Mitglied der Helmholtz-Gemeinschaft

JÜLICH FORSCHUNGSZENTRUM

Parallel Computing

November 20, 2017

W.Homberg

Why go parallel?

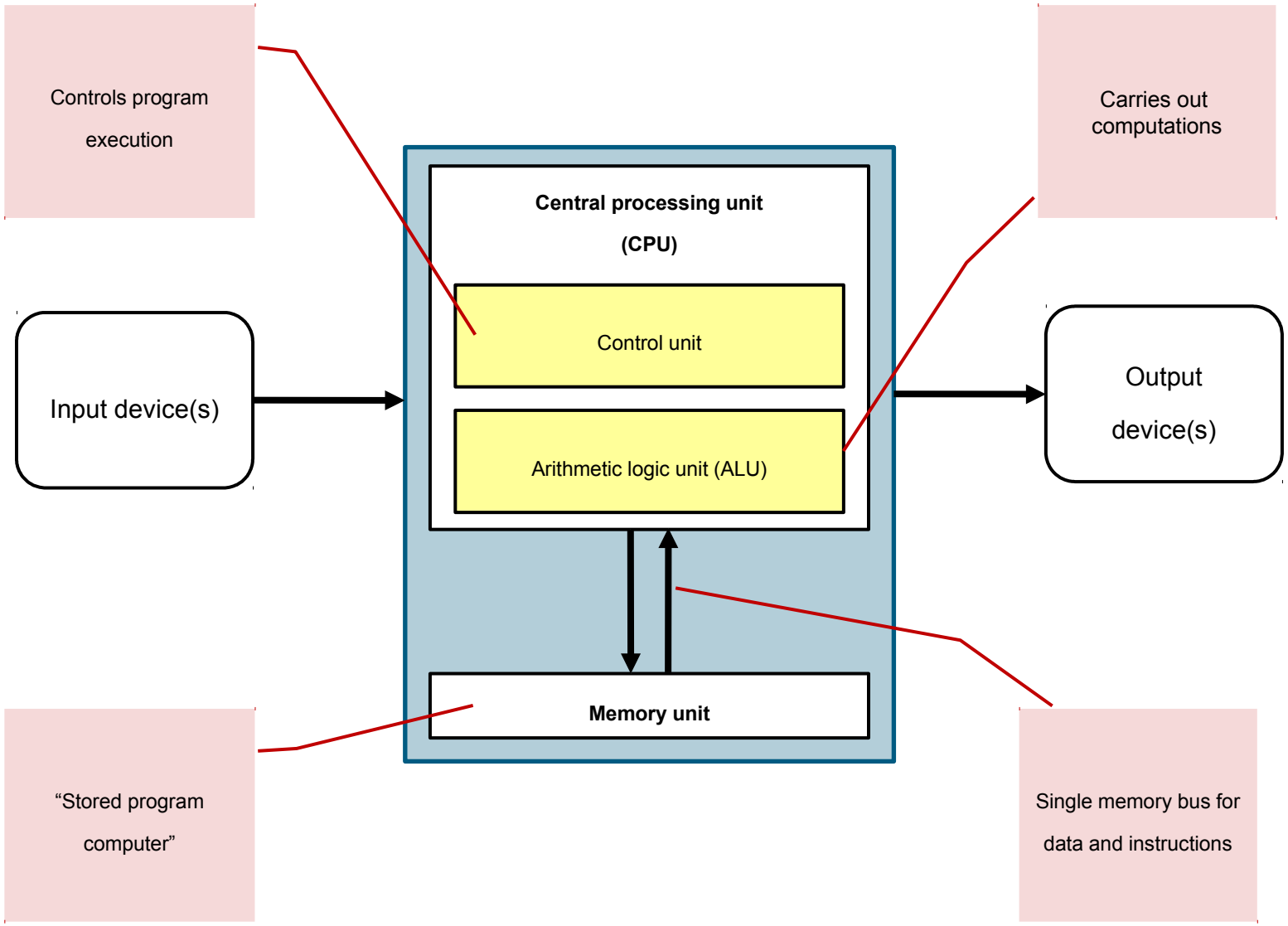
- Problem too large for single node
  - Job requires more memory
  - Shorter time to solution essential
- Better performance
  - More instructions per second (compute bound)
  - Better memory bandwidth (memory bound)
  - Lower operating costs (energy to solution)

Parallel hardware: CPU

- Every CPU is a parallel device (since Pentium II)
  - Vector units (SSE, AVX, …)
  - Independent floating point and integer units
  - Multiple hardware threads (2 on Jureca per core)
  - Multiple cores per CPU (12 on Jureca)

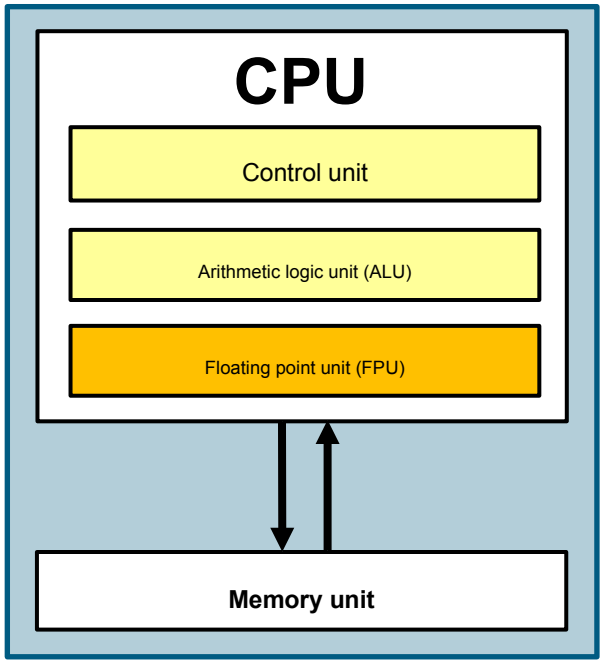
- Multiple sockets (2 on Jureca)

