Distributed Memory Parallelization in NGSolve

Lukas Kogler
June, 2017

Inst. for Analysis and Scientific Computing, TU Wien
From Shared to Distributed Memory
Parallelization via threads (→ unit for scheduling).
Programming model:
Parallelization via **threads** (→ unit for **scheduling**).
Programming model:

![Diagram of shared memory with threads]

Assumption:
All threads can access **all** parts of the memory equally fast.
Shared Memory

Parallelization via **threads** (→ unit for **scheduling**).
Programming model:

Assumption:
All threads can access all parts of the memory **equally fast**.
Programming model that better fits a cluster:

- Distributed Memory

- Parallelization via procs (→ unit for scheduling and memory management).
Programming model that better fits a cluster:

Parallelization via \texttt{procs} (→ unit for scheduling \textbf{and} memory management).
From Shared to Distributed Memory

MPI

NGS+MPI
  Principles
  Under the hood

Using MPI-parallel NGSolve

Solvers and Preconditioners

Experiences

Bonus slide; VTK + Paraview
MPI
A **standard** for message-passing **between independent processes**; implemented in multiple libraries.

De facto standard for numerical computations.

It features functions for Point-to-Point communication between procs, by sending and receiving of ”messages”.

**Portability** is a huge aspect - write it once, run it on a laptop or a supercomputer!

Also: collective communication, one-sided communication, …
Point-to-Point communication

Process identified by **rank**, a number from 0 to NP.

To exchange message, one has to specify: pointer to data, amount of data, type of data, source/destination, ...

On the sending end:

```c
Array<double> send_values(len);
MPI_Send(&send_values[0], len, MPI_DOUBLE, recv_proc, ...);
```

On the receiving end:

```c
Array<double> recv_values(len);
MPI_Recv(&recv_values[0], len, MPI_DOUBLE, send_proc, ...);
```
NGS+MPI
Distributing the Mesh
Distributing the Mesh

- **non-overlapping** partition (no ghost-elements)
- shared interface-nodes (edges in 2d, faces & edges in 3d)
- proc 0 (master proc) gets nothing!
Distributing the FESpace

A finite element consists of a space $V_{h,i}$ and a basis of its dual space $(\phi_{i,j})_j$. The FESpace is then "glued together":

$$V_h = \left\{ u \in \Pi_i V_{h,i} : \phi(u_i) = \phi(u_k) \quad \forall \phi \in V'_{h,i} \cap V'_{h,j} \right\}$$

→ Treat each subdomain as a "makro - Finite Element"!

Subdomain-spaces are glued together by functionals corresponding to nodes on interfaces.
Parallel Matrices

- Assemble element-matrices to submatrices $A_i$ on each subdomain.
- The global Matrix $A = \sum_i E_i A_i E_i^T$ (with embeddings $E_i$) is never assembled.
- No communication needed to assemble matrices!

Parallel Vectors

- Local vector on each subdomain
- Can be CUMULATED (consistent values) or DISTRIBUTED (partial values)

\[ x_j^{(C)} = E_j^T \sum_i E_i x_i^{(D)} \]

- DISTRIBUTED $\rightarrow$ CUMULATED requires communication!
\[(Ax)^{(C)}_j = E^T_j \sum_i E_i A_i x_i^{(C)} = E^T_j \sum_i E_i (Ax)^{(D)}_i\]

\[A_i x_j^{(C)} = (Ax)^{(D)}_j\]

Multiplication with local matrices takes a $C$-vector and returns a $D$-vector, to make it a $C$-vector again, we need communication!
We want to **apply** a (possibly nonlinear) operator involving dg-jumps, e.g:

```python
a = BilinearForm(V)
a += SymbolicBFI((u−u.Other())*(v−v.Other()), skeleton=True)
a.Apply(u.vec, res)
```

remember: **no ghost elements** → what is ”.Other()” on a subdomain interface??
• Only need to integrate \((u-u.\text{Other}) \times v\)
• Only need to integrate \((u-u.\text{Other()}) \times v\)
• We know ”both sides” have the same expression
• Only need to integrate \((u-u.\text{Other()}) \times v\)
• We know ”both sides” have the same expression
• Need trace values in IPs of all trial-proxies ”from the other side”!
• Only need to integrate \((u-u.\text{Other()}) \times v\)
• We know ”both sides” have the same expression
• Need trace values in IPs of all trial-proxies ”from the other side”!
• ”Other side” needs trace values of all trial-proxies from me!

