Addendum + Parallel Assembly

Scientific Computing Winter 2016/2017

Lecture 27

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

made wit pandoc

Convergence tests

Error estimates for homogeneous Dirichlet problem

- Search $u \in H_0^1(\Omega)$ such that

$$\int_\Omega \lambda \nabla u \nabla v \, dx = \int_\Omega fv \, dx \; \forall v \in H_0^1(\Omega)$$

- Then, $\lim_{h \to 0} ||u - u_h||_{1,\Omega} = 0$.
- If $u \in H^2(\Omega)$ (e.g. convex domain, smooth coefficients), then

$$||u - u_h||_{1,\Omega} \leq ch|u|_{2,\Omega} \leq c'h|f|_{0,\Omega}$$
$$||u - u_h||_{0,\Omega} \leq ch^2|u|_{2,\Omega} \leq c'h^2|f|_{0,\Omega}$$

and ("Aubin-Nitsche-Lemma")

$$||u - u_h||_{0,\Omega} \leq ch|u|_{1,\Omega}$$

Stiffness matrix calculation 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 $m, n = 0 \ldots d$:

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

Local stiffness matrix calculation for P1 FEM

$a_0 \dots a_d$: vertices of the simplex K, $a \in K$.

Barycentric coordinates: $\lambda_j(a) = \frac{|K_j(a)|}{|K|}$

For indexing modulo d+1 we can write

$$|K| = \frac{1}{d!} \det\left(a_{j+1} - a_j, \dots a_{j+d} - a_j\right)$$
$$|K_j(a)| = \frac{1}{d!} \det\left(a_{j+1} - a, \dots a_{j+d} - a\right)$$

From this information, we can calculate $\nabla\lambda_j(x)$ (which are constant vectors due to linearity) and the corresponding entries of the local stiffness matrix

$$s_{ij} = \int_K \nabla\lambda_i \nabla\lambda_j \, dx$$

Local stiffness matrix calculation for P1 FEM in 2D

$a_0 = (x_0, y_0) \ldots a_d = (x_2, y_2)$: vertices of the simplex K, $a = (x, y) \in K$.

Barycentric coordinates: $\lambda_j(x, y) = \frac{|K_j(x,y)|}{|K|}$

For indexing modulo d+1 we can write

$$|K| = \frac{1}{2} \det \begin{pmatrix} x_{j+1} - x_j & x_{j+2} - x_j \\ y_{j+1} - y_j & y_{j+2} - y_j \end{pmatrix}$$

$$|K_j(x, y)| = \frac{1}{2} \det \begin{pmatrix} x_{j+1} - x & x_{j+2} - x \\ y_{j+1} - y & y_{j+2} - y \end{pmatrix}$$

Therefore, we have

$$|K_j(x, y)| = \frac{1}{2} \left((x_{j+1} - x)(y_{j+2} - y) - (x_{j+2} - x)(y_{j+1} - y) \right)$$

$$\partial_x |K_j(x, y)| = \frac{1}{2} \left((y_{j+1} - y) - (y_{j+2} - y) \right) = \frac{1}{2} (y_{j+1} - y_{j+2})$$

$$\partial_y |K_j(x, y)| = \frac{1}{2} \left((x_{j+2} - x) - (x_{j+1} - x) \right) = \frac{1}{2} (x_{j+2} - x_{j+1})$$

Local stiffness matrix calculation for P1 FEM in 2D II

$$s_{ij} = \int_K \nabla\lambda_i \nabla\lambda_j \, dx = \frac{|K|}{4|K|^2} \left(y_{i+1} - y_{i+2}, x_{i+2} - x_{i+1}\right) \begin{pmatrix} y_{j+1} - y_{j+2} \\ x_{j+2} - x_{j+1} \end{pmatrix}$$

So, let $V = \begin{pmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{pmatrix}$

