HPC Programming

Trivial Parallelisation and OpenMP, Part I

Peter-Bernd Otte, 23.10.2018

Survey Outcome

- Programming languages: c, c++, python
- software used: Open source, but some are large (PandaROOT, BES3)
- Single-core (only one exception)
- Storage: GB to 10 TB (no problem, access pattern?)
- RAM: MB to 20 GB (1-3GB/core optimal)
- Runtime: seconds to hours (seconds to short: compare to set up costs of job)
- Number of program calls per analysis: 2-10k
- Boundaries: CPU, RAM (both great for HPC!), storage

Trivial parallelisation

Trivial Parallelisation

- Submit a single core job multiple times
- Quick and often only solution for large software blobs (large packages used in collaborations)
  - No principal difference compared to running on your desktop computer
- limits:
  - required RAM (3GB/core)
  - licensees (Mathematica, max 10 concurrent usages in university for such uses cases)
  - shared scratch in node (200GB)
  - parallel filesystem (loading at start, writing back results) max. → 10-100 jobs in parallel
- Hint: use job arrays

- Disadvantage: No control over speedup (this is gambling)

Sample Submit Script

1. Define and reserve resources (nodes with RAM)
2. Once allocated, run the executables as defined or interactively

More examples
https://mogonwiki.zdv.uni-mainz.de/dokuwiki/slurm_submit

```
#!/bin/bash
#SBATCH -o /home/pbotte/test/myjob.%j.%N.out
#SBATCH -D /home/pbotte/test/
#SBATCH -J MyJobName
#SBATCH -A m2_him_exp          ← account (NOT your account)
#SBATCH -N 1                   ← Request number of nodes
#SBATCH --partition=himster2_exp ← partition
#SBATCH --mem-per-cpu=1G
#SBATCH --mail-type=FAIL
#SBATCH --mail-user=pbotte@uni-mainz.de
#SBATCH --time=8:00:00         ← wall time (>run time)

module load gcc/6.3.0
echo TEST...
srun myExecutable
```

Submit with: `sbatch submitScript.sh`

Batch System: SLURM

- Introduction and docu:
  - https://mogonwiki.zdv.uni-mainz.de/dokuwiki/slurm_submit
  - https://slurm.schedmd.com/tutorials.html

- account to use: m2_himkurs / m2_him_exp
- Reservation: himkurs / empty for himster2 usage
- Submit into partition: parallel / himster2_exp
  - srun --pty -p parallel -A m2_himkurs -N 1 --reservation kurstest bash -i
- Check what is running: squeue -u pbotte
  - 1184615_79 parallel N203r001 **pbotte** R 1:00:40 52 z[0367-0386,0403-0413,0430-0450]
  - SSH login into your occupied nodes possible: eg ssh z0367

Batch System: SLURM

- Submit script for later execution (batch mode)
  - `sbatch --partition=himster2_exp`

- Create job allocation and start a shell to use it (interactive mode)
  - `salloc -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs`

- `srun`: Create a job allocation (if needed) and launch a job step (typically MPI job)
  - `srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs bash -i`

- `sattach`: Connect stdin/out/err for an existing job

Support: Use of our HPC resources

- you plan to use HIMster2 or Mogon2 for your analysis?
- Feel free to ask for support
- We will discuss your problem and find some starting point for you.

