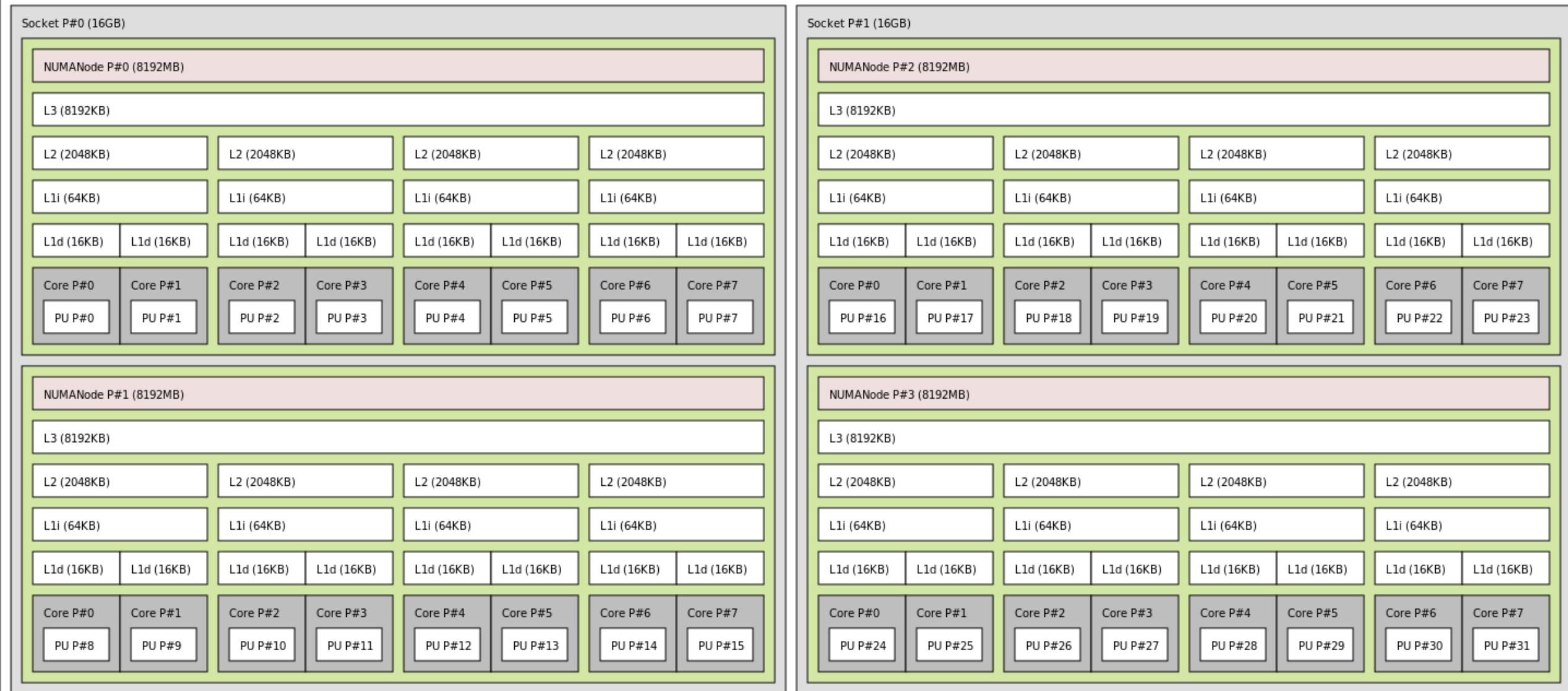
1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives
5. Data environment and combined constructs
6. Common pitfalls and good practice

Anatomy of a ccNUMA

- Different parallel processing concepts: pipelining, vector computing, multicore, ...
- Non-uniform memory access (NUMA): CPU design, where memory access time depends on the memory location relative to the core.
- cache coherent NUMA (ccNUMA, AMD: 2003, industry wide: 2011)