Then

$$x_1 - x_2 = V_{00} - V_{01}$$
$$y_1 - y_2 = V_{10} - V_{11}$$

and

$$2|K| \, \nabla\lambda_0 = \begin{pmatrix} y_1 - y_2 \\ x_2 - x_1 \end{pmatrix} = \begin{pmatrix} V_{10} - V_{11} \\ V_{01} - V_{00} \end{pmatrix}$$
$$2|K| \, \nabla\lambda_1 = \begin{pmatrix} y_2 - y_0 \\ x_0 - x_2 \end{pmatrix} = \begin{pmatrix} V_{11} \\ -V_{01} \end{pmatrix}$$
$$2|K| \, \nabla\lambda_2 = \begin{pmatrix} y_0 - y_1 \\ x_1 - x_0 \end{pmatrix} = \begin{pmatrix} -V_{10} \\ V_{00} \end{pmatrix}$$

Test problem

- Homogeneous Dirichlet problem:

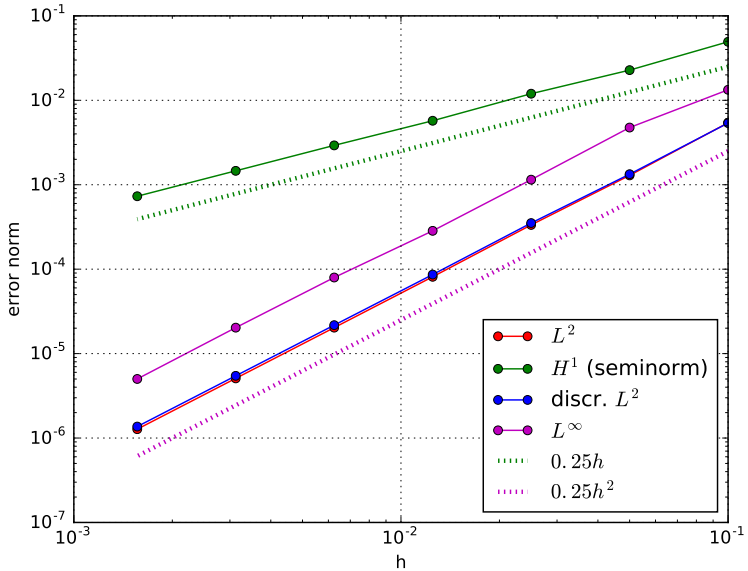
$$-\Delta u = 2\pi^2 \sin(\pi x)\sin(\pi y) \text{ in } \Omega = (0,1)\times(0,1)$$
$$u|_{\partial\Omega} = 0$$

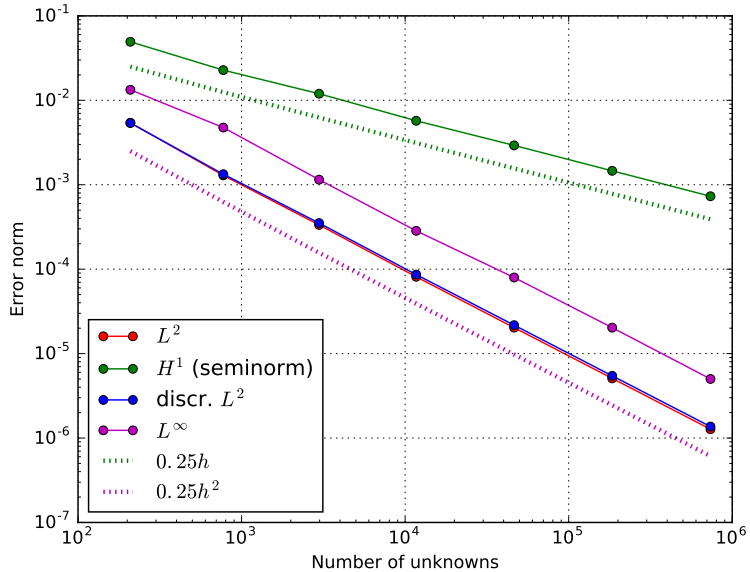
- Exact solution:

$$u(x,y) = \sin(\pi x)\sin(\pi y)$$

- Testing approach: generate series of finer grids with `triangle`, by control the triangle are parameter acoording to the desired mesh size h.
- Do we get the theoretical error estimates ?
- We did not talk about error estimates for the finite volume method. What can we expect ?
- For simplicity, we calculate not $||u_{exact} - u_h||$ but $\Pi_h u_{exact} - u_h||$ where Π_h is the P1 nodal interpolation operator.
- More precise test would have to involve high order quadrature for calculation of the norm.

