Scientific Computing WS 2018/2019

Lecture 23

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

TOP 500 2018 rank 1-6

Based on linpack benchmark: solution of dense linear system. Typical desktop computer: $R_{max} \approx 100 \ldots 1000 GFlop/s$

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,397,824	143,500.0	200,794.9	9,783
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	387,872	21,230.0	27,154.3	2,384
6	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	979,072	20,158.7	41,461.2	7,578

[Source:www.top500.org]

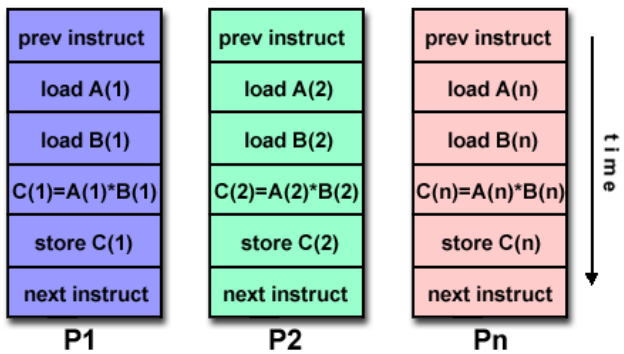
TOP 500 2018 rank 7-13

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
7	National Institute of Advanced Industrial Science and Technology (AIST) Japan	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR Fujitsu	391,680	19,880.0	32,576.6	1,649
8	Leibniz Rechenzentrum Germany	**SuperMUC-NG** - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path Lenovo	305,856	19,476.6	26,873.9	
9	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
10	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
11	DOE/NNSA/LLNL United States	**Lassen** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100 IBM / NVIDIA / Mellanox	248,976	15,430.0	19,904.4	
12	DOE/SC/LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	27,880.7	3,939
13	Korea Institute of Science and Technology Information Korea, South	**Nurion** - Cray CS500, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path	570,020	13,929.3	25,705.9	

[Source:www.top500.org]

Parallel paradigms

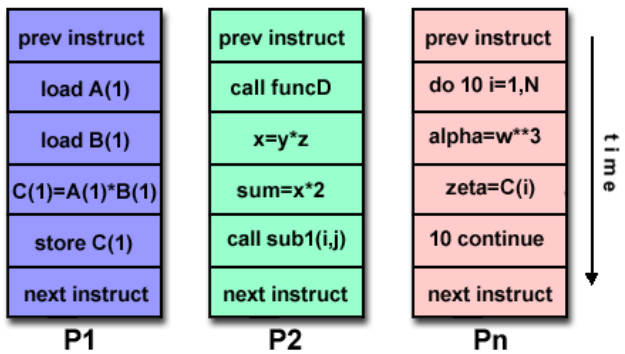
SIMD
Single Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

- "classical" vector systems: Cray, Convex ...
- Graphics processing units (GPU)

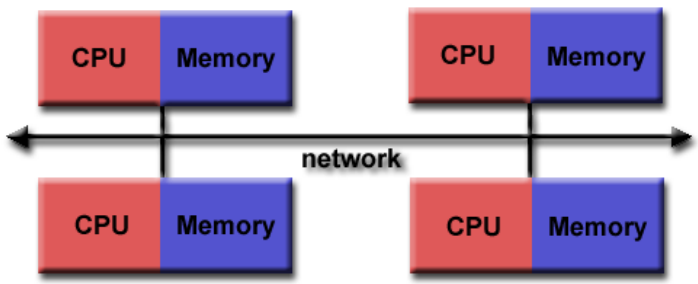
MIMD
Multiple Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

- Shared memory systems
  - IBM Power, Intel Xeon, AMD Opteron ...
  - Smartphones ...
  - Xeon Phi R.I.P.
- Distributed memory systems
  - interconnected CPUs

MIMD Hardware: Distributed memory



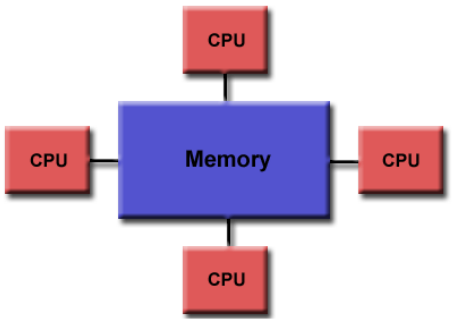
[Source: computing.llnl.gov/tutorials]

