Parallelization using MPI

With material by W. Gropp (http://wgropp.cs.illinois.edu) and J. Burkardt (https://people.sc.fsu.edu/ jburkardt)

Scientific Computing Winter 2016/2017

Lecture 28

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

made wit pandoc

Recap

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")

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

Test problem

- Homogeneous Dirichlet problem:

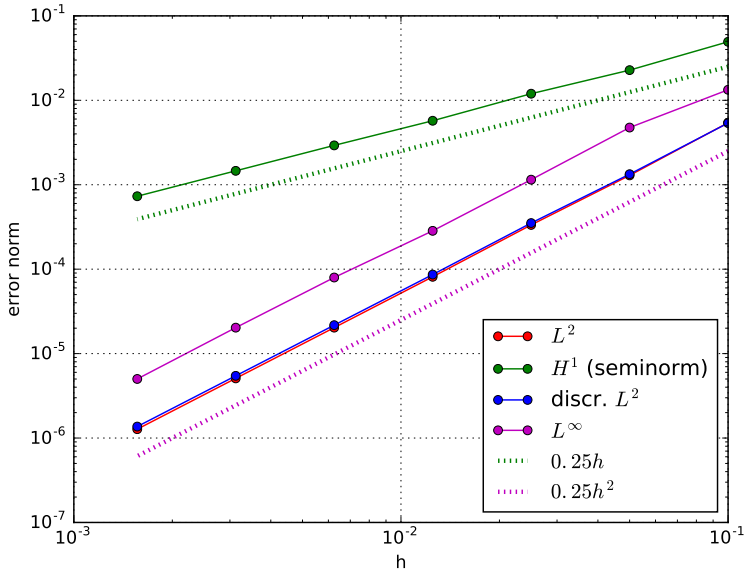
$$-\Delta u = 2\pi^2 \sin(\pi x)\sin(\pi y) \text{ in } \Omega = (0,1)\times(0,1)$$
$$u|_{\partial\Omega} = 0$$

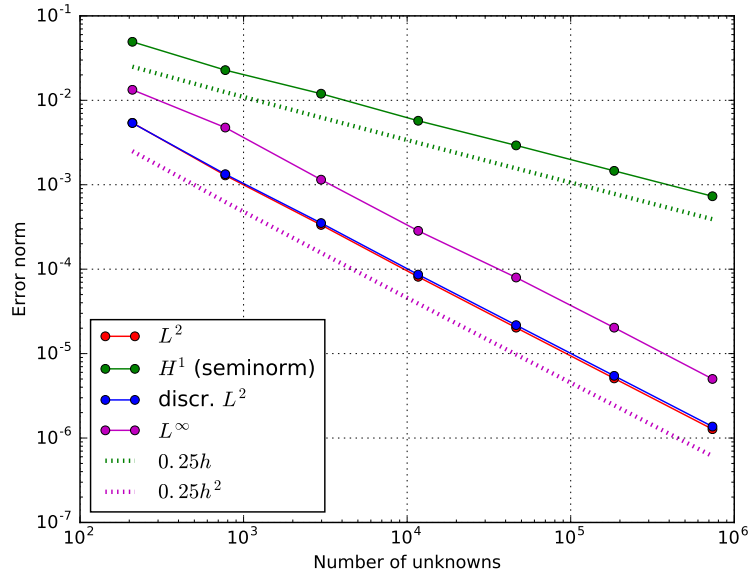
- Exact solution:

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

- 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.

