Why parallelization?

- Computers became faster and faster without that...
- But: clock rate of processors limited due to physical limits
- ⇒ parallelization is the main road to increase the amount of data processed
- Parallel systems nowadays ubiquitous: even laptops and smartphones have multicore processors
- Amount of accessible memory per processor is limited ⇒ systems with large memory can be created based on parallel processors

TOP 500 2016 rank 1-6

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

TOP 500 2016 rank 7-13

Parallel paradigms

- SIMD
  - Single Instruction Multiple Data
  - “classical” vector systems: Cray, Convex...
  - Graphics processing units (GPU)

- MIMD
  - Multiple Instruction Multiple Data
  - Shared memory systems
  - IBM Power, Intel Xeon, AMD Opteron
  - Smartphones...
  - Xeon Phi
  - Distributed memory systems
  - interconnected CPUs

MIMD Hardware: Distributed memory

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

MIMD Hardware: Shared Memory

- Symmetric Multiprocessing (SMP)/Uniform memory access (UMA)
  - Possibly varying memory access latencies
  - Combination of SMP systems
  - ccNUMA: Cache coherent NUMA

Hybrid distributed/shared memory

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

Parallelization

Scientific Computing Winter 2016/2017
Part IV
With material by W. Gropp (http://wgropp.cs.illinois.edu) and J. Burkardt (https://people.sc.fsu.edu/ jburkardt)
Jürgen Fuhrmann
juergen.fuhrmann@wias-berlin.de
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.

Example:
```c
#include <thread>

int main() {
    std::mutex mtx;  // Lock to protect from concurrent access

    std::thread t[10];
    for(int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }

    for(int i = 0; i < num_threads; ++i) {
        t[i].join();  // Wait for all threads to complete
    }
}
```

- 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

Example: $u = au + v$ und $s = u \cdot v$
```c
#pragma omp parallel for
for(int i=0; i<n ; i++)
  u[i]+=a*u[i];
```

- Code can be parallelized by introducing compiler directives
- Compiler directives are ignored if not in parallel mode
- Write conflict with $+$: several threads may access the same variable
- In standard situations, reduction variables can be used to avoid conflicts

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
Parallelization of PDE solution

\[ \Delta u = f \text{ in } \Omega, \quad u|_{\partial\Omega} = 0 \]
\[ \Rightarrow u = \int_{\Omega} (f(y)G(x,y))dy. \]

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

Structured and unstructured grids

- Easy next neighbor access via index calculation
- Efficient implementation on SIMD/GPU
- Strong limitations on geometry
- General geometries
- Irregular, index vector based access to next neighbors
- Hardly feasible for SIMD/GPU

Mesh partitioning

Partition set of cells in \( T_h \), and color the graph of the partitions.

Result: \( C \): set of colors, \( P_c \): set of partitions of given color. Then:
\[ T_h = \bigcup_{c \in C} \bigcup_{P_c \in P_c} P_c \]

- 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
  - "tri" points \( \rightarrow \) Partitions of color 3

- No interference between assembly loops for partitions of the same color
- Immediate parallelization without critical regions

Linear system solution

- Sparse matrices
- Direct solvers are hard to parallelize though many efforts are undertaken
- Iterative methods easier to parallelize
  - partitioning of vectors \+ coloring inherited from cell partitioning
  -\( \Rightarrow \) 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 system (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)
- \[ \text{https://www.open-mpi.org/faq/?category=mpi-apps} \]
- Compiler wrapper \texttt{mpicc} \+ \texttt{mpiexec} \+ proper flags \+ libraries to be linked
- Process launcher \texttt{mpirun}
- Launcher starts a number of processes which execute statements independently, occasionally waiting for each other
Threads vs processes

- Threads are easier to create than processes since they don’t require a separate address space.
- Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, processes don’t share the same address space.
- Threads are considered lightweight because they use far less resources than processes.
- Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don’t step on each other.
- This is really another way of stating #2 above.
- A process can consist of multiple threads.
- MPI is based on processes, C++11 threads and OpenMP are based on threads.