- "Linux Cluster"
- "Commodity Hardware"
- Memory scales with number of CPUs interconneted
- High latency for communication
- Mostly programmed using MPI (Message passing interface)
- Explicit programming of communications:
  gather data, pack, send, receive, unpack, scatter

```
MPI_Send(buf,count,type,dest,tag,comm)
MPI_Recv(buf,count,type,src,tag,comm,stat)
```

MIMD Hardware: Shared Memory

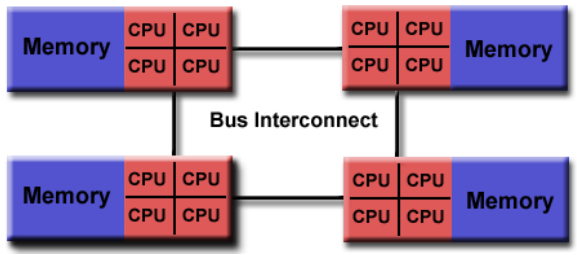
Symmetric Multiprocessing (SMP)/Uniform memory acces (UMA)



[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



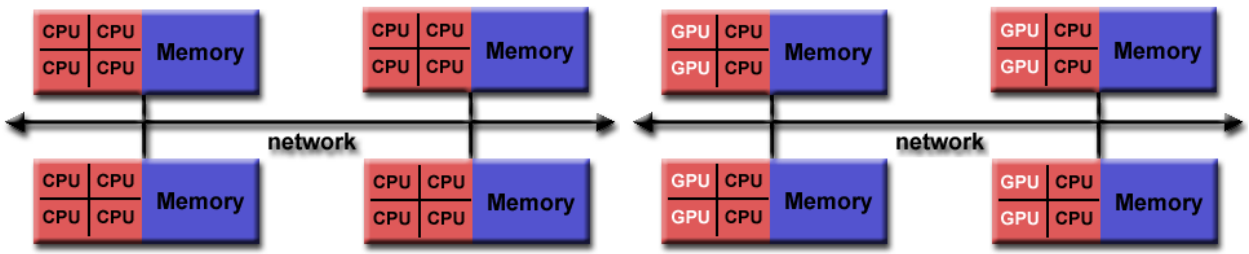
[Source: computing.llnl.gov/tutorials]

- Possibly varying memory access latencies
- Combination of SMP systems
- ccNUMA: Cache coherent NUMA

- Shared memory: one (virtual) address space for all processors involved
- Communication hidden behind memory acces
- Not easy to scale large numbers of CPUS
- MPI works on these systems as well

Hybrid distributed/shared memory

- Combination of shared and distributed memory approach
- Top 500 computers



[Source: computing.llnl.gov/tutorials]

- Shared memory nodes can be mixed CPU-GPU
- Need to master both kinds of programming paradigms

Shared memory programming: pthreads

- Thread: lightweight process which can run parallel to others
- pthreads (POSIX threads): widely distributed
- cumbersome tuning + syncronization
- basic structure for higher level interfaces

```
#include <pthread.h>
void *PrintHello(void *threadid)
{  long tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;   long t;
  for(t=0; t<NUM_THREADS; t++)
  {
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
    if (rc) {printf("ERROR; return code from pthread_create() is %d\n", rc); exit(-1);}
  }
  pthread_exit(NULL);
}
```

Source: computing.llnl.gov/tutorials

- compile and link with

```
gcc -pthread -o pthreads pthreads.c
```

Shared memory programming: C++11 threads

- Threads introduced into C++ standard with C++11
- Quite late... many codes already use other approaches
- But interesting for new applications

```
#include <iostream>
#include <thread>

void call_from_thread(int tid) {
  std::cout << "Launched by thread " << tid << std::endl;
}

int main() {
  std::thread t[num_threads];
  for (int i = 0; i < num_threads; ++i) {
    t[i] = std::thread(call_from_thread, i);
  }
  std::cout << "Launched from main\n";
  //Join the threads with the main thread
  for (int i = 0; i < num_threads; ++i) {
    t[i].join();
  }
  return 0;
}
```

Source: https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/

- compile and link with

```
g++ -std=c++11 -pthread cpp11threads.cxx -o cpp11threads
```

Thread programming: mutexes and locking

- If threads work with common data (write to the same memory address, use the same output channel) access must be synchronized
- Mutexes allow to define regions in a program which are accessed by all threads in a sequential manner.