Von Neumann Architecture

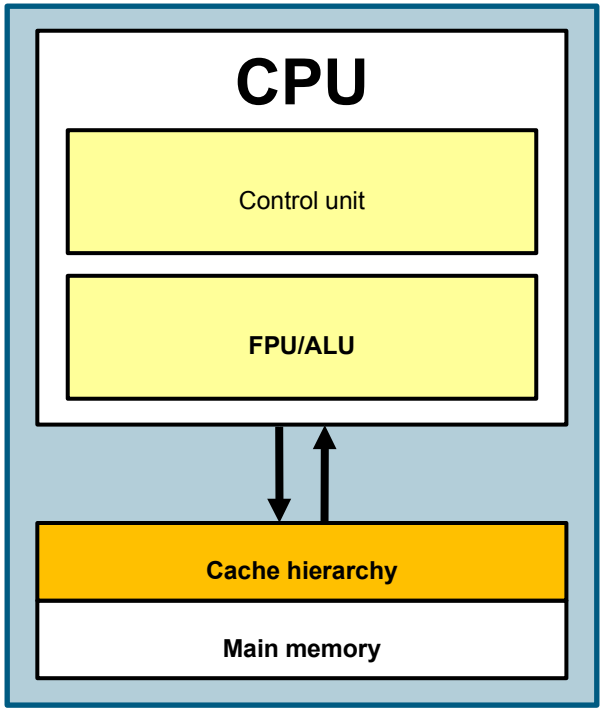


Enhancements

Computing with real numbers



Enhancing memory access



Computational power [flops]

- Floating point operations (scalar addition or multiplication) per second

Memory bandwidth [bytes/second]

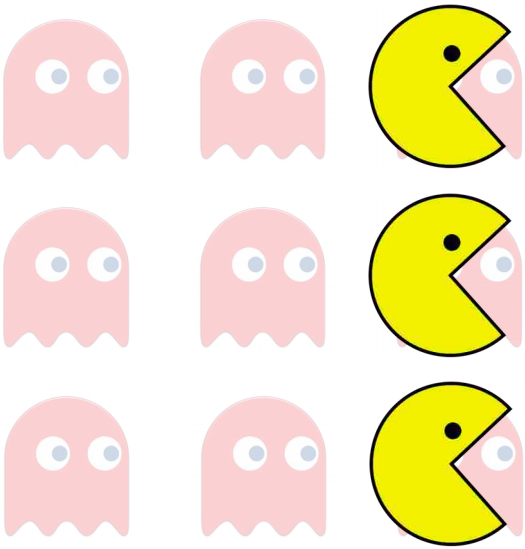
- Number of bytes transferred in every second

Latency [seconds]

- How fast the result of an operation (computation/memory access) is available

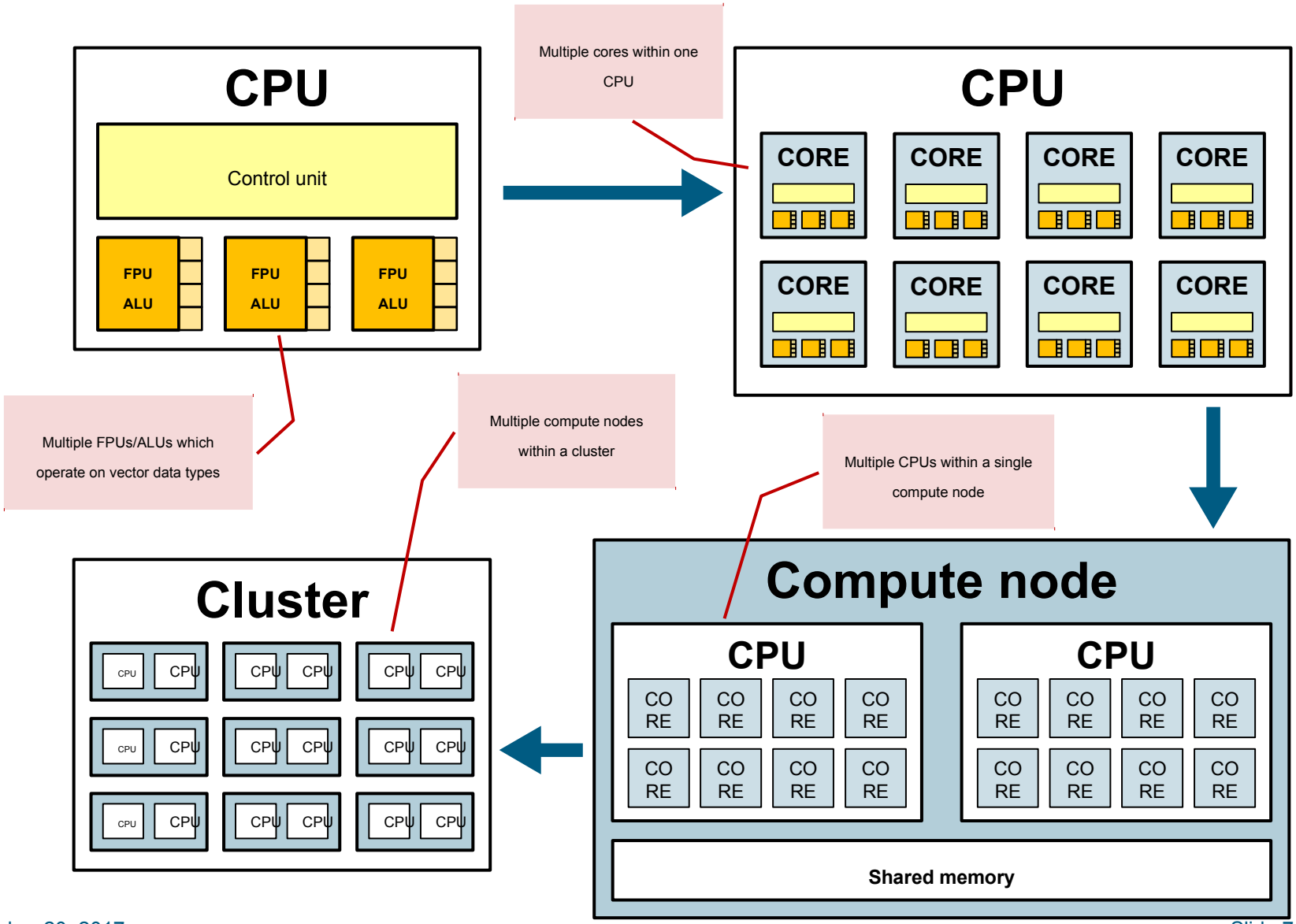
Going Parallel (I)

1 Pacman eats 9 ghosts in **3 seconds**…



3 Pacmans eat 9 ghosts in **1 second**…

Going Parallel (II)



Computation of Max. Theroretical Perf. (R_{peak})

Example: Intel Xeon E5-2600v3 Haswell CPU (JURECA)

Calculation (SP) per Core

R_{peak} = #FPUs (= 2)
* #Ops per FPU per cycle (= 2)
* vector length of the operands (= 8)
* processor clock (= 2.5 GHz)
= 80 GFLOPs

- 12 processor cores (SP): 960 GFLOPs
- 12 processor cores (DP): 480 GFLOPs

JuHYDRA – GPU Server

MEGWARE MiriQuid GPU-Server, 64 GB Memory, Peak 16/5 TFlops SP/DP

NVIDIA Tesla K20X (1x Kepler GK110)