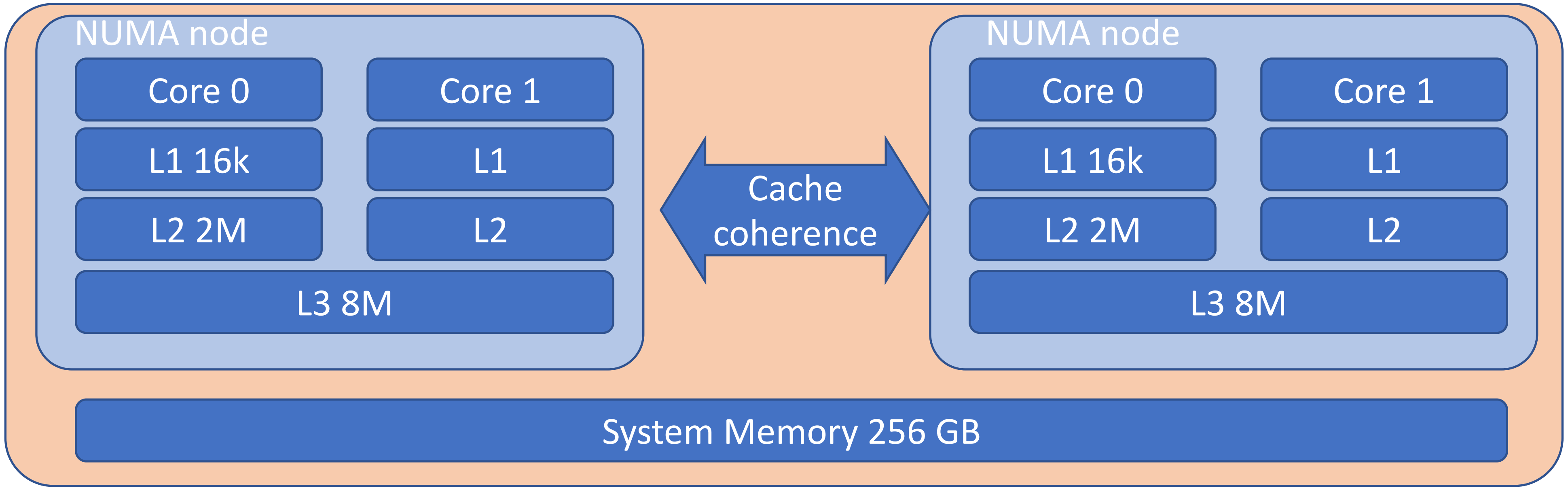
Introduction OpenMP

Introduction OpenMP

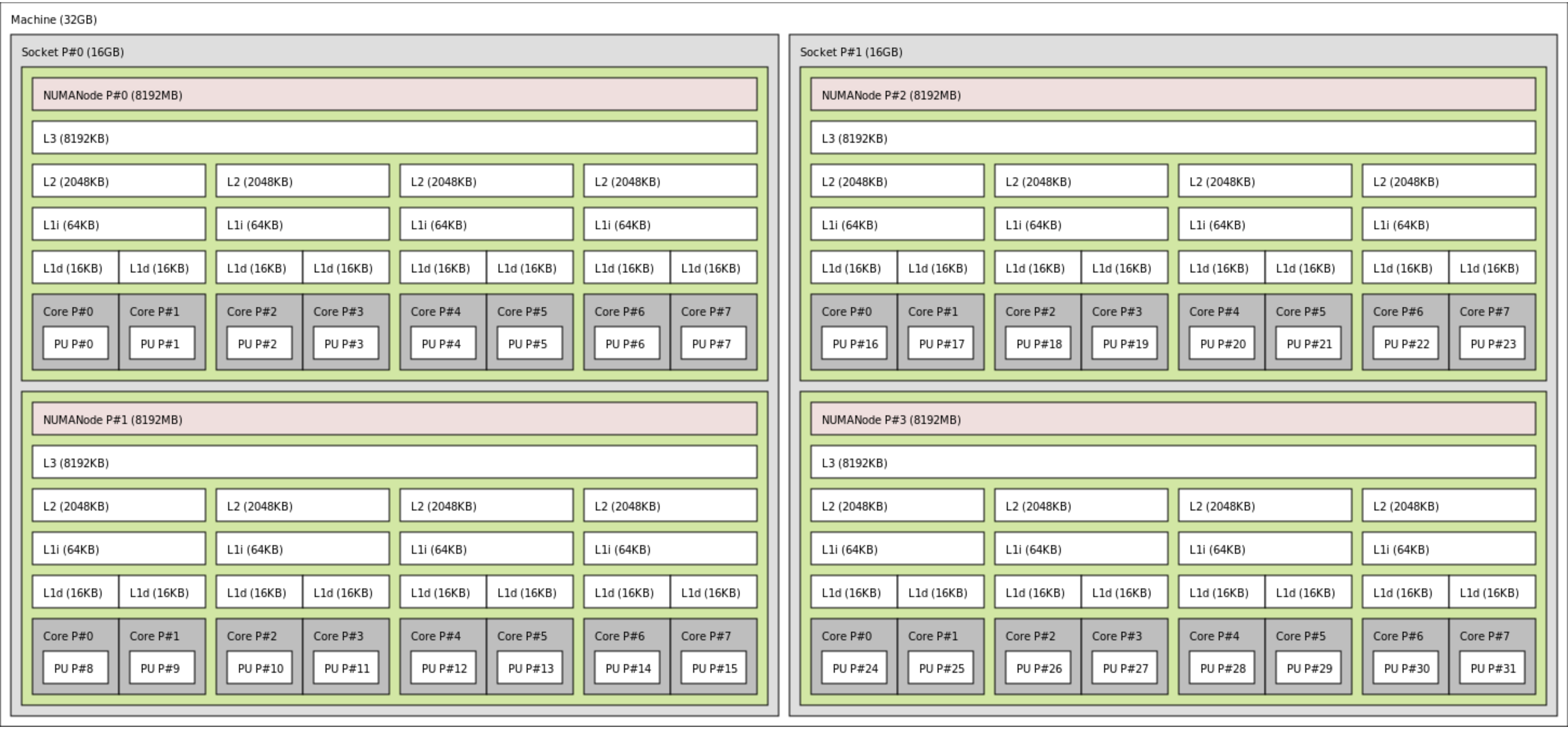
1. Hardware Anatomy
2. Motivation
3. Programming and Execution Model
4. Work sharing directives
5. Data environment and combined constructs
6. Common pitfalls and good practice