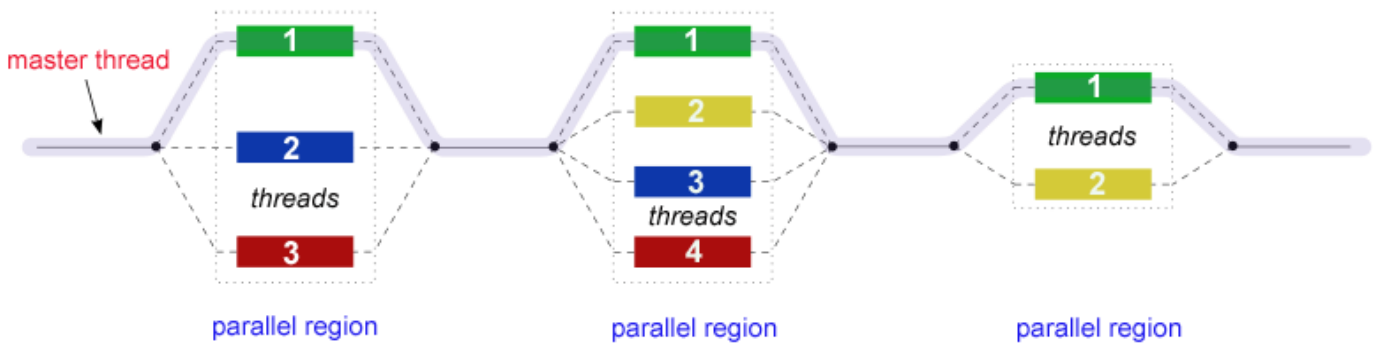
```
#include <mutex>
std::mutex mtx;
void call_from_thread(int tid) {
  mtx.lock()
  std::cout << "Launched by thread " << tid << std::endl;
  mtx.unlock()
}
int main()
{
  std::thread t[num_threads];
  for (int i = 0; i < num_threads; ++i) {
    t[i] = std::thread(call_from_thread, i);
  }
  std::cout << "Launched from main\n";
  for (int i = 0; i < num_threads; ++i) t[i].join();
  return 0;
}
```

- *Barrier*: all threads use the same mutex for the same region
- *Deadlock*: two threads block each other by locking two different locks and waiting for each other to finish

Shared memory programming: OpenMP

- Mostly based on pthreads
- Available in C++,C,Fortran for all common compilers
- Compiler directives (pragmas) describe *parallel regions*

```
... sequential code ...
#pragma omp parallel
{
  ... parallel code ...
}
(implicit barrier)
... sequential code ...
```



[Source: computing.llnl.gov/tutorials]

Shared memory programming: OpenMP II

```
#include <iostream>
#include <cstdlib>

void call_from_thread(int tid) {
  std::cout << "Launched by thread " << tid << std::endl;
}

int main (int argc, char *argv[])
{
  int num_threads=1;
  if (argc>1) num_threads=atoi(argv[1]);

  #pragma omp parallel for
  for (int i = 0; i < num_threads; ++i)
  {
    call_from_thread(i);
  }
  return 0;
}
```

- compile and link with

```
g++ -fopenmp -o cppomp cppomp.cxx
```

Example: $u = au + v$ und $s = u \cdot v$

```
double u[n],v[n];
#pragma omp parallel for
for(int i=0; i<n ; i++)
    u[i]+=a*v[i];

//implicit barrier
double s=0.0;
#pragma omp parallel for
for(int i=0; i<n ; i++)
    s+=u[i]*v[i];
```