- Flops: 3.94 / 1.31 TFlops SP / DP
- Compute Units: 14
- Processing Elements: 192 / CU
- Total # PEs: 14 x 192 = 2688
- CU frequency: 732 MHz
- Memory: 6 GB (ECC) – 384bit
- Memory frequency: 5.2 GHz
- Memory bandwidth: 250 GB/s
- Power consumption: 235 W

INTEL Xeon E5-2650 Processor (Sandy Bridge)

- Flops: 0.128 / 0.064 TFlops SP / DP
- Compute Units: 8 (Cores)
- Processing Elements: 4 / Core
- Total # PEs: 8 x 4 = 32
- Core frequency: 2.0 GHz (2.4 turbo mode)
- Power consumption: 95 W

AMD FirePro S10000 (2x Tahiti)

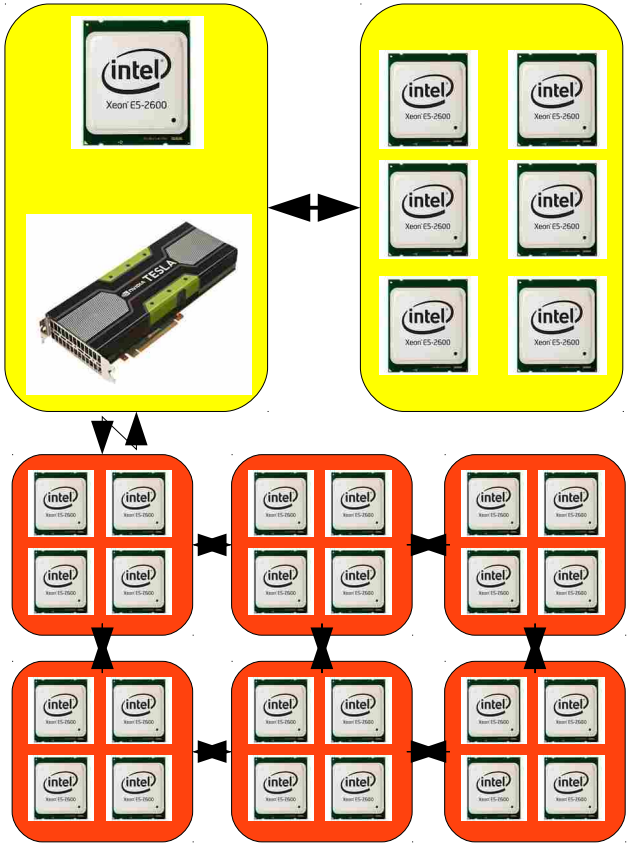
- Flops: 5.91 / 1.48 TFlops SP / DP
- Compute Units: 2x 28
- Processing Elements: 64 / CU
- Total # PEs: 2x 28 x 64 = 3584
- CU frequency: 825 MHz
- Memory: 6 GB (ECC) – 384bit
- Memory frequency: 5.0 GHz
- Memory bandwidth: 2x 240 GB/s
- Power consumption: 375 W

INTEL Xeon Phi (MIC) Coprocessor 5110P

- Flops: 2.02 / 1.01 TFlops SP / DP
- Compute Units: 60 (Cores)
- Processing Elements: 16 / Core
- Total # PEs: 60 x 16 = 960
- Core frequency: 1.053 GHz
- Memory: 8 GB
- Memory bandwidth: 320 GB/s
- Power consumption: 225 W

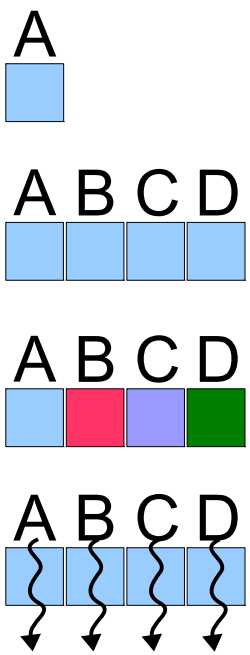
Heterogeneous systems

- Different devices within a node (CPU + GPU)
- Different nodes within a cluster
- Different clusters within a grid



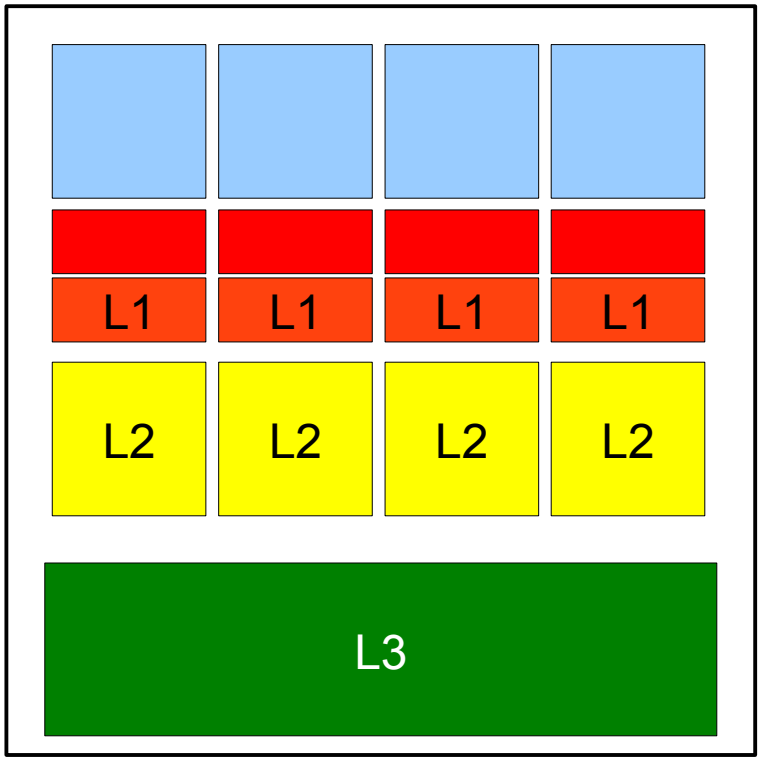
Flynn's characterization

- SISD
  Single Instruction, Single Data
- SIMD
  Single Instruction, Multiple Data
- MIMD
  Multiple Instructions, Multiple Data
- SIMT
  Single Instructions, Multiple Threads



