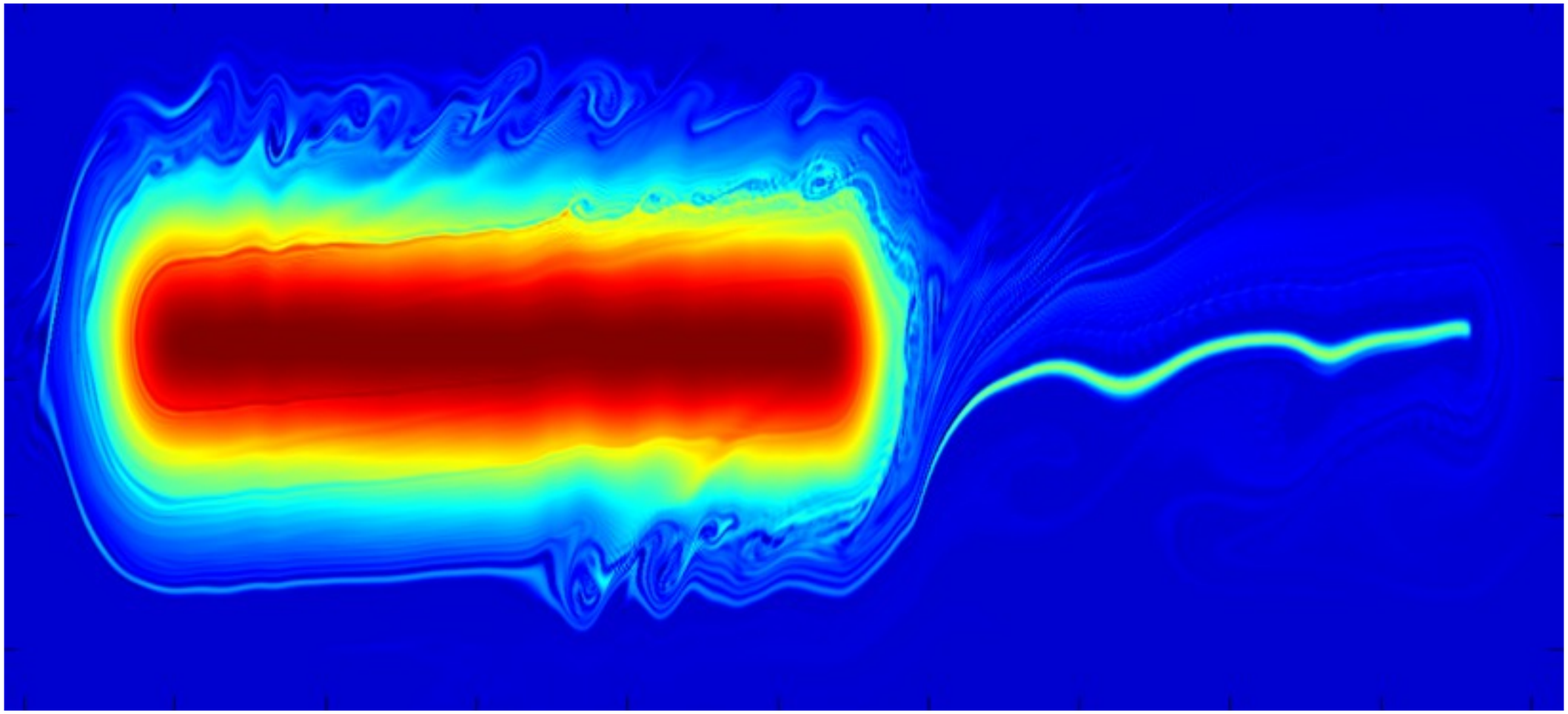
HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