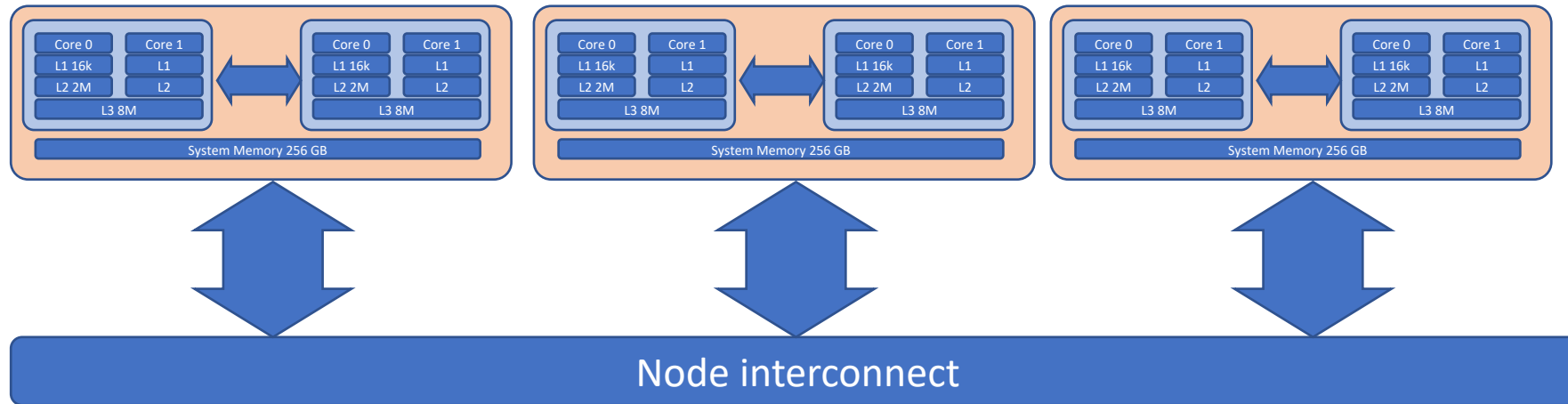
- ccNUMA uses inter-processor communication between cache controllers



- Output of hwloc tool: Topology of a ccNUMA Bulldozer server, 2 socket system

Anatomy of a cluster computer

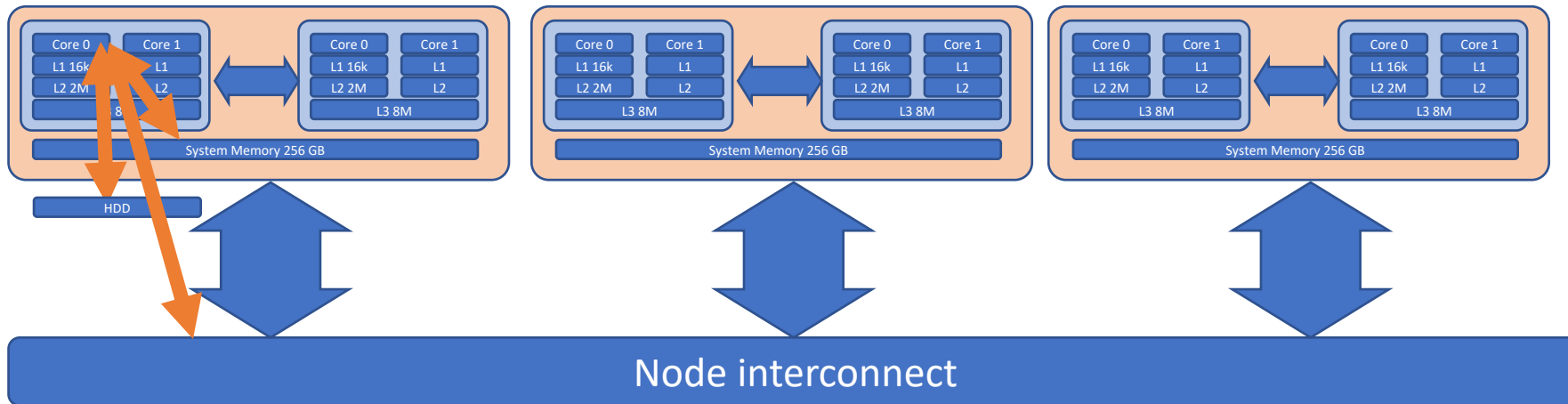
- $N * \text{ccNUMA} = \text{cluster}$:



- Fast lossless interconnect: OmniPath between ccNUMA nodes
- Inside a node: NUMA, ccNUMA
- Multiple nodes: Distributed memory parallelisation (DMP)

Anatomy of a cluster computer

- Latencies:



(all numbers are platform dependent)

Operation	min overhead in cycles
Hit L1 cache	1-10
Miss all caches	100
Page miss	100.000
(Data via interconnect)	1000 (1 μ s)

Introduction OpenMP



- Why OpenMP?
Relatively easy way: single-core program → multicore shared-memory
 - Released: 1997; widely and actively supported; currently version 4.5
 - Only Fortran and C
- Overview:
 - **OpenMP is a standard** programming model for **shared memory parallel programming**
 - **Set of compiler directives** and a few library routines
 - efficient
 - → less problems during runtime (on ccNUMA nodes!) compared to library based shared RAM synchronisation (Pthreads)
 - **Portable:** Large set of compilers and hardware architectures
 - Slow start, direct results: step-by-step introduction of parallelisation
 - Shared memory: results in good speedup
- Prerequisite
 - A error free single-core program

Implementation

Most modern compilers have support integrated, check their support

- Microsoft Visual C++ >2005,
 - Intel Parallel Studio (OpenMP 3.1 since version 13),
 - **GCC ab Version 4.2 (OpenMP 4.0 since version 5.0),**
 - Clang/LLVM (OpenMP 3.1 since Version 3.6.1),
 - Oracle Solaris Studio,
 - Fortran, C und C++ Compiler der Portland Group (OpenMP 2.5),
 - ...
- Homepage: openmp.org User group: compunity.org

Comparison OpenMP / MPI

OpenMP

- shared memory directives (compile time)
 - to define work decomposition
 - no data decomposition (data in shared memory)
- synchronisation is implicit

Possible speedup:

- memory limited: Total bandwidth / single core bandwidth = 4 (hardware dependent)
- CPU limited: Number cores (+ possible cache effects)
- storage limited: do not use

MPI (Message Passing Interface, later this course)

- software library (run time)
- user defines:
 - distribution of work & data
 - communication (when and how)

Possible speedup:

- Per node limits: see OpenMP
- RAM/CPU limited: utilisation of N nodes
- Storage limited: ? (use node local scratch)

Where to start?

Optimise your gain = speedup / work!

1. Try trivial parallelisation.
2. Parallelise your code with OpenMP, concentrate on time-consuming sections
3. Introduce MPI
large problems, work in team, check about available resources first (man power + hardware)
(different, if you join a group with existing MPI-code)
4. Hybrid programming: OpenMP + MPI
to gain the last 10% speedup



Easiest approach to multi-core programming in C

Glimpse: 1st OpenMP code

- OpenMP focusses on parallel loops with independent iterations

Compiler directive
#pragma

```
int main() {  
    int in[100], out [100];  
  
    for (int i=0; i<100; i++) {  
        out[i] = MyLongFunc(in[i]);  
    }  
}
```

```
int main() {  
    int in[100], out [100];  
    #pragma omp parallel for  
    for (int i=0; i<100; i++) {  
        out[i] = MyLongFunc(in[i]);  
    }  
}
```


Introduction OpenMP



1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives
5. Data environment and combined constructs
6. Common pitfalls and good practice

OpenMP: Programming Model

- shared memory model
- Distribution of work between multiple threads (“workers”)
 - Variables can be
 - Shared among all threads
 - Duplicated for each thread
 - Communication between threads through “barriers”
- Unintended data sharing → race conditions (undefined behaviour) or dead lock
- To avoid race conditions: use synchronisation

read&write access to the same data by multiple threads

Due to different scheduling of threads between runs

OpenMP: Execution Model

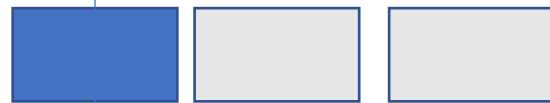
- Single Thread



- Parallel Region



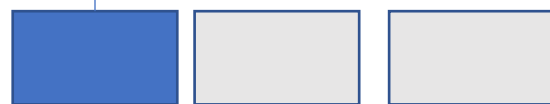
- Single Thread



- Parallel Region



- Single Thread



end

Thread 0 (Master)