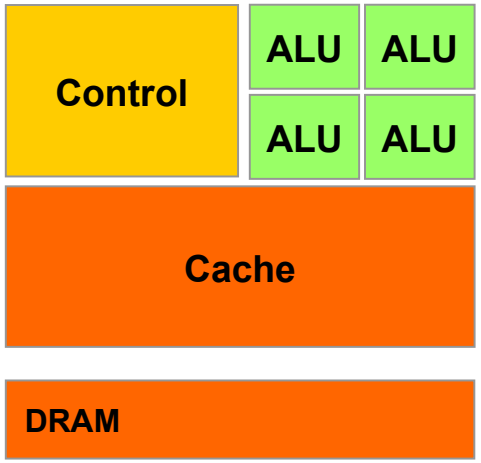
Memory

- Registers (per core)
- L1 cache (per core)
- L2 cache (per core/shared)
- L3 cache (shared)
- Main memory

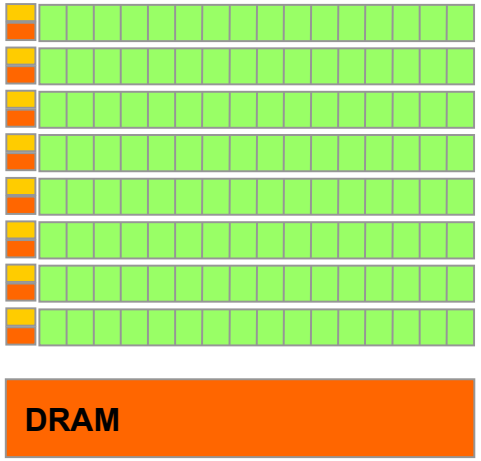


Latency and throughput

- Get your calculations done as quickly as possible (CPU)



- Perform calculations on a lot of data in parallel (GPU)



GPU Computing

Fig.: Nvidia

Kepler GPU (GK110):

- Each green square = single FPU
- Each FPU **(about 2700)** available for a different thread
- Overall, GK110 can handle **more than 30000 threads** simultaneously…
- …and even better, in our program we can send **billions of threads** to the GPU!

- GPU programming: *Thinking in large arrays of threads*
  - Proper organization of threads incl. data sharing very important
- Many APIs for many different programming languages available:
  - CUDA (only NVIDIA; e.g., runtime C++ API)
  - OpenCL (independent of hardware platform, also for CPUs)
  - OpenACC (for NVIDIA and AMD GPUs; based on compiler directives like OpenMP)

Parallel Computing (I)

Amdahl's Law

Runtime on single processor:

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization}$$

Runtime on *P* processors:

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization}$$

Speedup:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)}$$

Serial fraction γ:

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)}$$

Runtime on *P* processors (expressed with γ):

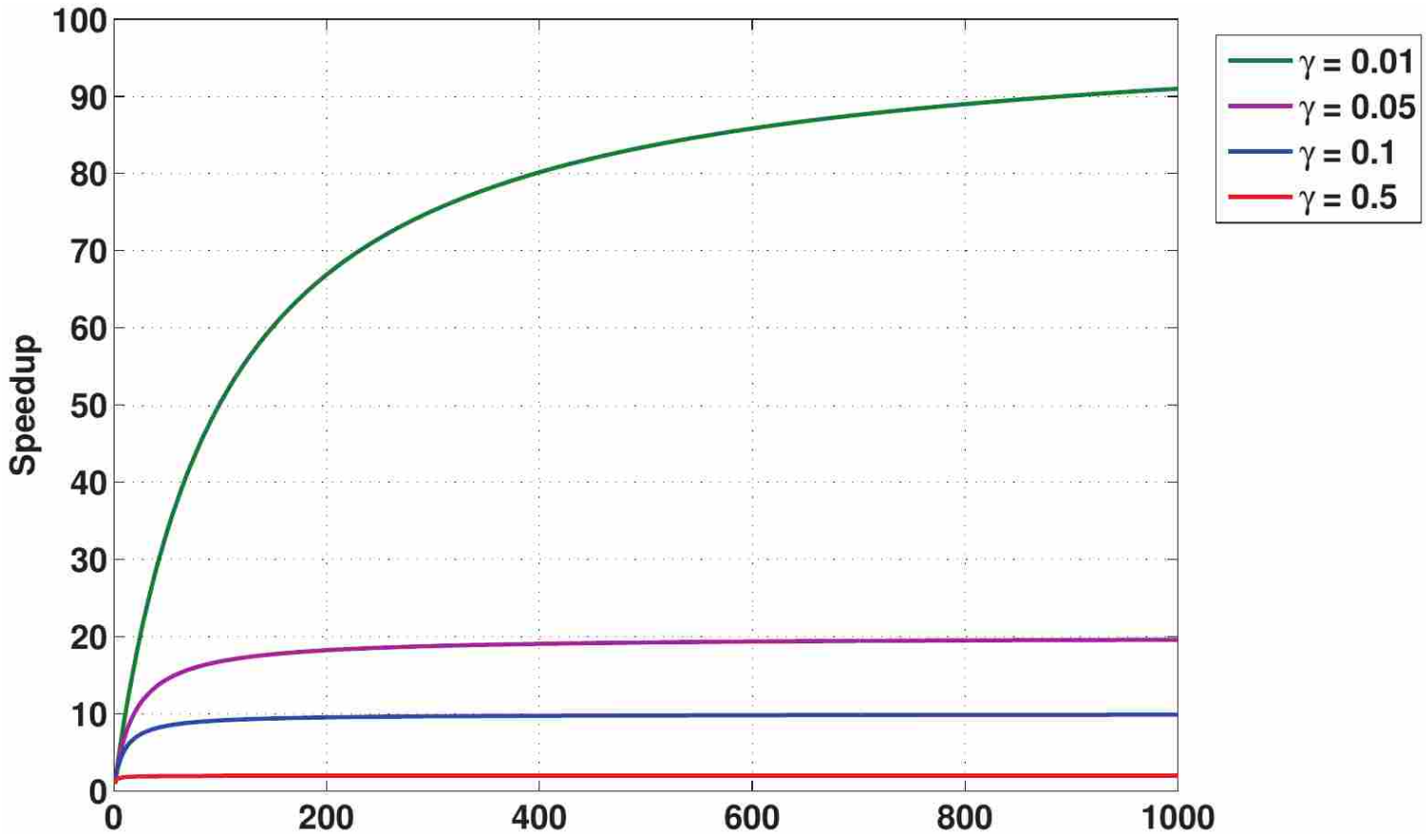
$$T_{total}(P) = \gamma T_{total}(1) + \frac{(1-\gamma) T_{total}(1)}{P}$$

Amdahl's law:

$$S(P) = \frac{T_{total}(1)}{\gamma T_{total}(1) + \frac{(1-\gamma) T_{total}(1)}{P}} = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

Parallel Computing (II)

Amdahl's Law



➔ Using highly parallel computers (and accelerators like GPUs) only makes sense for programs with minimal serial code fractions!

Parallel Computing (III)

Gustafson-Barsis's Law

… speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size.