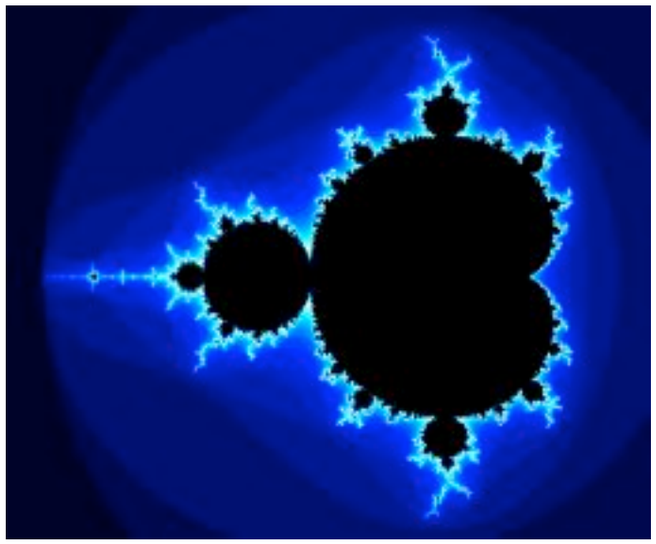


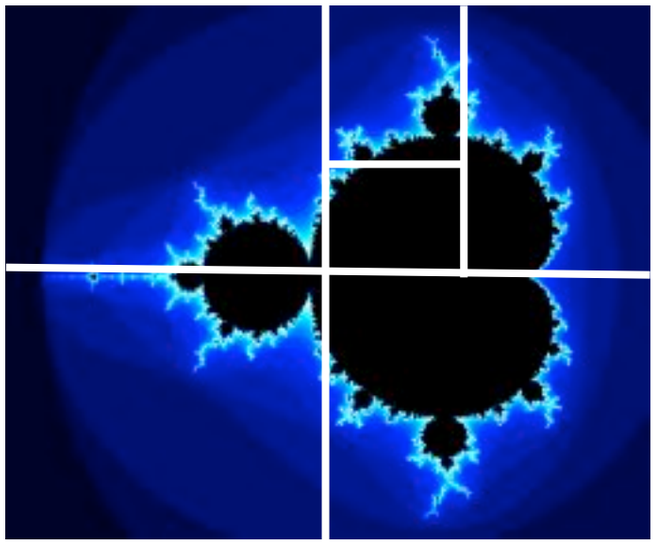
Chapter 6 - Open MP

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Dividing problems into independent sub-problems

- In our projects we are often in the situation that some calculations may be decomposed into sub-problems, each of which could then be solved/executed independently.
- Example: For the Mandelbrot set, calculations for each sub-domain are totally independent of each other.





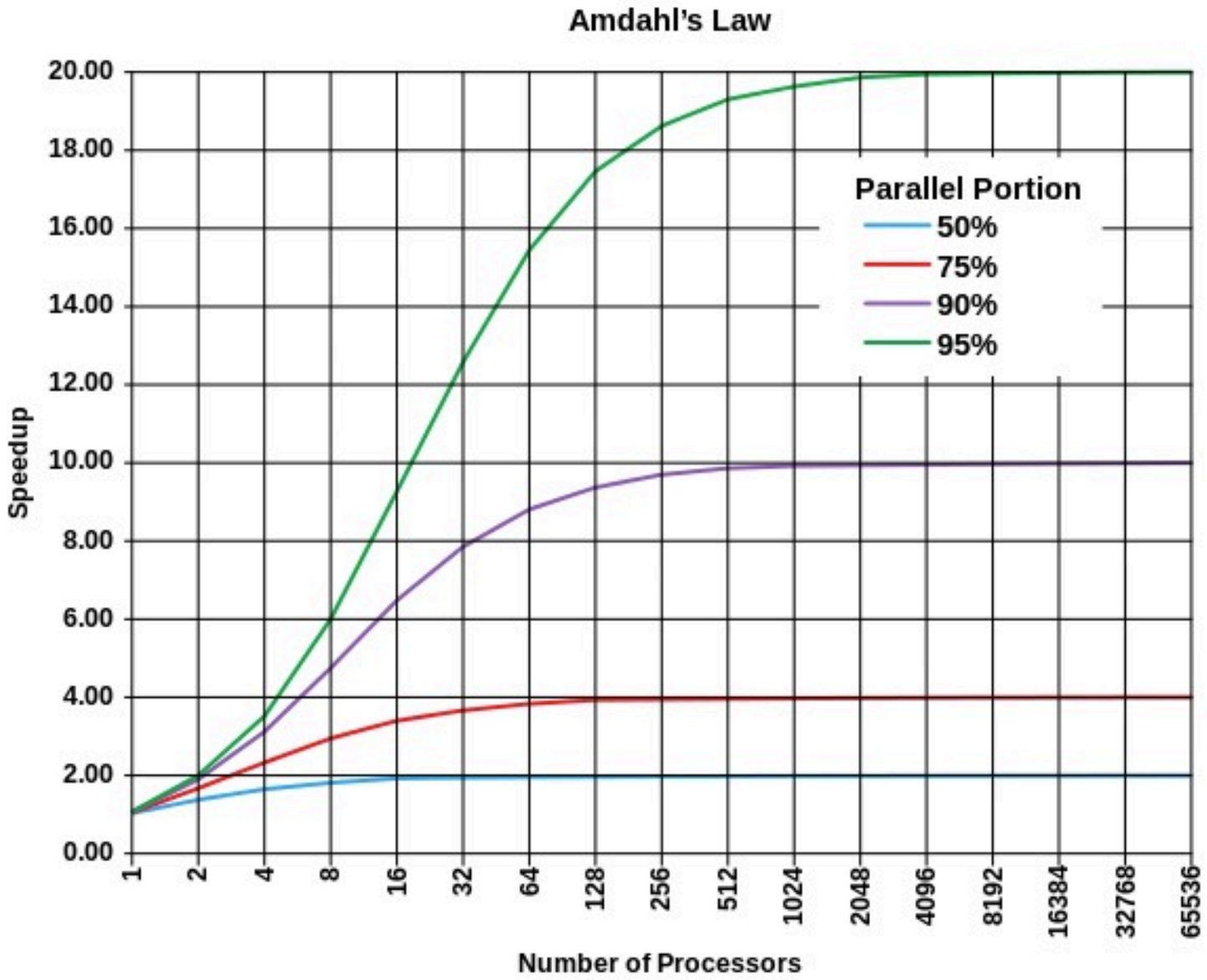
- Matrix-Vector multiplication: $Mx=b$
  We can split this into several tasks, each of which calculates a certain number of entries of b.
- Not for all problems it is directly clear how splitting in independent sub-problems can be achieved, e.g. solving linear systems $Mx=b$.