N threads

Thread 0 (Master)

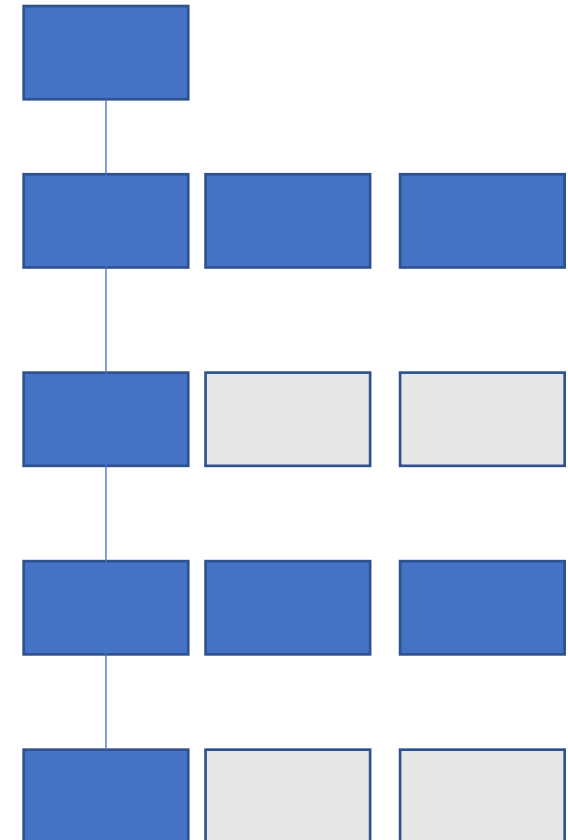
N threads

Thread 0 (Master)

Launch of multiple threads

Execution Model Description

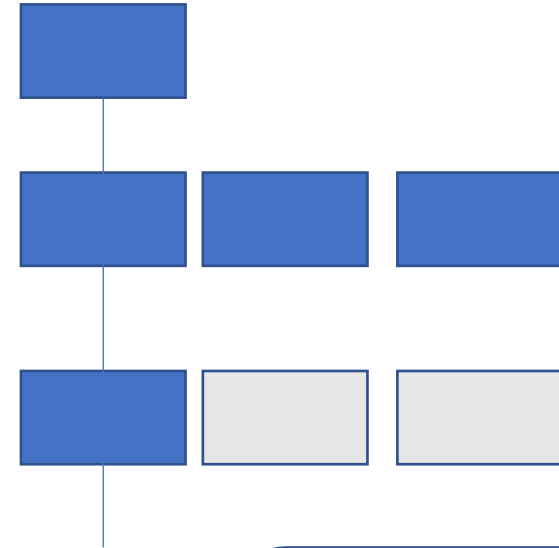
- Begin execution as a single process (master thread)
- Fork-join of parallel execution
 1. Start of 1st parallel construct: Master thread creates N threads
 2. Completion of a parallel construct: threads synchronise (implicit barrier)
 3. Master thread continues execution
- At next parallel construct: work balancing with existing threads



OpenMP Parallel Region Construct + Syntax

```
#pragma omp parallel [clause [, clause]]  
    block  
// omp end parallel
```

- *block* = to be executed by multiple threads in parallel. Each code executes the same code.
- Clause can be (“data scope”):
 - `private (list)` ← variables in list private to each thread & not initialised, standard for loop variables
 - `shared (list)` ← variables in list are shared among all thread, standard
 - `firstprivate`, `lastprivate`, `threadprivate`, `copyin`, `reduction`



Good practice:
Always declare
all variable
either in private
or shared to
avoid surprises
(race conditions)

Compiler Directives

- #pragma directives

```
#pragma omp directive_name [clause [, clause]]
```

- Conditional compilation

```
#ifdef _OPENMP  
    block, eg. printf("OpenMP sup.ted.");  
#endif
```

- Include file for library routines with compiler directives:

```
#ifdef _OPENMP  
#include <omp.h>  
#endif
```

- **Why this (good practice)?**

- Keep your code single-core and multi-core
- Do not copy your code (fork), always modify the main branch!
→ after years of development: main branch developed and your code is parallel but old!