Gustafson, John L. "Reevaluating Amdahl's law." Communications of the ACM 31.5 (1988): 532-533.

Amdahl's Law: problem size fix, minimise time-to-solution

Gustafson's Law: execution time fix, increase # processors

Serial fraction γ

Runtime on *P* processors in parallel: $\gamma + (1 - \gamma) = 1$

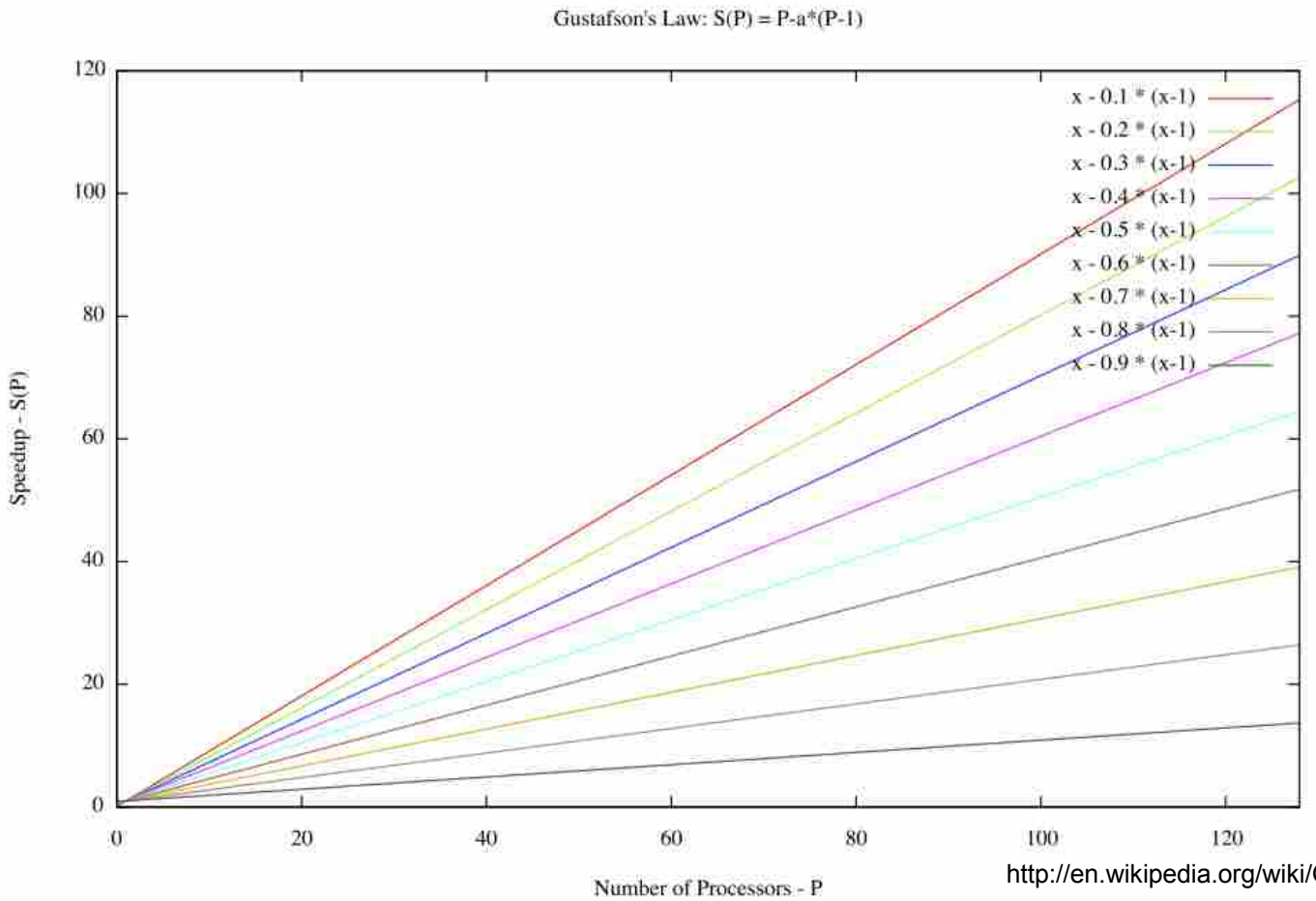
Runtime on 1 (hypothetical) processor in serial: $\gamma + P(1 - \gamma)$

Speedup: $S(P) = \gamma + P(1 - \gamma) = P - \gamma(P - 1)$

→ a sufficient large problem can be parallelised efficiently

Parallel Computing (IV)

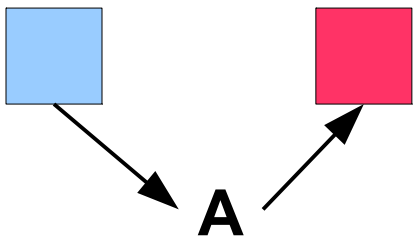
Gustafson-Barsis's Law



http://en.wikipedia.org/wiki/Gustafson's_law

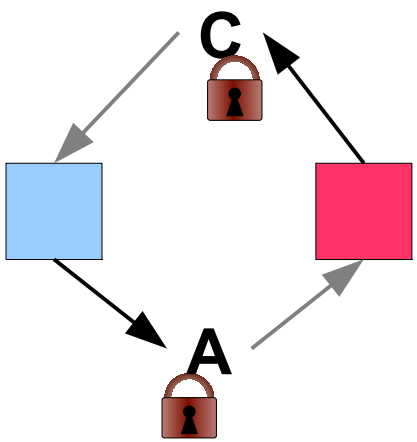
a sufficient large problem can be parallelised efficiently

Issues and pitfalls: race condition



- Blue writes **A**, red reads **A**.
  → avoid if possible
  → must not do this with different thread blocks

Issues and pitfalls: deadlock



- Blue creates lock to protect **A**, red has to wait.
- Red writes to **C** and protects **C** with a lock. Blue wants to read from **C** → deadlock

Issues and pitfalls: lack of locality

- Data on other core/GPU
- Data on host
- Data on disk
- ...

