Scientific Computing WS 2018/2019

Lecture 24

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]

Parallelization of PDE solution

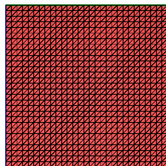
$$\Delta u = f \text{ in}\Omega, \qquad u|_{\partial\Omega} = 0$$

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

- 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

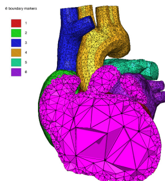
Structured and unstructured grids

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 $m, n = 0 \dots d$:

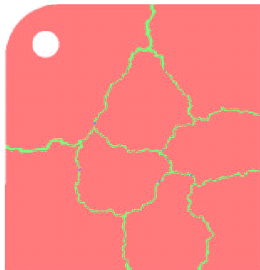
$$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \; dx$$
$$a_{j_{dof}(K,m), j_{dof}(K,n)} = a_{j_{dof}(K,m), j_{dof}(K,n)} + s_{mn}$$

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) $\to$ Partitions of color 1
  - Create separators along boundaries $\to$ Partitions of color 2
  - "triple points" $\to$ 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 $m, n = 0 \dots d$:
            $s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \, dx$
            $a_{j_{dof}(K,m), j_{dof}(K,n)} += s_{mn}$

- 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

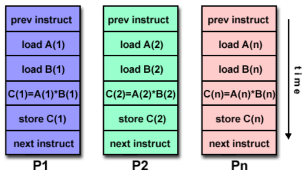
TOP 500 2018 rank 7-13

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

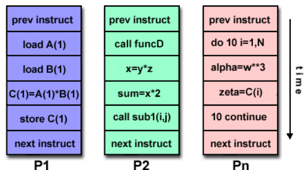
SIMD
Single Instruction Multiple Data



[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

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

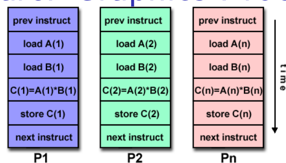
```
#include <pthread.h>
void *PrintHello(void *threadid)
{  long tid = (long)threadid;
  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;   long t;
  for(t=0; t<NUM_THREADS; t++)
  {
    printf("In 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
```

SIMD Hardware: Graphics Processing Units (GPU)

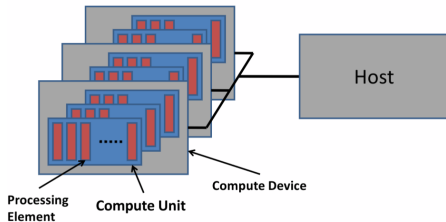


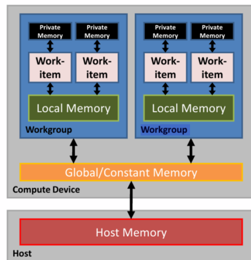
[Source: computing.llnl.gov/tutorials]

- Principle useful for highly structured data
- Example: textures, triangles for 3D graphis 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 based on compiler directives
- GPGPUs are *accelerator cards* added to a computer with own memory, many vector processing pipelines and special bus interconnect (NVidia Quadro GV100: 32GB +5120 units, NVLink; Tensor cores)
- CPU-GPU connection via mainbord bus / special link





[Source: amd-dev.wpengine.netdna-cdn.com]

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
  - Run the heavy duty loops in local memory
  - Send data back to host CPU
- For high performance one needs explicit management of these steps
- Bottleneck: Data transfer CPU $\leftrightarrow$ GPU
- High effiency only with good match between data structure and layout of GPU memory (2D rectangular grid)

NVIDIA Cuda

- Established by NVIDIA GPU vendor
- Works only on NVIDIA cards
- Claimed to provide optimal performance

CUDA Kernel code

- The kernel code is the code to be executed on the GPU aka “Device”
- It needs to be compiled using special CUDA compiler

```
#include <cuda_runtime.h>

/*
 * CUDA Kernel Device code
 *
 * Computes the vector addition of A and B into C.
 * The 3 vectors have the same
 * number of elements numElements.
 */
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

CUDA Host code I

```
int main(void)
{   int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // Allocate host vectors
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);

    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
    // Allocate device vectors
    float *d_A = NULL;
    float *d_B = NULL;
    float *d_C = NULL;
    assert(cudaMalloc((void **)&d_A, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_B, size)==cudaSuccess);
    assert(cudaMalloc((void **)&d_C, size)==cudaSuccess);
    ...
```

CUDA Host code II

```
...

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1)
                   / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

assert(cudaGetLastError()==cudaSuccess);
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

free(h_A);
free(h_B);
free(h_C);
cudaDeviceReset();
```

OpenCL

- "Open Computing Language"
- Vendor independent
- More cumbersome to code

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];
}
```

Declare functions with **__kernel** attribute

Defines an entry point or exported method in a program object

Use address space and usage qualifiers for memory

Address spaces and data usage must be specified for all memory objects

Built-in methods provide access to index within compute domain

Use **get_global_id** for unique work-item id, **get_group_id** for work-group, etc

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

OpenCL: Resource build up, kernel creation

```
// Fill our data set with random float values
int count = 1024 * 1024;
for(i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;

// Connect to a compute device, create a context and a command queue
cl_device_id device;
clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(0, 1, & device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);

// Create and build a program from our OpenCL-C source code
cl_program program = clCreateProgramWithSource(context, 1, (const char **) &src,
                                               NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create a kernel from our program
cl_kernel kernel = clCreateKernel(program, "square", NULL);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

OpenCL: Data copy to GPU