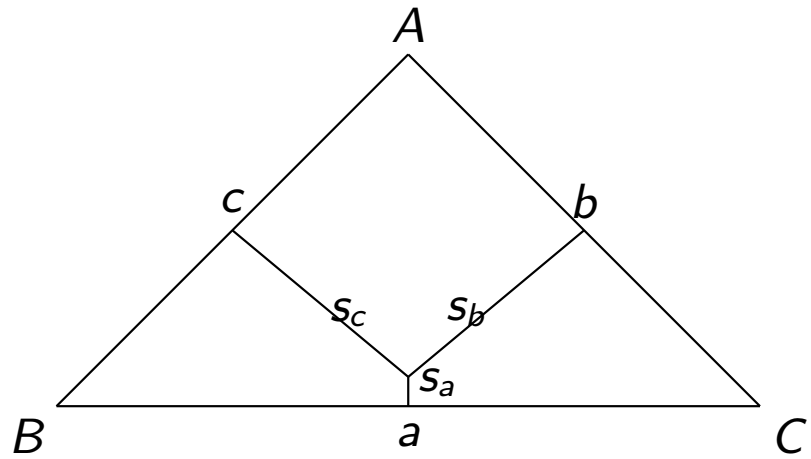
FEM Results





- Theoretical estimates are reproduced
- Useful test for debugging code...

Finite volume local stiffness matrix calculation I



Triangle edge lengths:

$$a, b, c$$

Semiperimeter:

$$s = \frac{a}{2} + \frac{b}{2} + \frac{c}{2}$$

Square area (from Heron's formula):

$$16A^2 = 16s(s-a)(s-b)(s-c) = (-a+b+c)(a-b+c)(a+b-c)(a+b+c)$$

Square circumradius:

$$R^2 = \frac{a^2b^2c^2}{(-a+b+c)(a-b+c)(a+b-c)(a+b+c)} = \frac{a^2b^2c^2}{16A^2}$$

Finite volume local stiffness matrix calculation II

Square of the Voronoi surface contribution via Pythagoras:

$$s_a^2 = R^2 - \left(\frac{1}{2}a\right)^2 = -\frac{a^2\left(a^2-b^2-c^2\right)^2}{4\left(a-b-c\right)\left(a-b+c\right)\left(a+b-c\right)\left(a+b+c\right)}$$

Square of edge contribution in the finite volume method:

$$e_a^2 = \frac{s_a^2}{a^2} = -\frac{\left(a^2-b^2-c^2\right)^2}{4\left(a-b-c\right)\left(a-b+c\right)\left(a+b-c\right)\left(a+b+c\right)}$$

Comparison with pdelib formula:

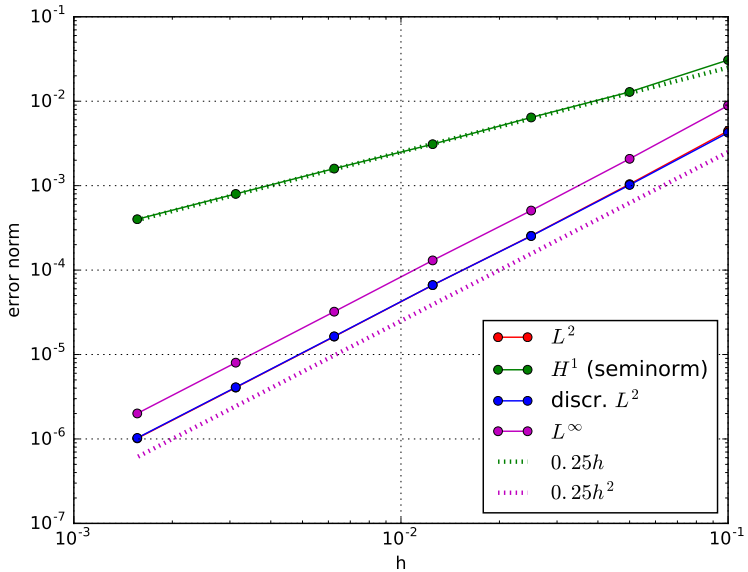
$$e_a^2 - \frac{(b^2+c^2-a^2)^2}{64A^2} = 0$$