- No memory coalescing
- Bank conflicts

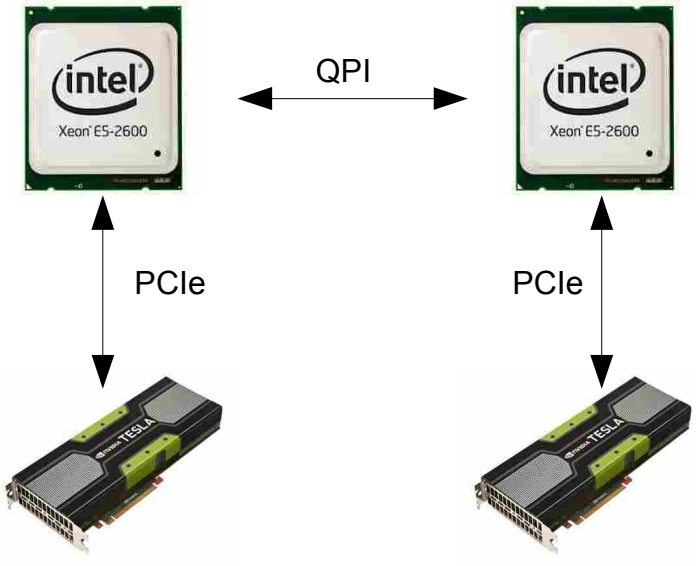
Issues and pitfalls: load imbalance

- Unused cores
- Adaptive refinement
- Lack of parallel work

Issues and pitfalls: overhead

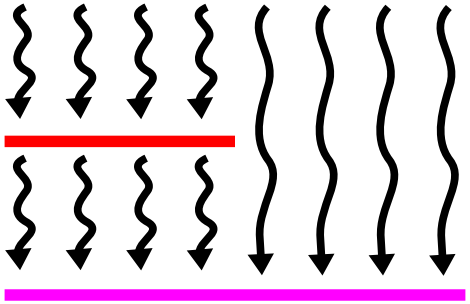
- Run time of kernel too short
- Computational intensity too low
- Too much communication

- IO: Done by CPU
  - can be done in parallel with compute kernel on GPU



Overhead: Synchronization

- Only threads within a threadblock/work-group can be synchronized
- Global synchronization is done with kernel calls
- Atomics can sometimes be used to avoid synchronization



Steps to parallelise an application: Profiling

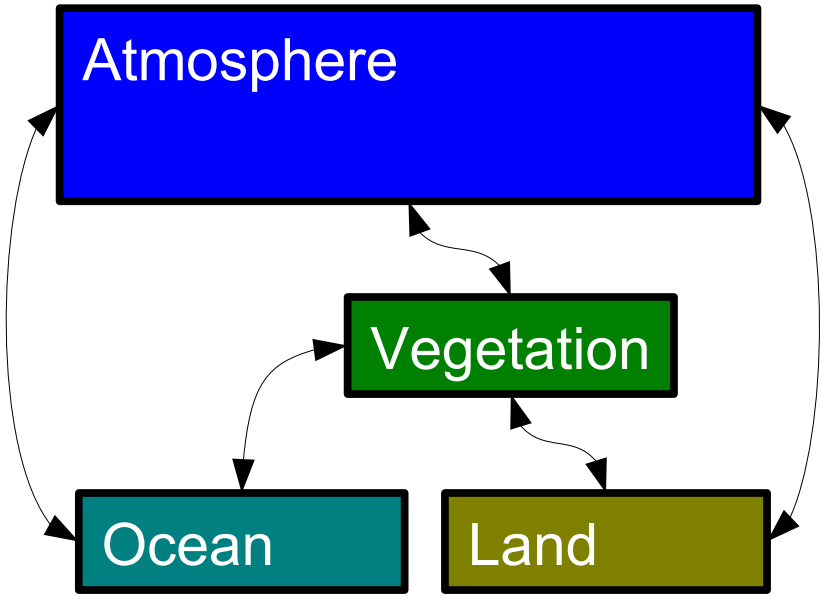
- Is there a hotspot
- How long does it take
- Does it have a high computational intensity
- How much data has to be moved to/from GPU

Steps to parallelise: Algorithm design

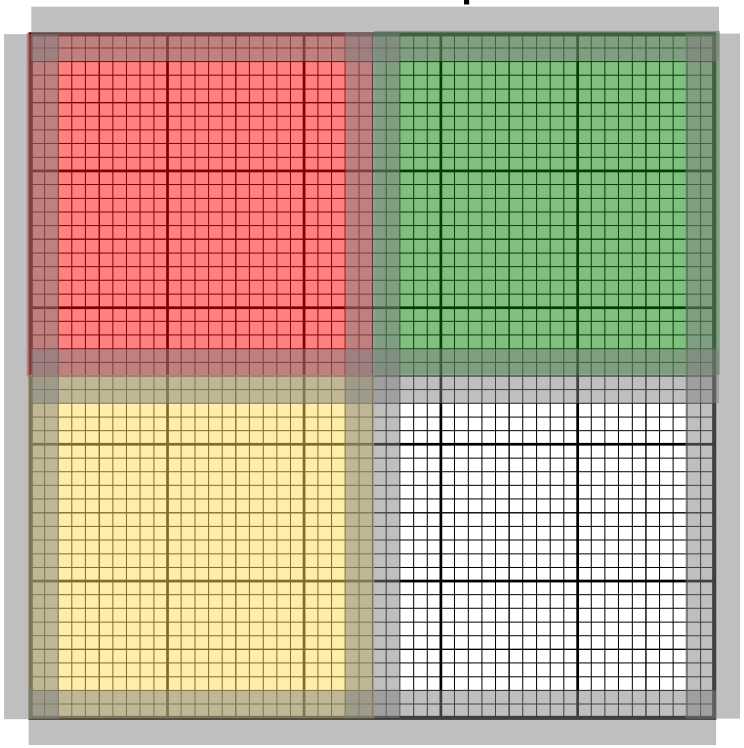
- Splitting up your work
- Embarrassingly parallel
- Static/dynamic load balancing
- Minimimze overhead
- Minimize latencies
- ...