Anatomy of a ccNUMA

- Different parallel processing concepts: pipelining, vector computing, multicore, …
- cache coherent Non-Uniform Memory Access (ccNUMA, AMD: 2003, industry wide: 2011)



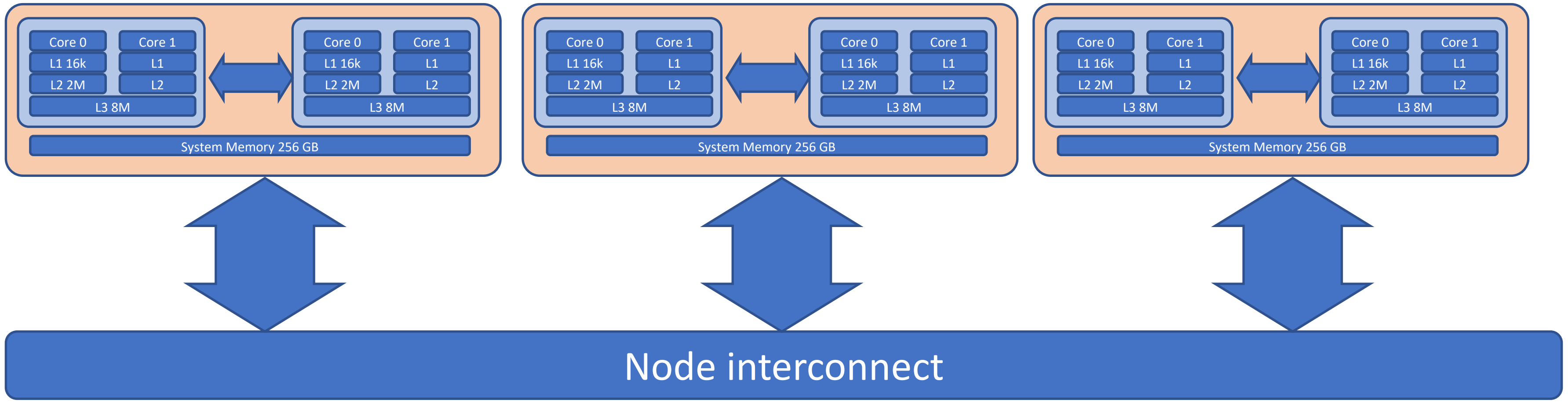
- ccNUMA uses inter-processor communication between cache controllers



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

Anatomy of a cluster computer

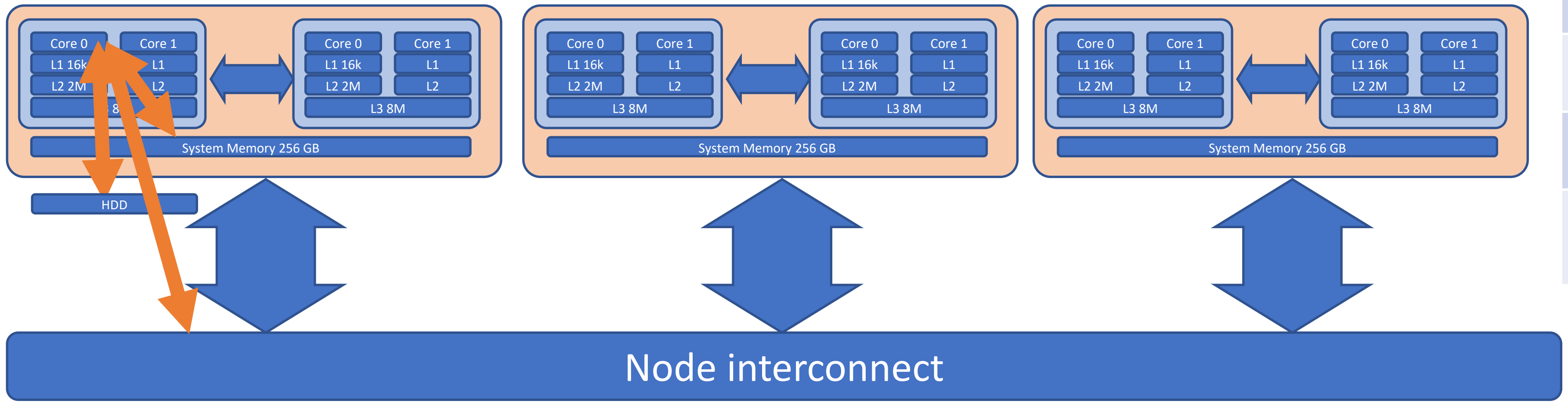
- N * ccNUMA = cluster:



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

Anatomy of a cluster computer

- Latencies:



Operation	min overhead in cycles
Hit L1 cache	1-10
Miss all caches	100-300
Page miss	100.000
(Data via interconnect)	1000 (1µs)

(all numbers are platform dependent)

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

Glimpse: 1st OpenMP code

- OpenMP focusses on parallel loops with independent iterations

```
int main() {
   int in[100], out [100];

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

Compiler directive
#pragma

```
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

OpenMP: Programming Model

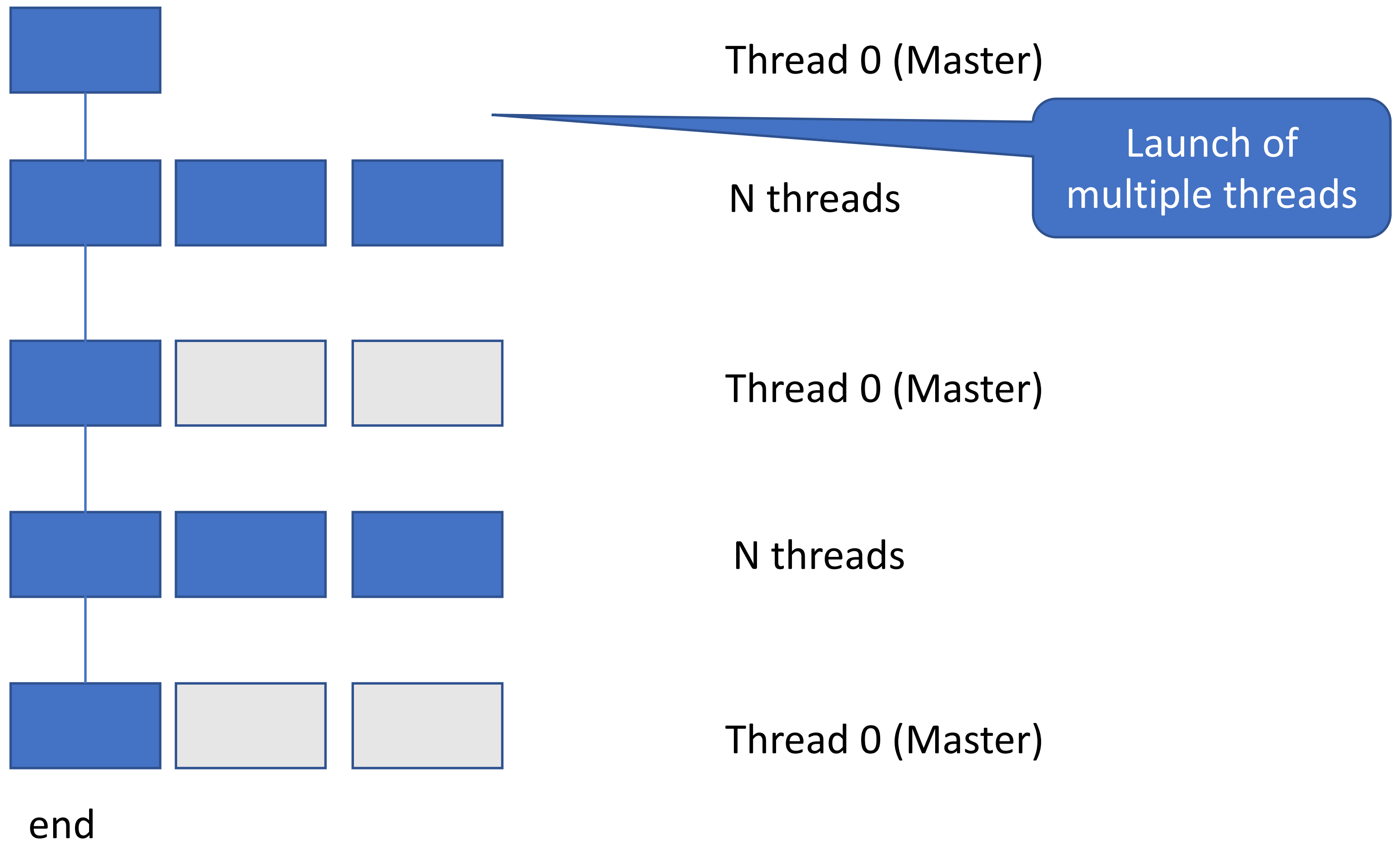
- shared memory model
- Distribution of work between multiple threads ("workers")
  - Variables can be
    - Shared among all threads
    - Duplicated for each thread
  - Communication between threads through "barriers"
- Unintended data sharing → race conditions (undefined behaviour) or dead lock
-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