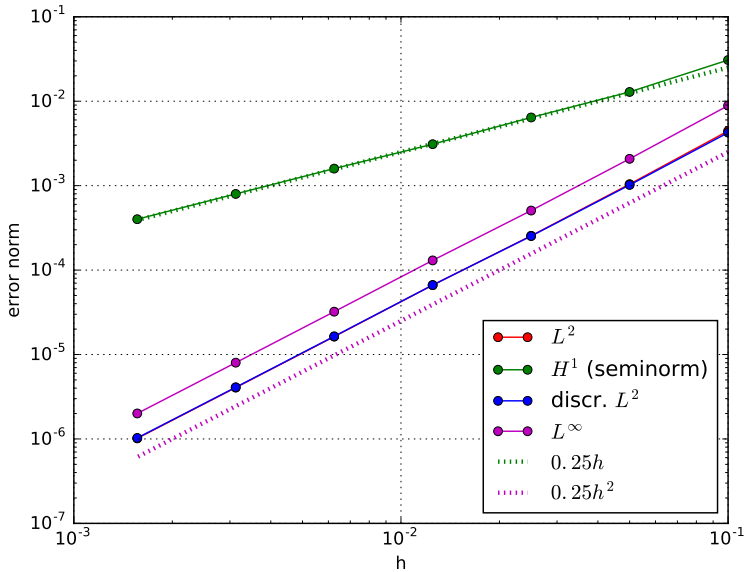
FEM Results

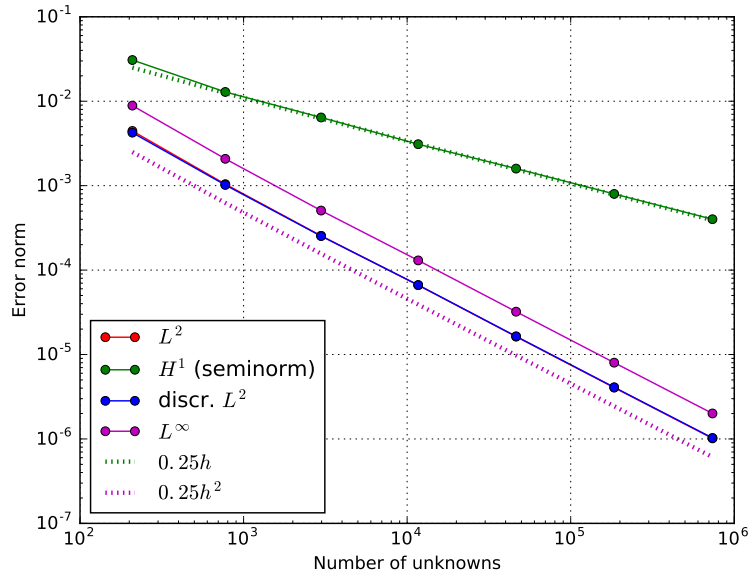




- Theoretical estimates are reproduced
- Useful test for debugging code. . .

FVM Results



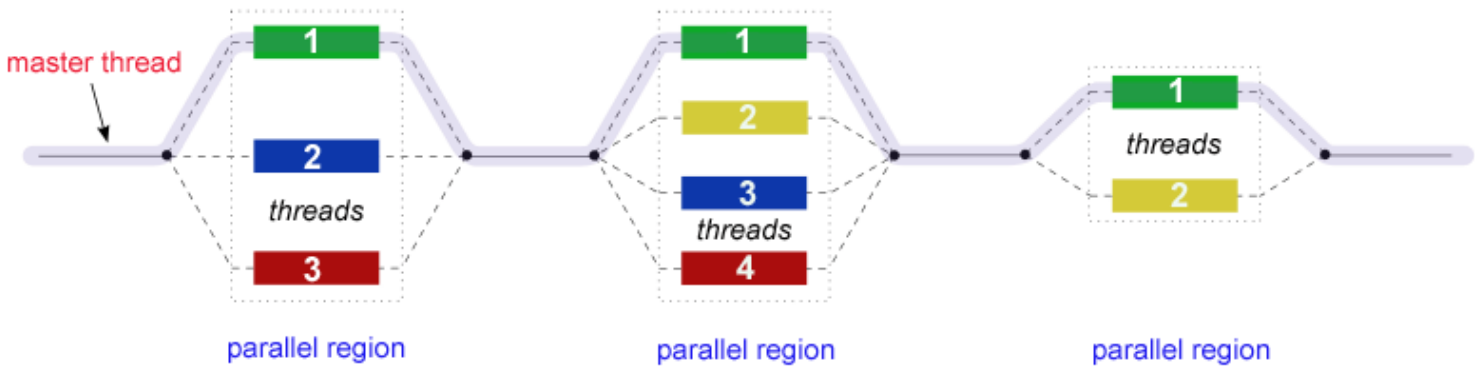


- Similar results as for FEM

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]

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

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 $m, n = 0 \ldots d$:

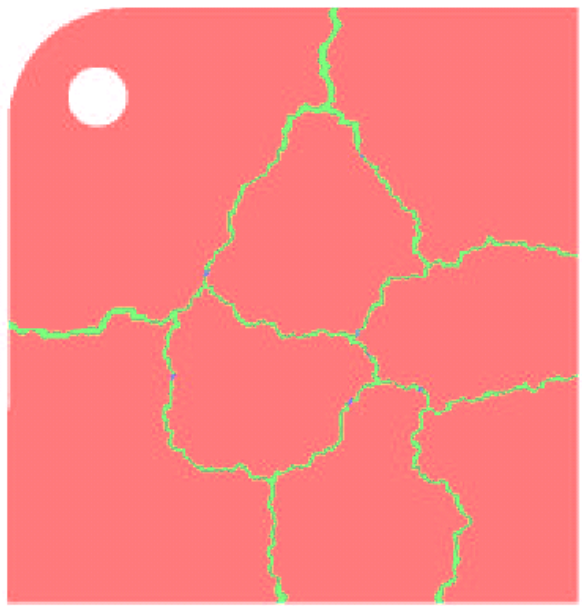
$$s_{mn} = \int_K \nabla\lambda_m \nabla\lambda_n \; dx$$
$$a_{j_{dof}(K,m), j_{dof}(K,n)} = a_{j_{dof}(K,m), j_{dof}(K,n)} + s_{mn}$$

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

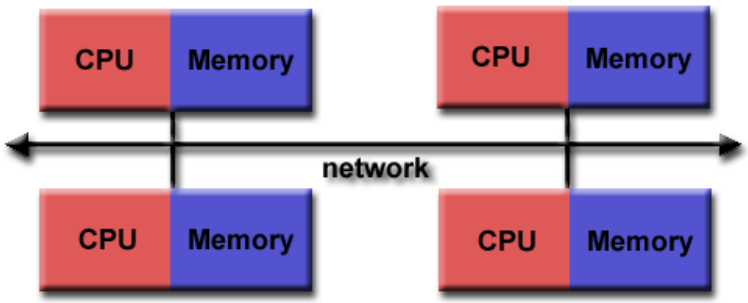
Parallel stiffness matrix assembly for Laplace operator for P1 FEM

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

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 - Message passing interface

- library, can be used from C,C++, Fortran, python
- de facto standard for programming on distributet 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)
  - `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

- Threads are easier to create than processes since they don't require a separate address space.
- Multithreading requires careful programming since threads share data strucures 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

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

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

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

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

if ( iproc == 0 )    cout << "The number of processes available is " << 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 -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 acces for proper executionx

MPI Send

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

- The message buffer is described by (start, count, datatype)
- 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