Amdahl's Law

- Suppose we identify a part of our program which could potentially be executed in parallel. By how much could we decrease the runtime of our program if we knew how to write efficient parallel code?
- Assume we have a completely serial program and that *p* is the fraction of the runtime that is taken up by the part we are going to parallelize. Then *(1-p)* is the fraction taken up by the serial part.
  In total we have a runtime $r = 1 = (1-p) + p$.
- If we could speed up the parallel part by a factor of N when using N workers we would get $r^* = (1-p) + p/N$.
- The gain is then $\frac{r}{r^*} = \frac{1}{(1-p) + p/N} \leq \frac{1}{1-p}$ and thus limited by the amount of serial work.
- Amdahl's law: The total achievable speed-up by parallelization is limited by the fraction of the program which can not be parallelized.

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Amdahl's law



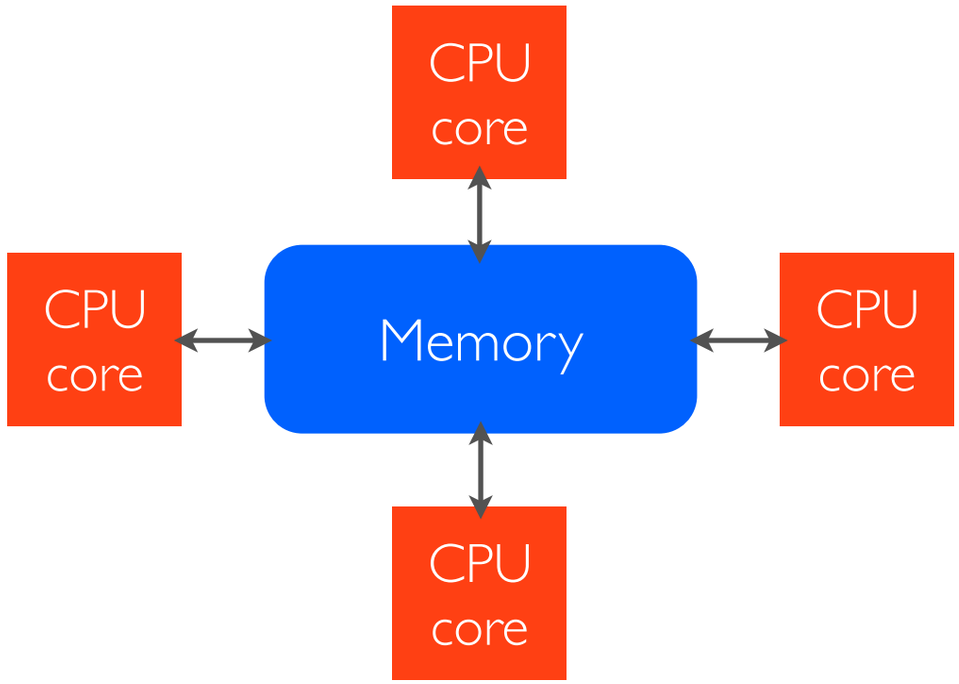
- Only when the parallel portion is the truly dominant part we might expect better performance due to parallelization.
- Amdahl's law is an estimate, true results may vary for the better or the worse.
- Scaling for the parallel part could be super-linear due to better cache usage.
- Scaling could be worse e.g. due to bad implementations or band-width limitation for communication.

(diagram nicked from Wikipedia)

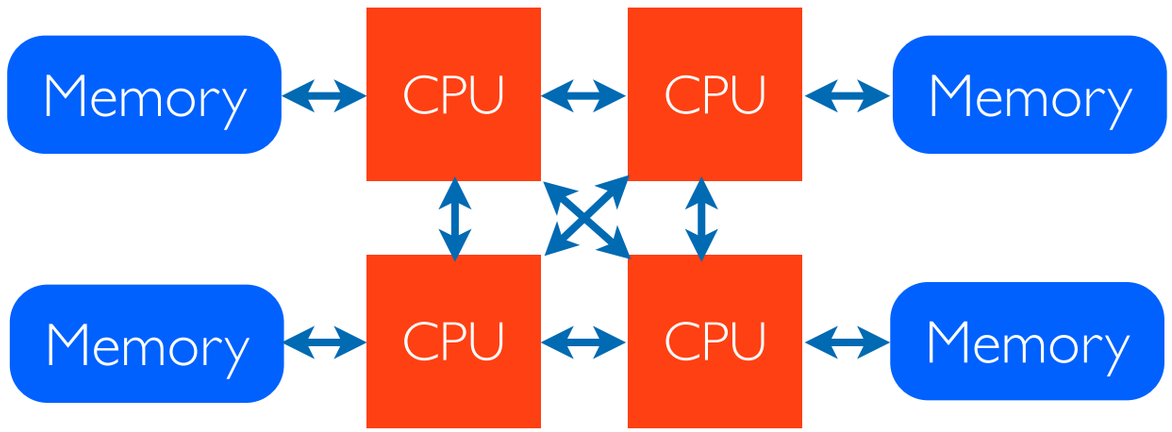
HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Shared memory

- In *shared memory parallelization* all processes which divide the work amongst themselves share a common memory.
- Shared memory allows to easily set up the initial values for each worker and to combine the final results. Additionally any communication between the different workers is typically very fast.