- Code can be parallelized by introducing compiler directives
- Compiler directives are ignored if not in parallel mode
- Write conflict with $+$ s: several threads may access the same variable

Preventing conflicts in OpenMP

- Critical sections are performed only by one thread at a time

```
double s=0.0;
#pragma omp parallel for
for(int i=0; i<n ; i++)
#pragma omp critical
{
  s+=u[i]*v[i];
}
```

- Expensive, parallel program flow is interrupted

Do it yourself reduction

- Remedy: accumulate partial results per thread, combine them after main loop
- "Reduction"

```
#include <omp.h>
int maxthreads=omp_get_max_threads();
double s0[maxthreads];
double u[n],v[n];
for (int ithread=0;ithread<maxthreads; ithread++)
   s0[ithread]=0.0;

#pragma omp parallel for
for(int i=0; i<n ; i++)
{
  int ithread=omp_get_thread_num();
  s0[ithread]+=u[i]*v[i];
}

double s=0.0;
for (int ithread=0;ithread<maxthreads; ithread++)
  s+=s0[ithread];
```

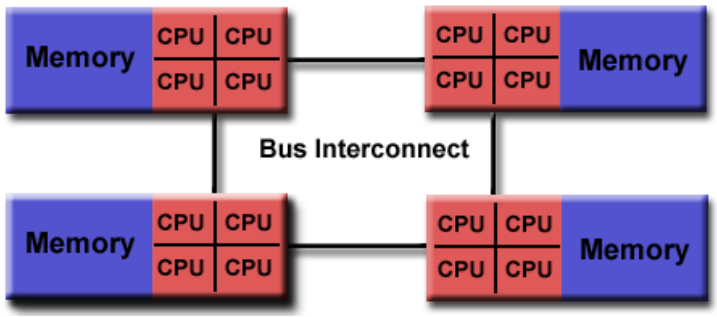
OpenMP Reduction Variables

```
double s=0.0;
#pragma omp parallel for reduction(+:s)
for(int i=0; i<n ; i++)
    s+=u[i]*v[i];
```

- In standard situations, reduction variables can be used to avoid write conflicts, no need to organize this by programmer

OpenMP: further aspects

```
double u[n],v[n];
#pragma omp parallel for
for(int i=0; i<n ; i++)
u[i]+=a*u[i];
```



[Quelle: computing.llnl.gov/tutorials]

- Distribution of indices with thread is implicit and can be influenced by scheduling directives
- Number of threads can be set via `OMP_NUM_THREADS` environment variable or call to `omp_set_num_threads()`
- First Touch Principle (NUMA): first thread which "touches" data triggers the allocation of memory with the processeor where the thread is running on

Parallelization of PDE solution

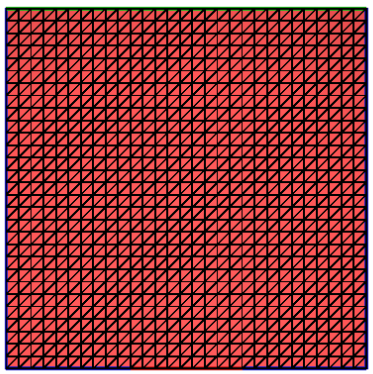
$$\Delta u = f \text{ in} \Omega, \qquad u|_{\partial\Omega} = 0$$

$$\Rightarrow u = \int_\Omega f(y) G(x,y) dy.$$

- Solution in $x \in \Omega$ is influenced by values of f in all points in Ω
- $\Rightarrow$ global coupling: any solution algorithm needs global communication

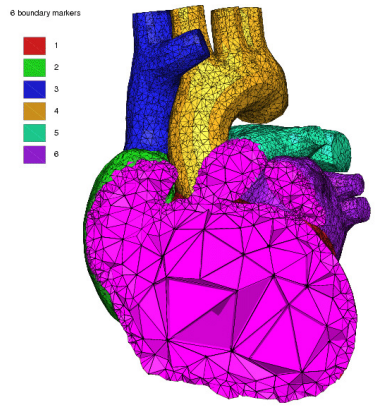
Structured and unstructured grids

Structured grid



- Easy next neighbor access via index calculation
- Efficient implementation on SIMD/GPU
- Strong limitations on geometry