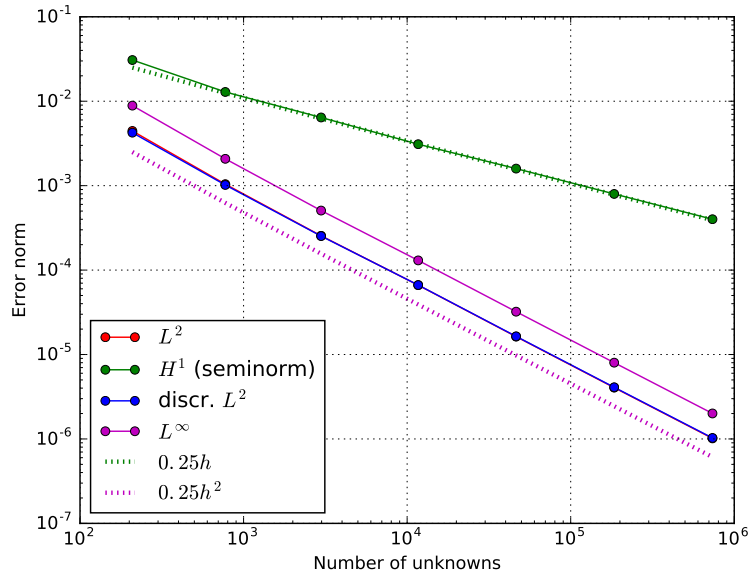
This implies the formula for the edge contribution

$$e_a = \frac{s_a}{a} = \frac{b^2+c^2-a^2}{8A}$$

The sign chosen implies a positive value if the angle $\alpha < \frac{\pi}{2}$, and a negative value if it is obtuse. In the latter case, this corresponds to the negative length of the line between edge midpoint and circumcenter, which is exactly the value which needs to be added to the corresponding amount from the opposite triangle in order to obtain the measure of the Voronoi face.

FVM Results





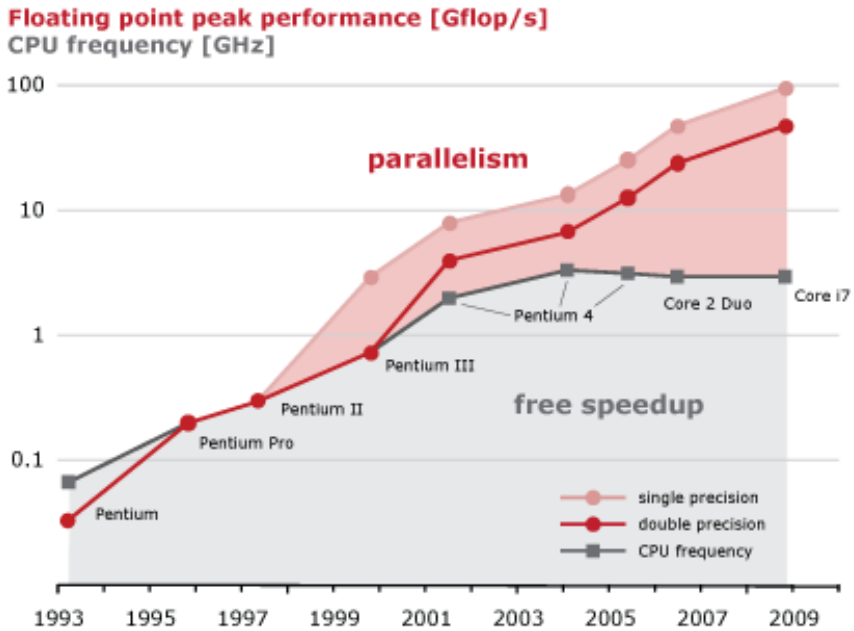
- Similar results as for FEM

~

Recap

Why parallelization ?