Multi-core environment



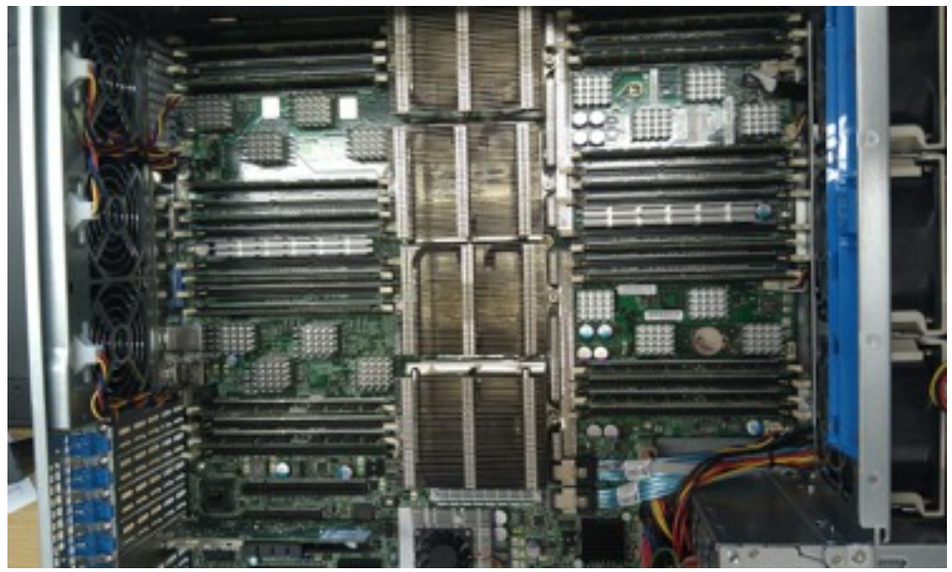
Multi-socket (multi-core) environment

Parallelization

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Shared memory systems

- Multi-core CPUs
- Multi-socket systems, e.g.: 4 x Xeon 8 Core = 32 Cores + 128 GB RAM
- Special multi-core machines, e.g. at HILBERT@ZIM, 512 cores, 16 TB RAM

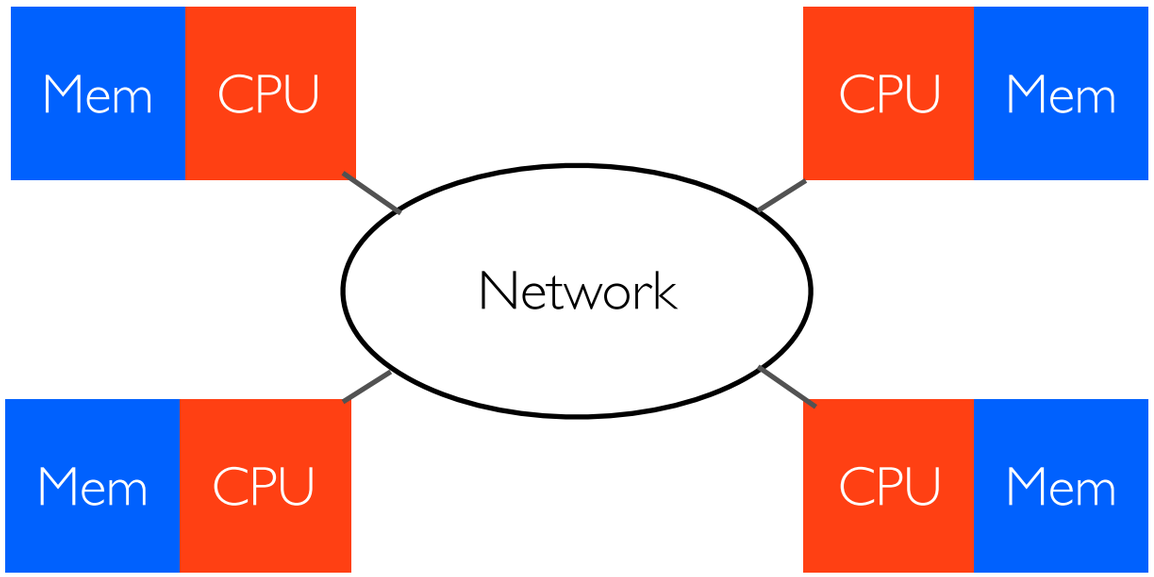


- Shared memory programs use threads to carry out several tasks in parallel
- Either threads can all work on a part of a common, large problem, or they can perform completely different tasks at the same time.
- Programming multi-threaded applications can be done via external libraries (e.g. pthreads or OpenMP).
- OpenMP is a simple way to make use of threads without having to worry too much about the particular operating system and the underlying hardware.