Unstructured grid



[Quelle: tetgen.org]

- General geometries
- Irregular, index vector based access to next neighbors
- Hardly feasible fo SIMD/GPU

Stiffness matrix assembly for Laplace operator for P1 FEM

$$a_{ij} = a(\phi_i, \phi_j) = \int_\Omega \nabla\phi_i \nabla\phi_j \; dx$$
$$= \int_\Omega \sum_{K \in \mathcal{T}_h} \nabla\phi_i|_K \nabla\phi_j|_K \; dx$$

Assembly loop:
Set $a_{ij} = 0$.
For each $K \in \mathcal{T}_h$:
For each $m, n = 0 \ldots d$:

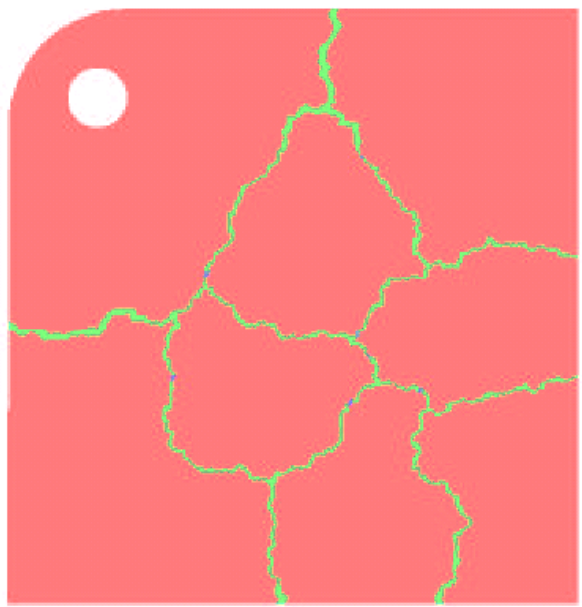
$$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \; dx$$
$$a_{j_{dof}(K,m), j_{dof}(K,n)} = a_{j_{dof}(K,m), j_{dof}(K,n)} + s_{mn}$$

Mesh partitioning

Partition set of cells in $\mathcal{T}_h$, and color the graph of the partitions.

Result: $\mathcal{C}$: set of colors, $\mathcal{P}_c$: set of partitions of given color. Then: $\mathcal{T}_h = \bigcup_{c \in \mathcal{C}} \bigcup_{p \in \mathcal{P}_c} p$

- Sample algorithm:
  - Subdivision of grid cells into equally sized subsets by METIS (Karypis/Kumar) $\rightarrow$ Partitions of color 1
  - Create separators along boundaries $\rightarrow$ Partitions of color 2
  - "triple points" $\rightarrow$ Partitions of color 3



Parallel stiffness matrix assembly for Laplace operator for P1 FEM

Set $a_{ij} = 0$.

For each color $c \in \mathcal{C}$
#pragma omp parallel for
For each $p \in \mathcal{P}_c$:
For each $K \in p$:
For each $m, n = 0 \ldots d$:
$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \, dx$
$a_{j_{dof}(K,m),j_{dof}(K,n)} += s_{mn}$

- Prevent write conflicts by loop organization
- No need for critical sections
- Similar structure for Voronoi finite volumes, nonlinear operator evaluation, Jacobi matrix assembly

Linear system solution

- Sparse matrices
- Direct solvers are hard to parallelize though many efforts are undertaken, e.g. Pardiso
- Iterative methods easier to parallelize
  - partitioning of vectors + coloring inherited from cell partitioning
  - keep loop structure (first touch principle)
  - parallelize
    - vector algebra
    - scalar products
    - matrix vector products
    - preconditioners

MPI - Message passing interface

- library, can be used from C,C++, Fortran, python
- de facto standard for programming on distributed memory systems (since $\approx$ 1995)
- highly portable
- support by hardware vendors: optimized communication speed
- based on sending/receiving messages over network
  - instead, shared memory can be used as well
- very elementary programming model, need to hand-craft communications