Decomposition

Functional decomposition



Domain decomposition



Climate Simulation

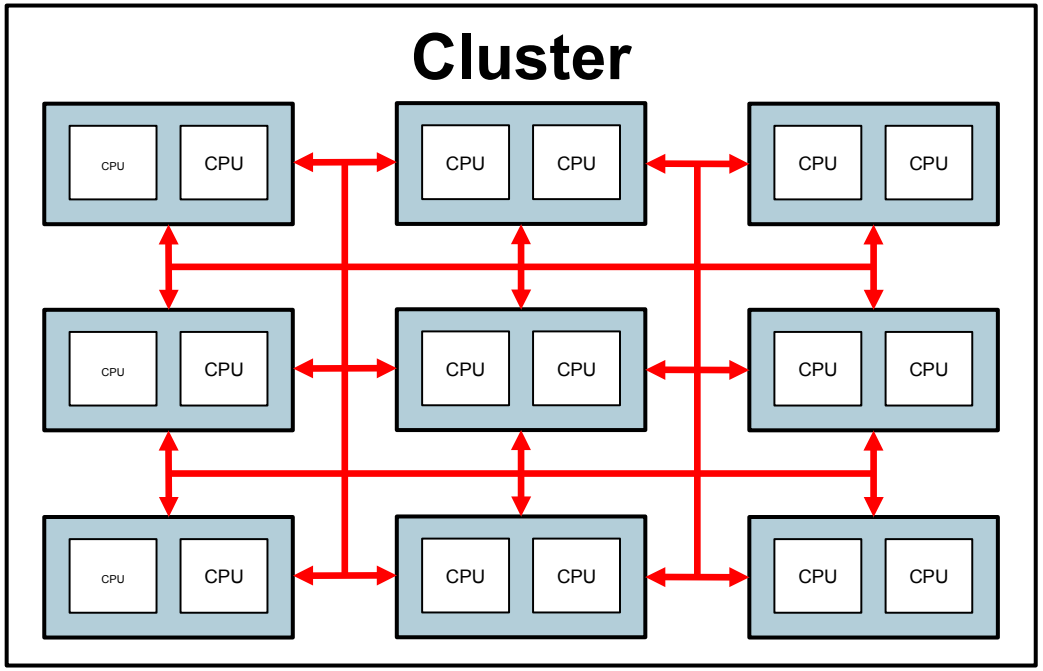
Parallel programming models

- Coarse-grained parallel
  - MPI
  - OpenMP
  - OpenACC
- Fine-grained parallel
  - CUDA
  - OpenCL

- MPI + OpenMP + OpenACC + CUDA
  - … and many other combinations possible

Message Passing Interface (MPI)

Parallelism between compute nodes



Main application of MPI: Communication between compute notes within a cluster

- Standardized and portable message-passing system
  - **Basic idea:** Processes (= running instances of a computer program) exchange messages via network
- Defined by the “MPI Forum” (group of researchers from academia and industry)
  - Release of MPI 1.0 in 1994
- Many implementations for C and Fortran (and also other prog. lang.) available (library, daemons, helper programs)

Message Passing Interface (MPI)

Example C Code

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int myrank, message_size=50, tag=99;
    char message[message_size];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0) {
        MPI_Recv(message, message_size, MPI_CHAR, 1, tag,
                 MPI_COMM_WORLD, &status);
        printf("received \"%s\"\n", message);
    } else {
        strcpy(message, "Hello, there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 0, tag,
                 MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Init MPI processing

Determine rank of process

Receive message on rank 0

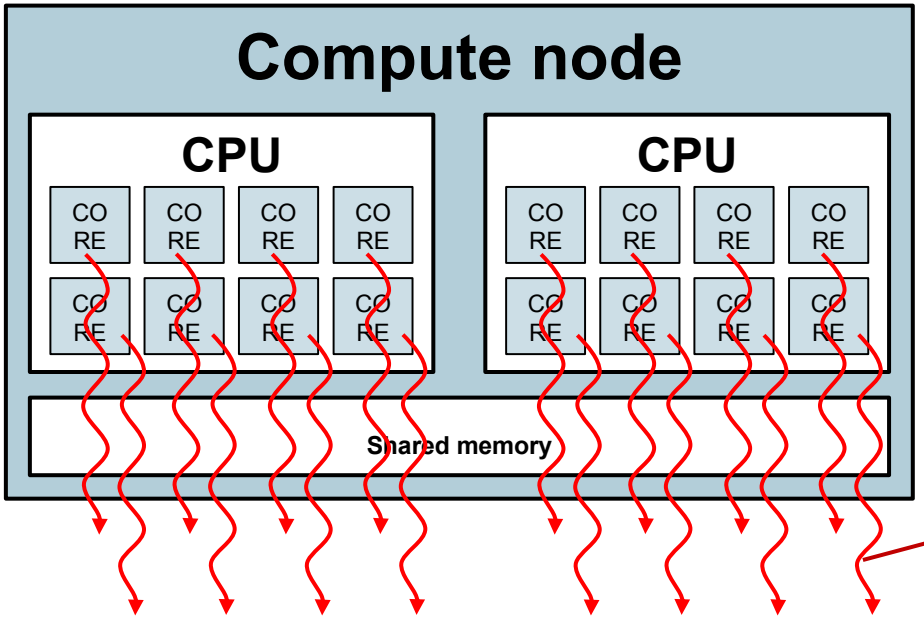
Send message on rank 1

Finalize MPI processing

Code example: Wikipedia

OpenMP

Parallelism between Cores



Main application of OpenMP: Programming interface for multiprocessing within a shared memory system via threads

Multiple threads during execution of a single program

(thread = lightweight sub-process within a process)

- Application programming interface (API) for shared memory multiprocessing (multi-platform: hardware, OS)
  - **Basic idea:** OpenMP runtime environment manages threads (e.g., creation) as required during program execution
    - ➔ *Makes live easy for the programmer*
- Defined by the nonprofit technology consortium "OpenMP Architecture Review Board" (group of major computer hardware and software vendors)
  - Release of OpenMP 1.0 in 1997
- Many implementations for C, C++, and Fortran (library, OpenMP-enabled compilers)

OpenMP
Example C++ Code

```
#include <iostream>
using namespace std;
#include <omp.h>

int main(int argc, char *argv[])
{
  int th_id, nthreads;
  #pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
    #pragma omp critical
    {
      cout << "Hello World from thread " << th_id << '\n';
    }
    #pragma omp barrier

    #pragma omp master
    {
      nthreads = omp_get_num_threads();
      cout << "There are " << nthreads << " threads" << '\n';
    }
  }

  return 0;
}
```

Open parallel section (threads will run in parallel on same code)