- Computers became faster and faster without that. . .

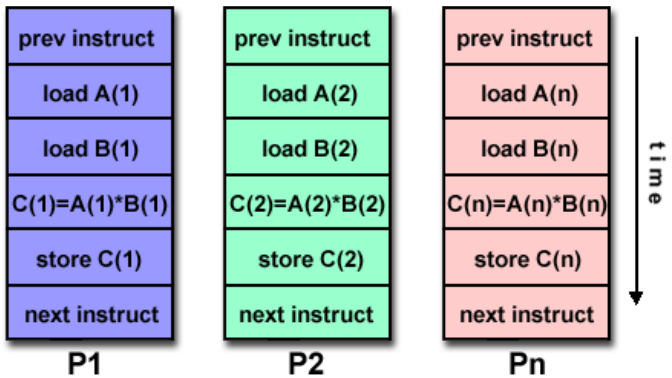


[Source: spiralgen.com]

- But: clock rate of processors limited due to physical limits
- $\Rightarrow$ 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 $\Rightarrow$ systems with large memory can be created based on parallel processors

Parallel paradigms

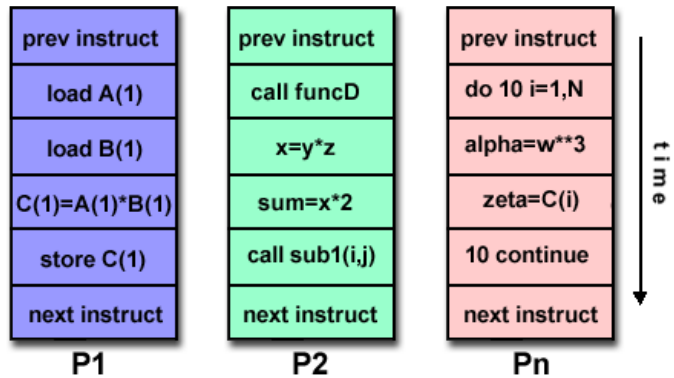
SIMD
Single Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

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

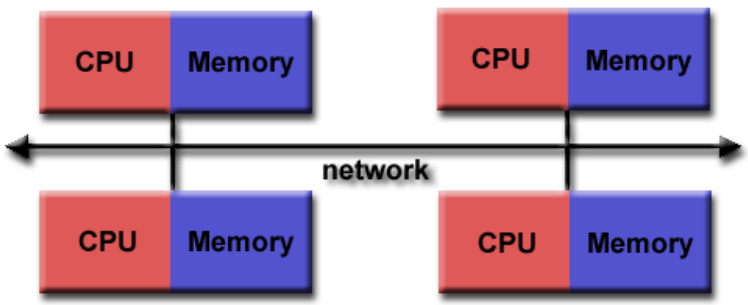
MIMD
Multiple Instruction Multiple Data



[Source: computing.llnl.gov/tutorials]

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

MIMD Hardware: Distributed memory



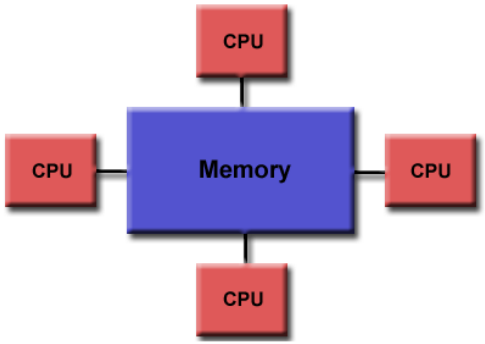
[Source: computing.llnl.gov/tutorials]