Environment Variables

- OMP_NUM_THREADS
 - Sets the number of threads
 - Set before execution, not during compilation
 - Bash: `export OMP_NUM_THREADS=16`
csh: `setenv OMP_NUM_THREADS 16`
- OMP_SCHEDULE
 - Applies only to do/for and parallel do/for directives that have the schedule type RUNTIME
 - Sets schedule type and chunk size for all such loops
 - Bash: `export OMP_SCHEDULE="GUIDED,4"`
csh: `setenv OMP_SCHEDULE "GUIDED,4"`

Runtime Library

(Most of the OpenMP functionality arises from the compiler during compilation, but...)

- Query, runtime and lock functions comes from omp.h library
#include <omp.h>
(Implementation dependent)
- int omp_get_num_threads()
returns the current number of threads (N) executing the parallel region from which it is called
- int omp_get_thread_num()
return the thread number (0..N-1).
Master thread is always 0
- wall clock timers: (similar to MPI_WTIME in MPI)
double omp_get_wtime();
provides elapsed time in a thread
(needs not to globally consistent!)

```
# ifdef _OPENMP
double wt1,wt2;
wt1=omp_get_wtime();
# endif

//heavy computing

# ifdef _OPENMP
wt2=omp_get_wtime();
printf("wct %12.4g sec\n", wt2-wt1);
# endif
```



Exercise 1: Parallel region

Learning objectives:

- Runtime library calls
- Conditional compilation
- environment variables

Steps:

1. Copy the skeleton files from the course web page (see next slide)
2. Compile and run as serial program
3. Compile as openmp program (-fopenmp with cc) and run with different numbers of threads
4. Compare the run times between serial and openmp program

Computational task

- Computational intensive function

Karl Weierstraß; 1841:

$$\pi = \int_{-\infty}^{\infty} \frac{dx}{1+x^2} = 2 \cdot \int_{-1}^1 \frac{dx}{1+x^2}$$

- Run a analysis with several runs, do statistics

Set up your workbench

- Read the latest hints online:
<https://gitlab.rlp.net/pbotte/learnhpc/tree/master/openMP>

Basic concept:

- Connect 2 times to Mogon2 / HIMster2 via SSH
 - 1) `srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs bash`
 1. Use the first SSH connection for editing (`gedit`, `vi`, `vim`, `nano`, `geany`)
 2. Use the second connection for the interactive compiling and execution on the nodes (no analysis on the head node!):
`OMP_NUM_THREADS=4 ./pi`
- Download the files:
`git clone https://gitlab.rlp.net/pbotte/learnhpc.git`
 - Check for directory: `openMP/exercise1/`

Hints:

- Check compiler version: `cc -v`
- Run: `OMP_NUM_THREADS=4 ./pi`
or `export OMP_NUM_THREADS 4`
- Possible to check reservation with: `squeue -u $USER`


Exercise 2: Parallel region

Learning objectives:

- Parallel regions, private and shared clauses

Steps:

1. Use the code from exercise 1, and compile as openmp program (-fopenmp with cc) and run with number of threads=4
2. Add a parallel region that prints the rank and the number of threads for each thread
→ expectation: undefined sequence of printf statements. No parallelisation of computation.
3. Try to create a race condition by (define variables outside of block!):
 1. First writing into registers:
`myrank = omp_get_thread_num();`
`num_threads = omp_get_num_threads();`
 2. And replace
`# pragma omp parallel private(myrank, num_threads) →`
`# pragma omp parallel`
4. Add a #else directive that prints if the program was not compiled with OpenMP



Compiler dependent
result / several runs
needed

Numba exercise

Learning objectives:

- Installation
- Compilation
- Parallelisation

Steps:

1. Details on how to copy and run:
Follow the instructions on: <https://gitlab.rlp.net/pbotte/learnhpc/tree/master/numba>
2. Compare the run times between standard python, compiled and parallel python program. Make sure, you have decent statistic and avoid the effect on first compilation.