Parallel computing models

Distributed memory

- In *distributed memory* parallel programs each processes has its own memory. In general the memory is physically separate.
- The processes can operate on completely separate machines (memory is really separate then) but also on one machine (but each process has his dedicated part of memory and no other process has direct access to this memory).



- Communication amongst the processes uses a network. This is potentially slow. The latency due to communication has to be well hidden in the code.
- Buy many machines and a fast network to build a really huge computer.

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Distributed memory systems

- Simple distributed memory systems (commonly referred to as cluster) can be build from few regular desktop PCs connected via GBit Ethernet.
- Interconnects via specialized networks like Infiniband or Myrinet allow for higher bandwidth and lower latency then Ethernet. Typically the network becomes the bottelneck which prevents applications to scale to more than a few tens of computers.
- It is also possible to build very large systems from almost standard hardware, e.g. JUROPA @ Jülich, ~25k Intel Xeon cores, 24 GB RAM per 8 cores, QDR Infiniband
- Specialized hardware allows to built very large systems, e.g. JuQUEEN, 460k cores, 448 TB RAM, ~6 PetaFlops

- Programming the communication between tasks on different nodes is almost exclusively done via MPI (message passing interface)

OpenMP

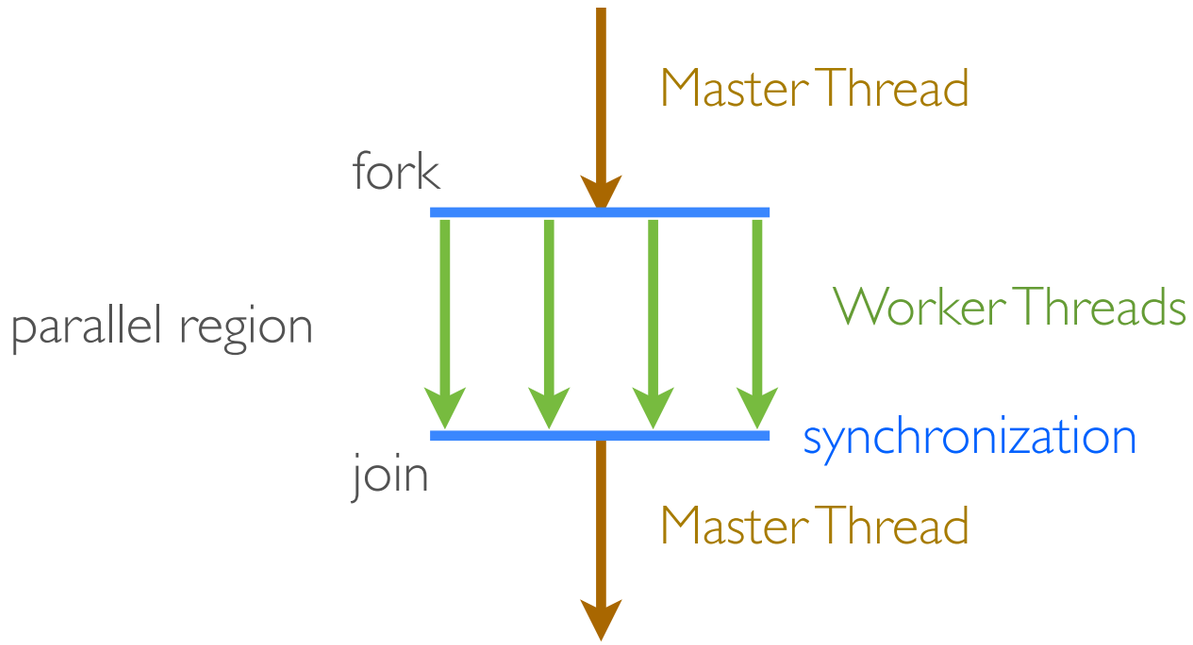
HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

- OpenMP is an extension available for C/C++ and Fortran which allows to use threads for computations, without the need to explicitly program threads.
- OpenMP consists of a number of statements which tell the compiler where it should try to split work amongst several threads.
- Once we have a serial code and want to speed up certain parts by using OpenMP the additional effort in programming is very small. Most of the work is done by the compiler.
- Writing multi-threaded programs is usually different for each operating system and might depend on the hardware. OpenMP is available for all major operating systems and compilers, i.e. in the end we will have very portable code.
- To enable OpenMP for the GNU C++ compiler we need to add options
  `g++ -lgomp -fopenmp`
- The compiler will use threads where we tell him to, but also will try to find places where it thinks that threads might be beneficial.