- "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

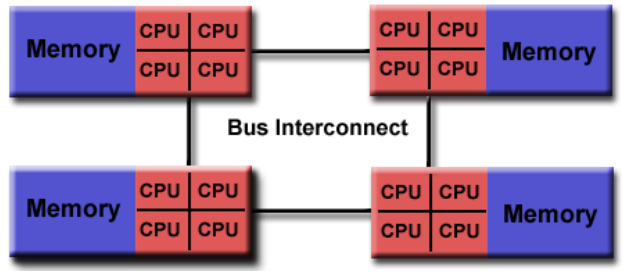
Symmetric Multiprocessing (SMP)/Uniform memory acces (UMA)



[Source: computing.llnl.gov/tutorials]

- Similar processors
- Similar memory access times

Nonuniform Memory Access (NUMA)



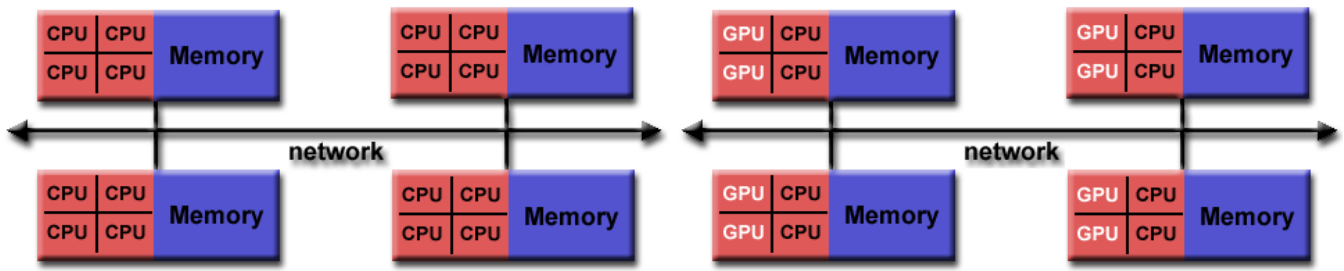
[Source: computing.llnl.gov/tutorials]

- 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 more high 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 intersting 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 the 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 syncronized
- Mutexes allow to define regions in a program which are accessed by all threads in a sequential manner.

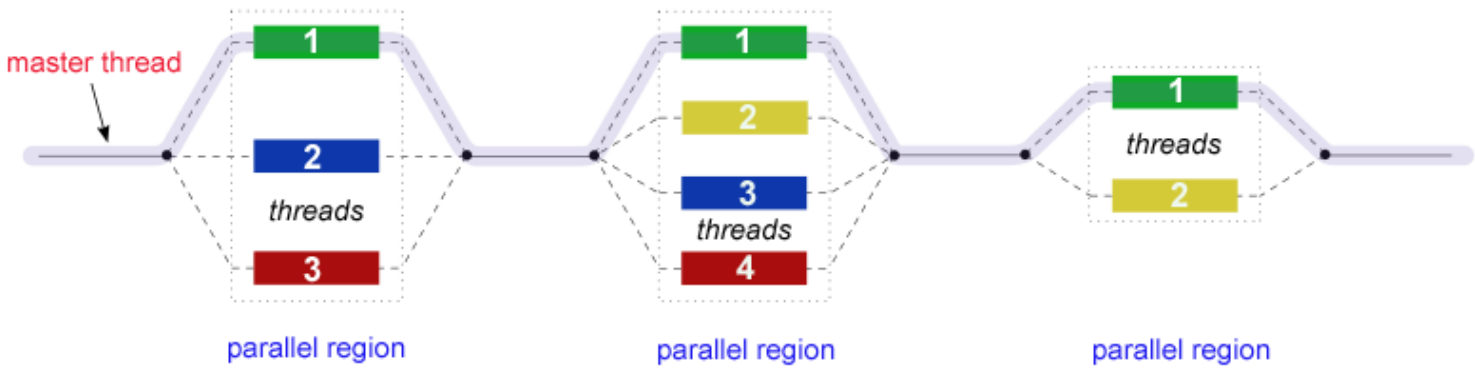
```
#include <iostream>
#include <thread>
#include <mutex>
std::mutex mtx;
void call_from_thread(int tid) {
    mtx.lock()
    std::cout << "Launched by thread " << tid << std::endl;
    mtx.unlock()
}
int main() {
    std::thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    std::cout << "Launched from the main\n";
    //Join the threads with the main thread
    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 reduction(+:s)
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
- In standard situations, reduction variables can be used to avoid conflicts

Do it yourself reduction

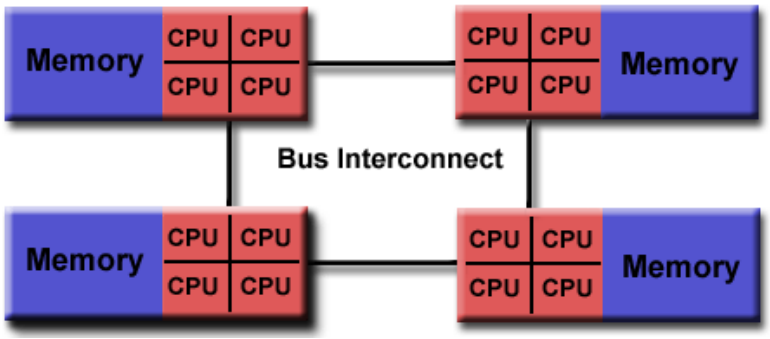
```
#include <omp.h>
int maxthreads=omp_get_max_threads();
double s0[maxthreads];
double u[n],v[n];
for (int ithread=0;ithread<maxthreads; ithread++)
  s0[ithread]=0.0;