read&write access to the same data by multiple threads

Due to different scheduling of threads between runs

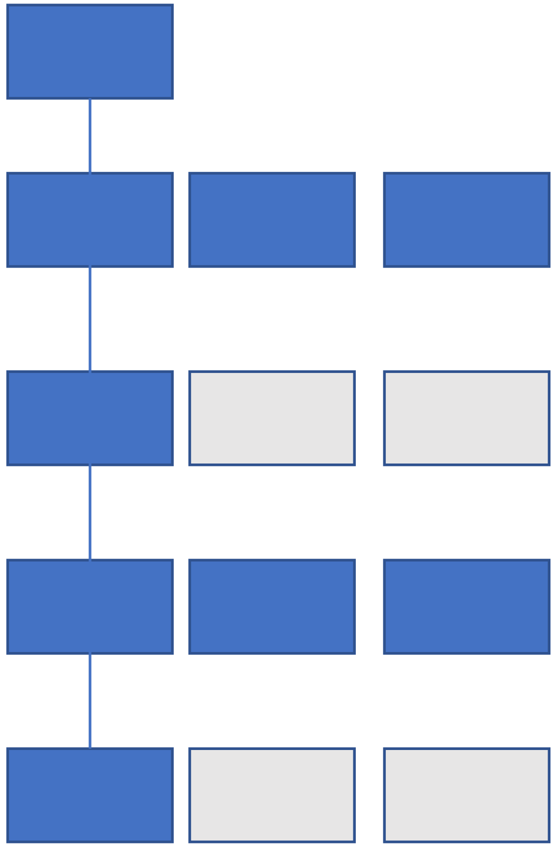
OpenMP: Execution Model

- Single Thread
- Parallel Region
- Single Thread
- Parallel Region
- Single Thread



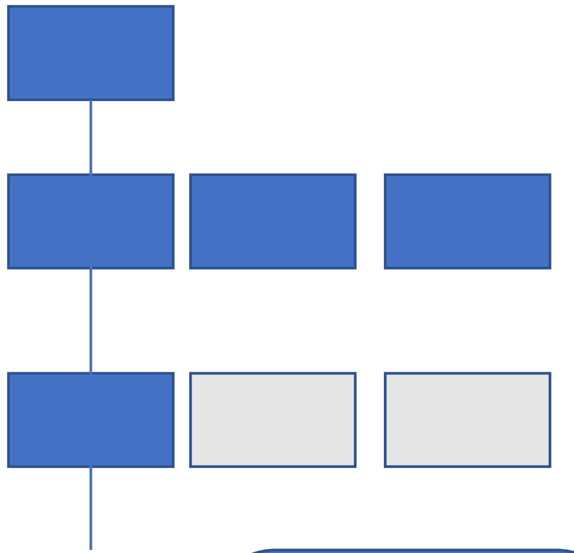
Execution Model Description

- 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
// emp 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_THEADS=16
    csh: setenv OMP_NUM_THEADS 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:

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

- Run a analysis with several runs, do statistics

Set up your workbench

- Connect 2 to Mogon2 / HIMster2 via SSH
  srun --pty -p parallel -N 1 --time=02:00:00 -A m2_himkurs --reservation=himkurs -N 1 bash –l
  1. Use the first SSH connection for editing (gedit, vi, vim, nano, geany) and
     compiling: cc -fopenmp -o hello hello.c
  2. Use the second connection for the interactive execution on the nodes (no analysis on the head node!):
     OMP_NUM_THREADS=4 ./hello

- Download the files via: wget https://www.hi-mainz.de/fileadmin/user_upload/IT/lectures/WiSe2018/HPC/files/02.zip && unzip 02.zip

Hints:

- Check compiler version: cc -V
- Run: OMP_NUM_THREADS=4 ./hello
  or export OMP_NUM_THREADS 4
- Possible to check reservation with: squeue -u USERNAME

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:
   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

   Compiler dependent result
4. Add a #else directive that prints if the program was not compiled with OpenMP