OpenMP

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

- OpenMP divides the program into serial and parallel parts



- OpenMP is used by inserting *pragmas* in the code.
- A pragma is a command, preceeded by `#pramga`, e.g.

```
#pragma omp parallel
#pragma omp atomic
```

- Pragmas are instructions for the compiler to generate multi-threaded code

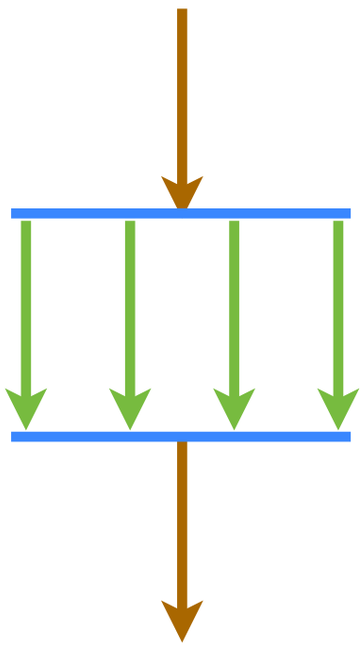
OpenMP

- OpenMP defines *parallel regions*. The parts of the program which are enclosed in a parallel region are executed by all threads in parallel.

<u>Example</u>

serial region

fill two vectors with values



parallel region

add the two vectors (each thread will work on a part of the vectors)

serial region

do whatever comes next in the calculation

- Most commonly parallel regions enclose a for loop. If the operations are independent of each other for all iterations, executing several iterations in parallel is mostly trivial (vector addition is a good example for this).

parallel sections

- The pragma `#pargma omp parallel` creates a parallel section. All statements inside the section are executed by all threads.
- At the beginning of a parallel section several threads are started. Each thread is assigned a thread ID.
- The thread with ID 0 is the master thread.
- Only the master thread will continue execution when the parallel section is closed.

```
#include <omp.h>
#include <iostream>

int main(void){
    int i;

    int NT= omp_get_num_threads();
    std::cout << "We have currently " << NT << " threads" << std::endl;

    #pragma omp parallel
    {
        const int id = omp_get_thread_num();
        NT = omp_get_num_threads();
        std::cout << "Here is thread " << id <<  " of" << NT << std::endl;
    }

}
```

Output:

```
We have currently 1 threads
Here is thread Here is thread 01 of of4Here is thread 4Here is thread
2
3 of of44
```

→ Problems due to race conditions

parallel sections

- To avoid the scrambled output of the previous program we have to make sure that the operators << of the different output lines do not compete with each other.
- We at first generate the complete line we want to print in a stringstream object and then use only one << per line to print it to the screen.

```
#include <omp.h>
#include <iostream>
#include <sstream>
#include <string>

int main(void){
    int i;

    int NT= omp_get_num_threads();
    std::cout << "We have currently " << NT << " threads" << std::endl;
    #pragma omp parallel
    {
        const int id = omp_get_thread_num();
        NT = omp_get_num_threads();
        std::stringstream s;
        s << "Here is thread " << id <<  " of" << NT << "\n" << std::flush;
        std::cout << s.str();
    }
}
```

Output:

```
We have currently 1 threads
Here is thread 0 of 4
Here is thread 1 of 4
Here is thread 2 of 4
Here is thread 3 of 4
```

shared and private variables

- Variables that are declared before the parallel region starts are *shared*, i.e. every thread will have read and write access to these variables. If one thread modifies the value of the variable, all threads will see this change.

- If we want each thread to have a local copy of a variable existing before the parallel region starts, we can mark the variable to be *private*. In this case each thread will have a local copy of the variable that may be changed without influencing the value of this variable in other threads.

- Variables that are created within a parallel region are always local to the thread, i.e. each thread will create memory for such a variable.

shared and private variables

- Parallel addition of two vectors

```
int main ()
{
     const int N = 10000000;
     double* a=new double[N];
     double* b=new double[N];
     int i;

     #pragma omp parallel shared(N,a,b) private(i)
     {
          #pragma omp for
          for(i=0; i<N; i++)
               a[i] += b[i];
     }

     delete[] a;
     delete[] b;
}
```

Directives and clauses

- OpenMP pragmas all follow the scheme
  `#pragma omp directive [clause ...]`
- OpenMP directives are
  - parallel / parallel for
  - for
  - barrier
  - single
  - section / sections
  - ...
- Depending on the directive one might append one or more of the following clauses:
  - shared / private
  - firstprivate / lastprivate
  - nowait
  - default
  - reduction
  - ...

Directives and clauses

- The *nowait* clause instructs the thread that when it is done with a loop inside a parallel region it does not have to wait for all other threads to finish the loop before it starts working on what comes next.
- A *barrier* is a point which first all threads have to reach before any of them may continue with the next instructions.

```
#pragma omp parallel shared(N,a,b,x,y) private(i)
{
    #pragma omp for nowait
    for(i=0; i<N; i++)
        a[i] += b[i];

    #pragma omp for nowait
    for(i=0; i<N; i++)
        x[i] += y[i];

    #pragma omp barrier

    #pragma omp for
    for(i=0; i<N; i++)
        x[i] += a[i];
}
```

- A regular for-loop (without nowait clause) contains an implicit barrier.
  Hence, we can leave the barriere out of this program when we also take away nowait from the second for-loop