→ Exchange trial-proxy traces!
Table<double> tv_send;  //np rows
Table<double> tv_recv;  //np rows

for (facet : mpi_facet)
    CalcTraceValues(facet, tv_send[p][facet_part])

for (p : Range(0,np) ) {
    MPI_Isend(tv_send[p],p,...)  // send trace values
    MPI_Irecv(tv_recv[p],p,...)  // recv trace values
}

MPI_Waitall(...)  // wait for communication to finish

for (facet : mpi_facet)
    ApplyFromTraceValues(my_el, tv_recv[p][facet_part])
Typical for programing with MPI:

- understand the problem
- understand the exact data dependencies
- simplify & reformulate to minimize those
- implement
Using MPI-parallel NGSolve
Ideally, only need to switch on MPI with CMake option -DUSE_MPI=ON.
In fact, on clusters, this can be a bit more messy.
Generally, we try to hide MPI as well as possible, however, in fact, the user has to do some stuff differently.

Things to consider:

- meshing is sequential only
- the netgen-ui is shm-parallel only (→ use VTK+ParaView)
- you have to take care of parallelizing anything that happens outside the ngsolve-libraries (e.g. shell in- and output)
- any other considerations for running large jobs (e.g. workload-manager)
- not quite as feature rich as shm-parallel version
ngspy - wrapper around python3 In the simplest case:

$ ngspy some_script.py
ngspy - wrapper around python3 In the simplest case:

$ mpirun -np 272 ngspy some_script.py
**ngspy** - wrapper around python3 In the simplest case:

```shell
$ mpirun -np 272 ngspy some_script.py
```

On large clusters → usually a workload-manager (SLURM, ...)

```shell
$ sbatch job_script
```
**ngspy** - wrapper around python3 In the simplest case:

```bash
$ mpirun -np 272 ngspy some_script.py
```

On large clusters → usually a workload-manager (SLURM,...)

```bash
$ sbatch job_script
```

```bash
#!/usr/bin/bash
#
#SBATCH --job-name=ngspy
#SBATCH --partition=phi
#SBATCH -N 10
#SBATCH --ntasks 640
#SBATCH --ntasks-per-node=64
#SBATCH --ntasks-per-core=1

mpirun ngspy mpi_poisson.py
```
Solvers and Preconditioners
Own solvers/preconditioners:

- BDDC

- ”Soon”: Own AMG (→ Bernd Schwarzenbacher, tomorrow)

Interfaces for:

- MUMPS
- HYPRE BoomerAMG
- HYPRE AMS

Switch on with -DUSE_MUMPS=ON / -DUSE_HYPRE=ON; superbuild downloads and builds the libraries automatically.
Own solvers/preconditioners:

- BDDC
- "Soon": Own AMG (→ Bernd Schwarzenbacher, tomorrow)

Interfaces for:

- MUMPS
- HYPRE BoomerAMG
- HYPRE AMS

Switch on with -DUSE_MUMPS=ON / -DUSE_HYPRE=ON; superbuild downloads and builds the libraries automatically.

Does definitely NOT work (yet): multigrid, Block-jacobi, Block-GS
Experiences
Experiences

Works well on 2000 cores across 100 nodes.

- 3d-poisson, p2, 450M NDOF, 40 sec. setup + 13 sec solve (CG+AMG)

Works on 10 xeon phi’s (640 mpi-procs).

- 3d-poisson, p1, 36M NDOF, 28 sec. setup + 6.5 sec solve (CG+BoomerAMG)
Bonus slide; VTK + Paraview