MPI Hello world

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

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

// Create index vector for processes
std::vector<unsigned long> idx(nproc+1);

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

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

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

// The target process is specified by dest, which is the rank of the target process.
MPI_Bcast(start, count, datatype, root, comm);

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

// The target process is specified by dest, which is the rank of the target process.
MPI_Bcast(start, count, datatype, root, comm);

if (iproc == 0) cout << "The number of processes available is " << nproc << endl;

MPI_Finalize();
```

- Compile with mpic++ mpi-hello.cpp -o mpi-hello
- All MPI programs begin with MPI_Init() and end with MPI_Finalize() when 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 -n N mpi-hello

MPI hostfile

```
host1 slots=n1
host2 slots=n2
```

- Distribute code execution over several hosts.
- Need ssh public key access and common file system access for proper execution.

MPI Send

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

- The message buffer is described by (start, count, datatype).
- The message buffer can be reused. The message may not have been received by the target process.
- The tag codes some type of message.

MPI Receive

```c
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

```c
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.

SIMD Hardware: Graphics Processing Units (GPU)

- Principle useful for highly structured data.
- Example: textures, triangles for 3D graphics rendering.
- During the 90’s, Graphics Processing Units (GPUs) started to contain special purpose SIMD hardware for graphics rendering.
- 3D Graphic APIs (DirectX, OpenGL) became transparent to programmers: rendering could be influenced by “shaders” which essentially are programs which are compiled on the host and run on the GPU.
General Purpose Graphics Processing Units (GPGPU)
- Graphics companies like NVIDIA saw an opportunity to market GPUs for computational purposes
- Emerging APIs which allow to describe general purpose computing tasks for GPUs: CUDA (Nvidia specific), OpenCL (ATI/AMD designed, general purpose), OpenACC(future ?)
- GPGPUs are accelerator cards added to a computer with own memory and many vector processing pipelines (NVIDIA Tesla K40: 12GB + 2880 units)
- CPU-GPU connection generally via mainboard bus

GPU Programming paradigm
- CPU:
  - sets up data
  - triggers compilation of "kernels": the heavy duty loops to be executed on GPU
  - sends compiled kernels ("shaders") to GPU
  - sends data to GPU, initializes computation
  - receives data back from GPU
- GPU:
  - receive data from host CPU
  - just run the heavy duty loops on local memory
  - send data back to host CPU
- CUDA and OpenCL allow explicit management of these steps
- High efficiency only with good match between data structure and layout of GPU memory [2D rectangular grid]

OpenCL: Resource build up, kernel creation
- Set the arguments to our kernel, and enqueue it for execution
- Print a brief summary detailing the results

OpenCL: Kernel execution, result retrieval from GPU
- Validate our results
- Print a brief summary detailing the results

OpenCL: Data copy to GPU
- Allocate count and output buffers, and fill the input with data
- Copy our host buffer of random values to the input device buffer

OpenCL Summary
- Need good programming experience and system management skills in order to set up tool chains with properly matching versions, vendor libraries etc.
  - (I was not able to get this running on my laptop in finite time ...)
- Very cumbersome programming, at least as explicit as MPI
- Data structure restrictions limit class of tasks which can run efficiently on GPUs

Example: OpenCL: computational kernel

```
__kernel void square(
    __global float* input, __global float* output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

OpenACC (Open Accelerators)
- Idea similar to OpenMP, use compiler directives
- Future merge with OpenMP intended
- Intended for different accelerator types (GPU, Xeon Phi ...)
OpenACC Sample program

```c
#define N 2000000000
#define vl 1024
int main(void) {
  double pi = 0.f;
  long long i;
  #pragma acc parallel vector_length(vl)
  #pragma acc loop reduction(+:pi)
  for (i=0; i<N; i++) {
    double t= (double)((i+0.5)/N);
    pi +=4.0/(1.0+t*t);
  }
  printf("pi=%11.10f\n",pi/N);
  return 0;
}
```

.compile with
```
gcc-5 openacc.c -fopenacc -foffload=nvptx-none
-foffload="-O3" -O3 -o openacc-gpu
```

... but to do this one has to compile gcc with a special configuration...

Other ways to program GPU

- WebGL: directly use capabilities of graphics hardware via html, Javascript in the browser
  ```html```
  ```javascript```
  ```html```
  ```javascript```

- Example: Gray-Scott model for Reaction-Diffusion: two chemical species.
  - U is created with rate \( f \) and decays with rate \( f \)
  - U reacts with \( V \) to more \( V \)
  - \( U, V \) move by diffusion
  ```
  \[
  \begin{align*}
  &\frac{\partial u}{\partial t} - D_u \Delta u + u v^2 - f (1-u) = 0 \\
  &\frac{\partial v}{\partial t} - D_v \Delta v - u v^2 + (f + k) v = 0
  \end{align*}
  \]
  ```

The shader

```html```
```javascript```
```html```
```javascript```

Why does this work so well here?

- Data structure fits very well to topology of GPU
  - rectangular grid
  - 2 unknowns to be stored in x,y components of vec2
- GPU speed allows to "break" time step limitation of explicit Euler
- Data stay within the graphics card: once we loaded the initial value, all computations, and rendering use data which are in the memory of the graphics card.
- Depending on the application, choose the best way to proceed
  - e.g. deep learning (especially training speed)