Directives and clauses

- The *if* clause can be used to decide whether a section should be carried out in parallel or sequentially. For small sized problems the overhead for thread creation and initialization is larger than the benefit from multi-threading.

```
#pragma omp parallel if (N>100) shared(N,a,b,x,y) private(i)
{
    #pragma omp for nowait
    for(i=0; i<N; i++)
        a[i] += b[i];

}
```

Clauses for variables

- *private(list):*
  Variables from the list are independent of global variables existing outside the OpenMP region. Each thread has its own variable.
  Attention: All variables from the list are not initialized when entering and leaving the region!
- *shared(list):*
  All threads share the same variable and thus the same memory to store the variable.
  Changes are global amongst the threads. One thread changes the value of the variable, all others will see it.
- *firstprivate(list):*
  When entering the OpenMP region, all variables in the list are initialized with the values of the equally named variables outside the region.
- *lastprivate(list):*
  The values of these variables after leaving the OpenMP region are set to the private values of the thread that wrote these values last.

Clauses for variables

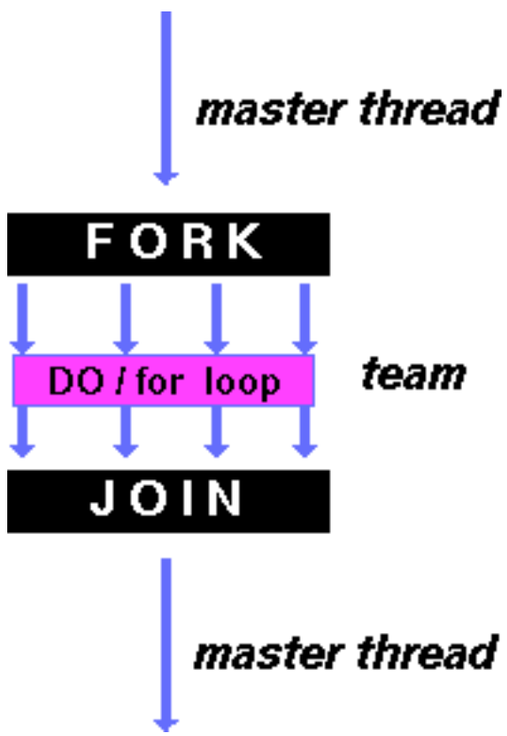
```
int A = 10;
#pragma omp parallel
{
   #pragma omp for private(i) firstprivate(A) \
                                 lastprivate(B)
    for (i=0; i<n; i++){
       // (...)
        B = A + i; // A would be undefined, if not declared as firstprivate
       // (...)
    }
    // (...)
    C = B;
    // B would be undefined unless declared as lastprivate
    // (...)
}
```

Clauses for variables and reductions

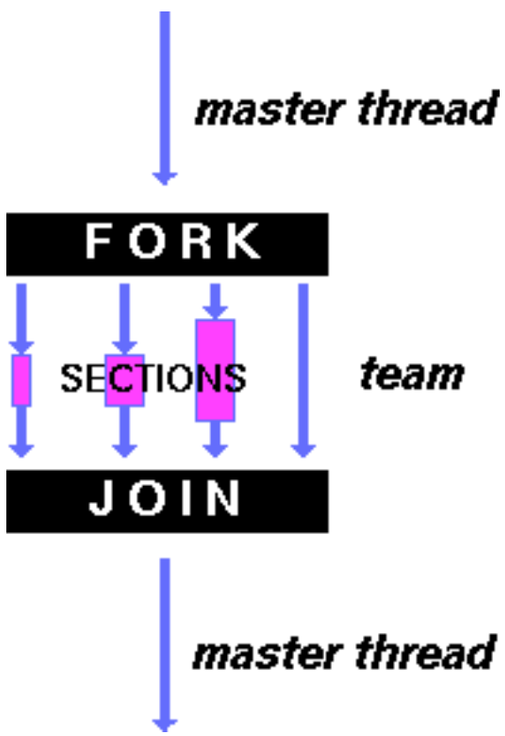
- *default(shared | none | private):*
  This clause defines the default for variables inside an OpenMP region if nothing is specified. The default private is not possible in C++, but in Fortran.
- *reduction(operator : list):*
  For each variable in the list a private copy will be created. At the end of the OpenMP region all values of these private variables from the different tasks will reduced to a single value via the specified operator. The result will be stored in the global shared variable of the same name.
  Operators may be +, *, -, /, ^, &, |, &&, || of the C++ base types.