Get ID of local thread

Critical section: Only one thread at a time is allowed to write to the screen

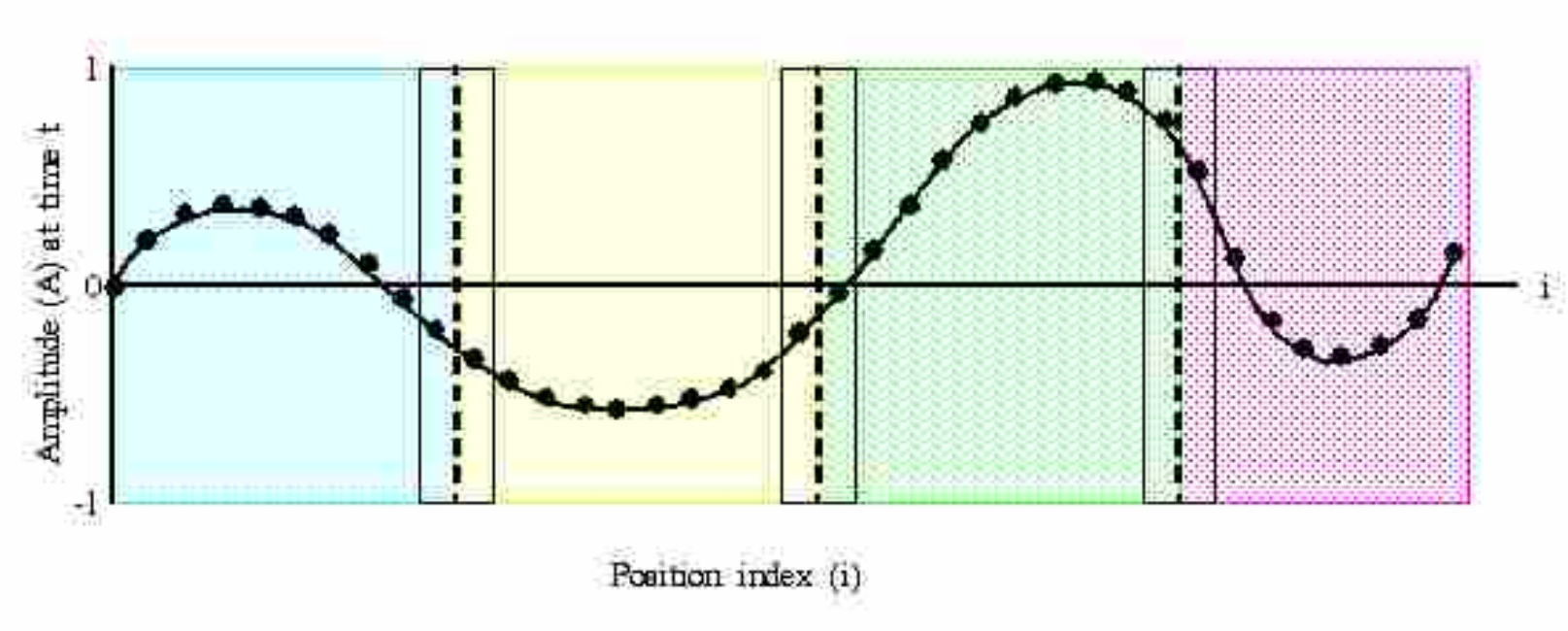
All threads meet here and wait for each other!

Only master thread executes the following section

Get overall number of threads which are running in parallel section

Code example: Wikipedia

Parallelisation Pattern: Domain decomposition

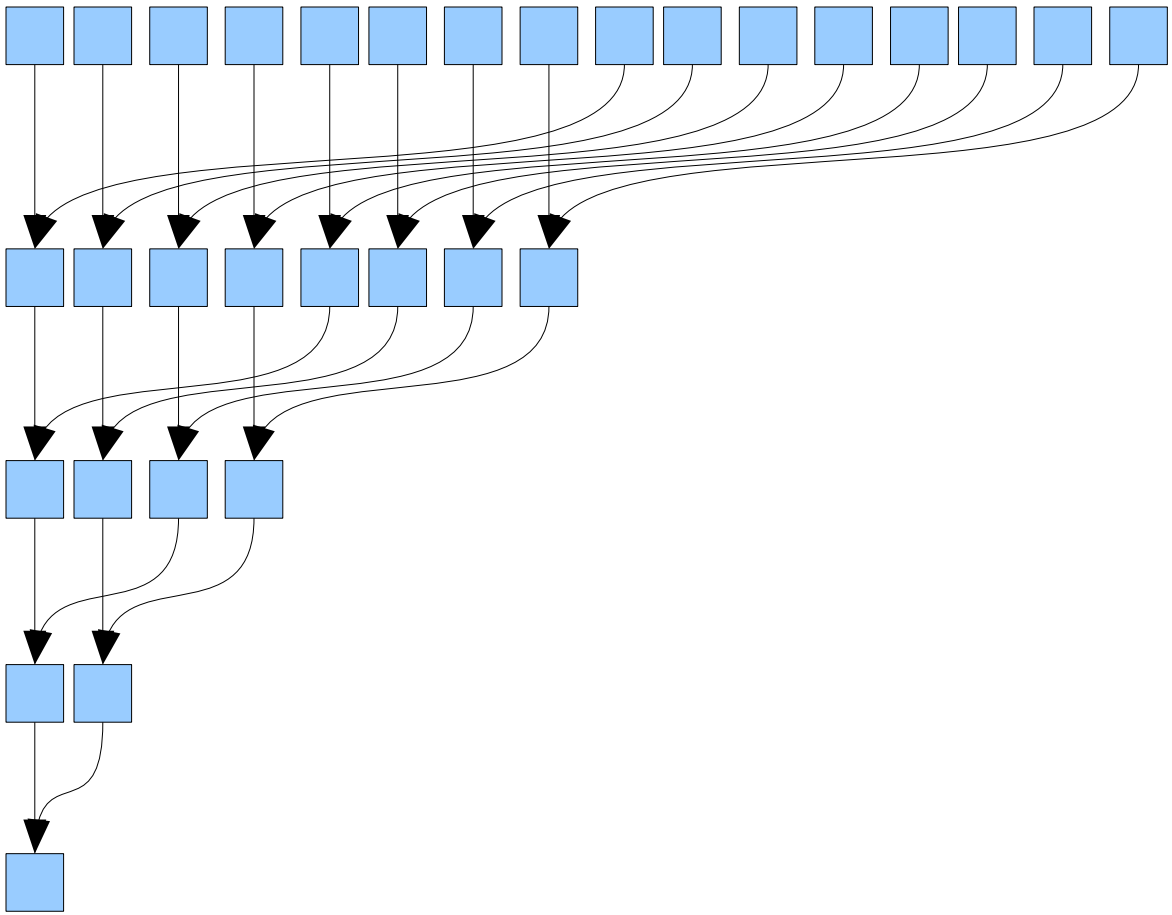


From “Introduction to Parallel Computing” @ https://computing.llnl.gov/tutorials/parallel_comp/

Pattern: Stencil

0	1	0	3	0	2	7	8	9	0	0	6	7	3	2	8
8	7	6	3	8	4	9	**0**	1	6	4	7	3	2	8	9
0	0	1	2	3	6	**7**	**4**	**8**	2	9	0	1	9	8	3
8	7	3	6	4	1	7	**8**	2	9	1	8	2	7	1	1
8	8	8	8	7	4	7	3	6	2	0	0	1	9	2	8
1	2	3	4	5	1	1	7	8	7	9	2	8	1	9	0
1	7	3	6	4	9	1	7	3	6	1	9	2	0	1	6
1	7	3	6	4	9	1	7	1	9	0	0	2	8	1	0

Pattern: Reduction



References

- Introduction to Parallel Computing @ https://computing.llnl.gov/tutorials/parallel_comp/
- Structured Parallel Programming by Michael McCool, James Reinders, Arch Robinson