Adaptive Refinement in Hierarchical Hybrid Grids

Tobias Gradl
Lehrstuhl für Systemsimulation
Universität Erlangen-Nürnberg

2008-02-11
The HHG Framework

Publications

Projects

Adaptive Refinement
HHG principle

Solve finite element problems with multigrid techniques.

Coarse input mesh, refine in a structured way.

Same stencil for all points within a patch
Advantages

- Multigrid is straightforward
- Very memory efficient: $10^{11}$ unknowns are possible
- Very fast
HHG properties

Advantages

▶ Multigrid is straightforward
▶ Very memory efficient: $10^{11}$ unknowns are possible
▶ Very fast

Limitation

▶ Coarse input mesh is needed
Mesh is split up at coarsest level

→ volumes, faces, edges, vertices

Facilitates parallelization for distributed memory parallel computers.
Publications

Multigrid cycle optimization

Project in cooperation with Alexander Thekale

Full multigrid cycle:

Goal: minimize cost.
Constraint: error on finest level $\leq$ constant $\times$ discretization error.
DEISA*: Access to Europe’s largest computers, support with “enabling work”.

Project: Test HHG on a variety of architectures

- Summer 2008
- 100,000 CPU hours
- Combine with application

* Distributed European Infrastructure for Supercomputing Applications
Applications of HHG

- Room acoustics
- Current BMBF* project proposal: earth mantle convection

*Bundesministerium für Bildung und Forschung
Adaptive refinement: motivation

Varying density of unknowns required by

- physics (e.g. singularities)
Adaptive refinement: motivation

Varying density of unknowns required by

- physics (e.g. singularities)
- geometry (e.g. walls of a room)
Two approaches

- Red-green → conforming grids
Two approaches

- Red-green → conforming grids

Two approaches
Refinement with hanging nodes
Two approaches

- Red-green $\rightarrow$ conforming grids

- Hanging nodes $\rightarrow$ non-conforming grids
Two approaches

- Red-green → conforming grids
Two approaches

- Red-green $\rightarrow$ conforming grids
- Hanging nodes $\rightarrow$ non-conforming grids
Comparison: red-green vs. hanging nodes

Both have (dis-)advantages (some of them HHG specific):

- Red-green: impossible for purely quadrilateral/hexahedral grids
- Hanging nodes: too large refined areas
- Red-green: more elements on coarsest level

The ideal HHG grid has few coarse level elements.
⇒ Use refinement with hanging nodes whenever possible.
Mathematical foundation

Unkowns:

\[
\begin{align*}
\left( u^h \right)_i &= \left( I_{h_i}^h u^h \right)_i & \text{for } h < h_i, i = 1..n
\end{align*}
\]
Mathematical foundation

Unkowns:

\[(u^h)_{i} = (l^h_{i} u^{h_i})_{i} \text{ for } h < h_i, \ i = 1..n\]

Residual:

\[A^H u^H = f^H \text{ solved up to discretization error } \Leftrightarrow r^H = 0\]

\[r^h = f^h - A^h u^h = f^h - A^h \left(l^h_H u^H\right)\]

\[r^H = l^H_h r^h \perp 0\]
Refinement with hanging nodes in HHG

\[ Au = f , \quad r = f - Au , \] compact basis functions
Refinement with hanging nodes in HHG

\[ Au = f, \quad r = f - Au, \text{ compact basis functions} \]

Uniform refinement: only one boundary layer
Refinement with hanging nodes in HHG

\[ Au = f, \quad r = f - Au, \] compact basis functions

Uniform refinement: only one boundary layer

Adaptive refinement: two boundary layers
Refinement with hanging nodes in HHG

\[ Au = f, \quad r = f - Au, \quad \text{compact basis functions} \]

Uniform refinement: only one boundary layer

Adaptive refinement: two boundary layers

1. Smooth \( u \)
Refinement with hanging nodes in HHG

\[ Au = f, \quad r = f - Au, \quad \text{compact basis functions} \]

Uniform refinement: only one boundary layer

Adaptive refinement: two boundary layers

1. Smooth \( u \)
2. Compute & restrict \( r \)
Implementation

Become flexible, but stay fast.

- Preserve structured regions ...or at least...
- Treat unstructured regions efficiently.
- Avoid additional communication ...or at least...
- Preserve communication locality.
Example: edge refinement

```cpp
for (int i=0; i<2; i++) {
    this->vertex(i)->triggerResRefinement ();

    if (this->vertex(i)->memoryArray(unk->getId())->refinementLevels() < levels) {
        this->vertex(i)->refineInterpolateUnk(unk, levels);
    }

    memoryArray(unk->getId())->setupVertexDependency
        (this->vertexReferenceIDMap_[i], i,
         this->vertex(i)->memoryArray(unks[iu]->getId()),
         levels);
}
for (int i=0; i<this->faces_.size(); i++) {
    this->faces_[i]->triggerResRefinement ();
    this->faces_[i]->triggerUnkRefinement ();
}
for (int i=0; i<this->elements_.size(); i++) {
    this->elements_[i]->triggerResRefinement ();
    this->elements_[i]->triggerUnkRefinement ();
}
```
Thank you for your attention!
Any questions?