```
// Allocate input and output buffers, and fill the input with data
cl_mem input = clCreateBuffer(context,  CL_MEM_READ_ONLY,  sizeof(float) * count,
                              NULL, NULL);

// Create an output memory buffer for our results
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count,
                               NULL, NULL);

// Copy our host buffer of random values to the input device buffer
clEnqueueWriteBuffer(queue, input, CL_TRUE, 0, sizeof(float) * count, data, 0,
                     NULL, NULL);

// Get the maximum number of work items supported for this kernel on this device
size_t global = count; size_t local = 0;
clGetKernelWorkGroupInfo(kernel, device, CL_KERNEL_WORK_GROUP_SIZE, sizeof(int),
                         &local, NULL);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

OpenCL: Kernel execution, result retrieval from GPU

```
// Set the arguments to our kernel, and enqueue it for execution
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global, &local, 0, NULL, NULL);

// Force the command queue to get processed, wait until all commands are complete
clFinish(queue);

// Read back the results
clEnqueueReadBuffer( queue, output, CL_TRUE, 0, sizeof(float) * count, results, 0,
                     NULL, NULL );

// Validate our results
int correct = 0;
for(i = 0; i < count; i++)
    correct += (results[i] == data[i] * data[i]) ? 1 : 0;

// Print a brief summary detailing the results
printf("Computed '%d/%d' correct values!\n", correct, count);
```

[Source: http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-overview.pdf]

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.

Compiler directive based GPU programming

- OpenMP
  - OpenMP4.0
  - Implentation in commercial compilers
  - GCC, Clang implementations under development
- OpenACC
  - Idea similar to OpenMP: use compiler directives
  - Future merge with OpenMP initially intended, now they seem to be competitors
  - Intended for different accelerator types (Nvidia GPU ...)
  - Commercial compiler vendors, e.g. PGI (with free academic license valid one year)
  - GCC, Clang implementations under development

OpenACC code

- "Shader":

```
void vecaddgpu( float *restrict r, float *a, float *b, int n, int nrepeat)
{
  int irepeat;
  #pragma acc kernels loop present(r,a,b)
  for (irepeat=0;irepeat<nrepeat; irepeat++)
  for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i] + irepeat;
}
```

- Invocation from CPU

```
a = (float*)malloc( n*sizeof(float) );
b = (float*)malloc( n*sizeof(float) );
r = (float*)malloc( n*sizeof(float) );
e = (float*)malloc( n*sizeof(float) );
#pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
{
  vecaddgpu( r, a, b, n, nrepeat );
}
```

- Compile with PGI compiler (https://www.pgroup.com/)

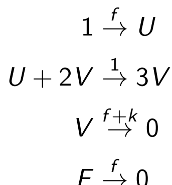
```
pgcc -ta=tesla -fast -o add2 add2.c
```

Other ways to program GPU

- Directly use graphics library
- Modern OpenGL with shaders
- WebGL: OpenGL in the browser. Uses html and javascript.

WebGL Example

- Gray-Scott model for Reaction-Diffusion: two species.
  - U is created with rate f and decays with rate f
  - U reacts wit V to more V
  - V deacays with rate $f + k$.
  - U, V move by diffusion



$$1 \xrightarrow{f} U$$

$$U + 2V \xrightarrow{1} 3V$$

$$V \xrightarrow{f+k} 0$$

$$F \xrightarrow{f} 0$$

- Stable states:
  - No V
  - " Much of V, then it feeds on U and re-creates itself
- Reaction-Diffusion equation from mass action law:

$$\partial_t u - D_u \Delta u + uv^2 - f(1-u) = 0$$

$$\partial_t v - D_v \Delta v - uv^2 + (f+k)v = 0$$

Discretization

- ... GPUs are fast so we choose the explicit Euler method:

$$\frac{1}{\tau}(u_{n+1} - u_n) - D_u \Delta u_n + u_n v_n^2 - f(1 - u_n) = 0$$
$$\frac{1}{\tau}(v_{n+1} - u_v) - D_v \Delta v_n - u_n v_n^2 + (f + k) v_n = 0$$

- Finite difference/finite volume discretization on grid of size h

$$-\Delta u \approx \frac{1}{h^2}(4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1})$$

The shader

```
<script type="x-webgl/x-fragment-shader" id="timestep-shader">
precision mediump float;
uniform sampler2D u_image;
uniform vec2 u_size;
const float F = 0.05, K = 0.062, D_a = 0.2, D_b = 0.1;
const float TIMESTEP = 1.0;
void main() {
vec2 p = gl_FragCoord.xy,
     n = p + vec2(0.0, 1.0),
     e = p + vec2(1.0, 0.0),
     s = p + vec2(0.0, -1.0),
     w = p + vec2(-1.0, 0.0);

vec2 val = texture2D(u_image, p / u_size).xy,
     laplacian = texture2D(u_image, n / u_size).xy
    + texture2D(u_image, e / u_size).xy
    + texture2D(u_image, s / u_size).xy
    + texture2D(u_image, w / u_size).xy
    - 4.0 * val;

vec2 delta = vec2(D_a * laplacian.x - val.x*val.y*val.y + F * (1.0-val.x),
    D_b * laplacian.y + val.x*val.y*val.y - (K+F) * val.y);

gl_FragColor = vec4(val + delta * TIMESTEP, 0, 0);
}
</script>
```

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
- No communication with CPU in the first place
- 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)

Examinations

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

- Due to illness of Prof. Nabben, I can confirm new dates only next week.
- 13:00 times do **not** work! Please reschedule (sorry).