How to install

- OpenMP/C++11 threads come along with compiler
- MPI needs to be installed in addition
- Can run on multiple systems
- openmpi available for Linux/Mac (homebrew)/ Windows (cygwin)
  - `https://www.open-mpi.org/faq/?category=mpi-apps`
  - Compiler wrapper `mpic++`
    - wrapper around (configurable) system compiler
    - proper flags + libraries to be linked
  - Process launcher `mpirun`
- launcher starts a number of processes which execute statements independently, ocassionally waiting for each other

Threads vs processes

- MPI is based on *processes*, C++11 threads and OpenMP are based on *threads*.
- Processes are essentially like commands launched from the command line and require large bookeeping, each process has its own address space
- Threads are created within a process and share its address space, require significantly less bookeeping and resources
- Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, with processes there can be no write conflicts
- When working with multiple processes, one becomes responsible for inter-process communication

MPI Programming Style

- Generally, MPI allows to work with completely different programs
- Typically, one writes *one program* which is started in multiple incarnations on different hosts in a network or as different processes on one host
- MPI library calls are used to determine the identiy of a running program and the region of the data to work on
- Communication + barriers have to be programmed explicitely.

MPI Hello world

```
//  Initialize MPI.
MPI_Init ( &argc, &argv );

//  Get the number of processes.
MPI_Comm_size ( MPI_COMM_WORLD, &nproc );

//  Determine the rank (number, identity) of this process.
MPI_Comm_rank ( MPI_COMM_WORLD, &iproc );

if ( iproc == 0 )
{
  cout << "Number of available processes: " << nproc << "\n";
}
cout << "Hello from proc  " << iproc << endl;
MPI_Finalize ( );
```

- Compile with `mpic++ mpi-hello.cpp -o mpi-hello`
- All MPI programs begin with `MPI_Init()` and end with `MPI_Finalize()`
- the *communicator* `MPI_COMM_WORLD` designates all processes in the current process group, there may be other process groups etc.
- The whole program is started N times as system process, not as thread: `mpirun -np N mpi-hello`

MPI hostfile

```
host1  slots=n1
host2  slots=n2

...
```

- Distribute code execution over several hosts
- MPI gets informed how many independent processes can be run on which node and distributes the required processes accordingly
- MPI would run more processes than slots available. Avoid this situation !
- Need ssh public key access and common file system access for proper execution
- Telling mpi to use host file:
  `mpirun --hostfile hostfile -np N mpi-hello`

MPI Send

```
MPI_Send (start, count, datatype, dest, tag, comm)
```

- Send data to other process(es)
- The message buffer is described by (start, count, datatype):
  - start: Start address
  - count: number of items
  - datatype: data type of one item
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.
- The tag codes some type of message

MPI Receive

`MPI_Recv(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.
- source is rank in communicator specified by comm, or `MPI_ANY_SOURCE`
- status contains further information
- Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.

MPI Broadcast

```
MPI_Bcast(start, count, datatype, root, comm )
```

- Broadcasts a message from the process with rank "root" to all other processes of the communicator
- Root sends, all others receive.

Differences with OpenMP

- Programmer has to care about all aspects of communication and data distribution, even in simple situations
- In simple situations (regularly structured data) OpenMP provides reasonable defaults. For MPI these are not available
- For PDE solvers (FEM/FVM assembly) on unstructured meshes, in both cases we have to care about data distribution
- We need explicit handling of data at interfaces with MPI, while with OpenMP, possible communication is hidden behind the common address space

Examination dates

Tue Feb 26.
Wed Feb 27.
Wed Mar 14.
Thu Mar 15.
Tue Mar 26.
Wed Mar 27.
Mon Apr 29.(?)
Tue Apr 30.(?)

Time: 10:00-13:00 (6 slots per examination date)

Please inscribe yourself into the corresponding sheets. (See also the back sides).

Room: t.b.a. (MA, third floor)

Prof. Nabben answers all administrative questions.

Please bring your yellow sheets 3 days before the examination to Frau Gillmeister

No lecture on Tue Jan 29!