#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: further aspects

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

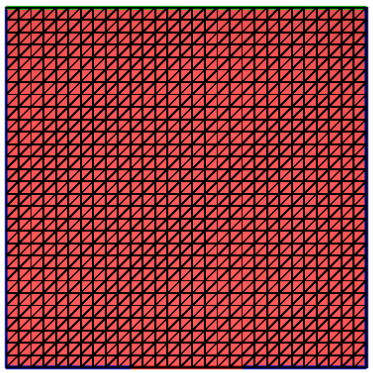


[Quelle: computing.llnl.gov/tutorials]

- 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

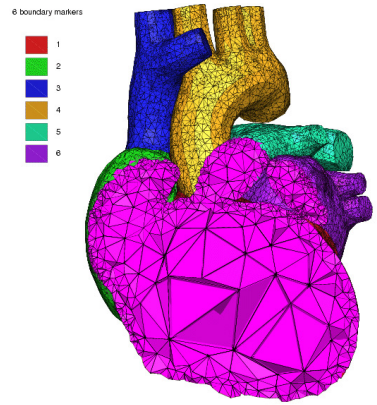
Structured and unstructured grids

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 $m, n = 0 \ldots d$:

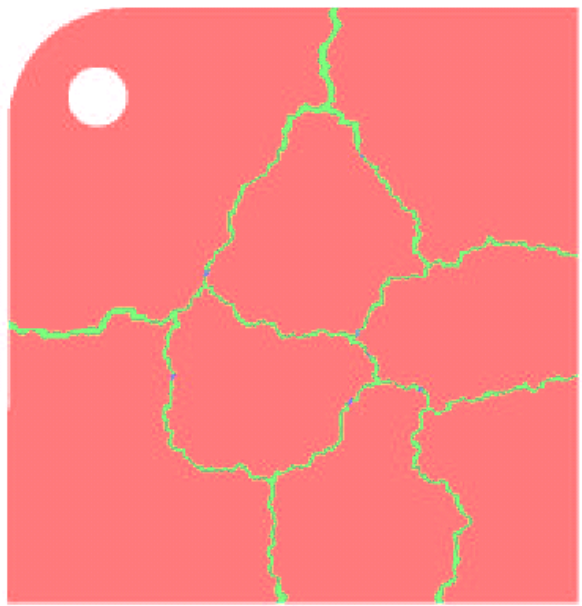
$$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \; dx$$
$$a_{j_{dof}(K,m), j_{dof}(K,n)} = a_{j_{dof}(K,m), j_{dof}(K,n)} + s_{mn}$$

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



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

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 $m, n = 0 \dots d$:
$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \, dx$
$a_{j_{dof}(K,m), j_{dof}(K,n)} += s_{mn}$

- 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
- 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