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

int main ()
{
     const int N = 10;
     double* a=new double[N];
     double* b=new double[N];
     double result=0;

     for (int i=0; i < N; i++){
       a[i] =  1.0;   b[i] =  2.0;
     }

     #pragma omp parallel default(shared)
     {
        #pragma omp for reduction(+:result)
        for (int i=0; i < N; i++)
           result = result + (a[i] * b[i]);
     }

     cout << "result= " << result << endl;
     delete[] a;
     delete[] b;
}
```

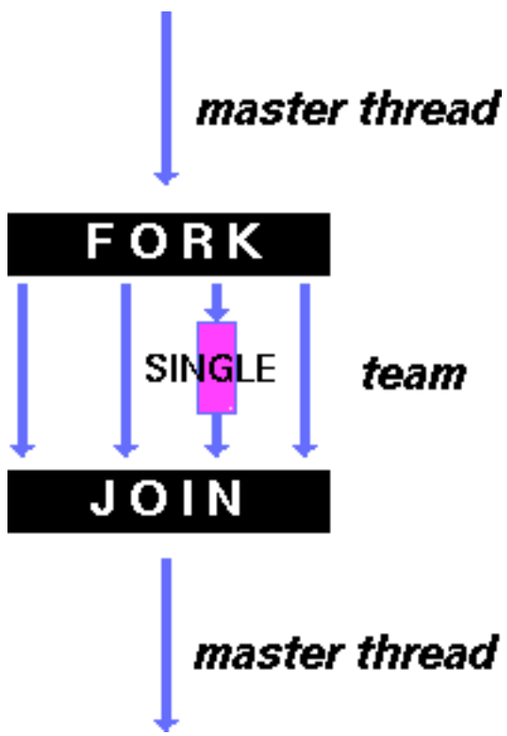
Work sharing contructs



- `#pragma omp for` splits for loops over multiple threads. All threads work together on one problem.
(Data parallel work sharing)



- `#pragma omp sections` assigns different tasks to different threads. Each thread works on a different problem.
(Task parallel work sharing)



- `#pragma omp single` defines a block that is only executed by a single task from a team of tasks.

All of the above work sharing sections have to inside a parallel region.

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Parallel sections, task parallelism

- #pragma omp section defines a block of code that should be executed in parallel to other sections (i.e. blocks of code). These sections have to be enclosed in a #pragma omp sections block.
- When there are more sections than threads, OpenMP will decide how to schedule the sections.
- When there are more threads then sections some threads will go without work.

```
#pragma omp parallel shared(a,b,c,d) private(i)
  {

#pragma omp sections nowait
  {

  #pragma omp section
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  #pragma omp section
  for (i=0; i < N; i++)
    d[i] = a[i] * b[i];

  }  /* end of sections */

}  /* end of parallel section */
```

parallel for, data parallelism

- For-loops are the most common case in which we might apply OpenMP to parallelize work.
- `#pragma omp for [clause ...] newline`

  ```
  schedule (type [,chunk])
  ordered
  private (list)
  firstprivate (list)
  lastprivate (list)
  shared (list)
  reduction (operator: list)
  collapse (n)
  nowait
  for_loop
  ```

- The schedule clause defines how the single iterations are spread amongst the threads. The default behavior for scheduling is different for different implementations of OpenMP.
- If each iteration of the for-loop has the same computational complexity *static* scheduling is the best. In this way the total number of iterations is spread evenly amongst the threads. (Additionally the chunk size may be given)
- In case of *dynamic* scheduling the for-loop will be divided between threads, all threads get the same amount of work initially but then later they might get different amounts of work, depending on their performance. This can be interesting if the work per iteration is not constant.

OpenMP

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

- If there is only a single work sharing construct within a parallel region there are shortcut directives like `#pramga omp parallel for` or `#pragma omp parallel sections` available.

```
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  } /* end of parallel section */
```

```
#pragma omp parallel for shared(a,b,c,chunk) private(i) \
                         schedule(dynamic,chunk)

  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
```

```
#pragma omp parallel shared(a,b,c,d) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    } /* end of sections */

  } /* end of parallel section */
```

```
#pragma omp parallel sections shared(a,b,c,d) private(i) \
nowait
  {
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

  } /* end of parallel sections */
```

Race conditions

- Race conditions are expressions that depend on the precise execution order of unsynchronized threads.
- Since the execution order can be different each time the program might work sometimes and not other times. These errors are hard to detect.
- Example:

```
int i=0;

#pragma omp parallel shared(i)
{
    i++;
}
```

i should be equal to the number of threads at the end of the parallel region...

- Due to unlucky interruption of one thread the result can be wrong.
- This particular example can be fixed by using `#paragma omp atomic`

```
int i=0;
#pragma omp parallel shared(i)
{
    #pragma omp atomic
    i++;
 }
```

clock	thread I	thread II
1	load i (i=0)	
2	incr i (i=1)	
3	swap out	load i (i=0)
4		incr i (i=1)
5		store i (i=1)
6	store i (i=1)	swap out

Critical sections

- #pragma omp atomic can only be used for native C++ datatypes and for simple commands. Accessing elements of arrays is also not possible in this way.
- For more complex code elements which should only be executed by one thread at a time there is `#pragma omp critial`. This makes sure that the statements inside the block are savely executed by only one thread at a time.

```
// go parallel
#pragma omp parallel
{
    // do something sensible in parallel
    ...
    #pragma omp critical
    {
        // protected region starts here
        // only one thread at a time works here
    }
    // more parallel work
    ...
}
```

- Atomic operations and critical sections of course degrade the parallel performance. The code essentially becomes serial at these positions, so this should be avoided wherever possible.