Parallelism on Adaptive Unstructured Meshes

Roy Stogner
roystgnr@cfdlab.ae.utexas.edu

1Univ. of Texas at Austin

March 31, 2008
Outline

1. Finite Element Computation
2. Parallelism
3. ParallelMesh
4. Parallel Algorithms
   - Parallel Programming
   - Inter-Processor Communication
   - Adaptivity Issues
4.5. Summary
FEM Computational Costs

- Discrete system solves
- Discrete Jacobian assembly
- Discrete residual assembly
- Sparse Jacobian allocation
- I/O
- Mesh generation
- Mesh movement
Adaptive FEM Costs

- Error estimator evaluation
- Adaptive refinement/coarsening
- Inter-mesh projections
- Adaptivity flagging
- Adaptive constraint calculations
## Parallelism Goals

<table>
<thead>
<tr>
<th>Reduced CPU time</th>
<th>Reduced memory requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed CPU usage</td>
<td>Larger attainable problem size</td>
</tr>
<tr>
<td>Asynchronous I/O</td>
<td></td>
</tr>
</tbody>
</table>

- Reduced CPU time
  - Distributed CPU usage
  - Asynchronous I/O

- Reduced memory requirements
  - Larger attainable problem size
Parallelism Levels

- SIMD Instructions
  - Assemblies, MatVec operations

- Shared-memory Threads
  - Mesh Partitioning
Parallelism Levels

Separate Simulations
- Parametric studies
- Uncertainty analysis

Distributed-memory Processes
- Asynchronous I/O
- Mesh Partitioning
Mesh Classes

- **Abstract iterator interface** hides mesh type from most applications
- **UnstructuredMesh** "branch" for most library code
- **ParallelMesh** implements data storage, synchronization

```
MeshBase
+boundary_info: AutoPtr<BoundaryInfo>
+elem(unsigned int)
+node(unsigned int)
+elements_begin/end()
+nodes_begin/end()
+active_local_elements_begin/end()
+read()
+write()

CartesianMesh  UnstructuredMesh
+find_neighbors()
+read/write()  +add/delete_elem()

SerialMesh  ParallelMesh
+_elements: vector<Elem*>
+elements_begin()

SerialMesh  ParallelMesh
+_elements: mapvector<Elem*>
+allgather()
+delete_remote_elements()```
SerialMesh Partitioning

- Each element, node is "local" to one processor
- Each processor has an identical Mesh copy
- Mesh stays in sync through redundant work
- FEM data synced on "ghost" elements only
ParallelMesh Partitioning

- Processors store only local and ghost objects
- Each processor has a small Mesh subset
- Mesh stays in sync through MPI communication
ParallelMesh Partitioning

Pros
- Reduced memory use
- $O(N_E/N_P)$ CPU costs
ParallelMesh Partitioning

Cons

- Increased code complexity
- Increased synchronization “bookkeeping”
- Greater debugging difficulty
Gradual Parallelization

- New internal data structure
- Methods to delete, reconstruct non-semilocal objects
- Parallelized DofMap methods
- Parallelized MeshRefinement methods
- Parallel Mesh I/O support
- Load balancing support

Also working on parallel support in Boundary, Function, Generation, Modification, Generation, Tools classes
Parallel:: API

Encapsulating MPI

- Improvement over MPI C++ interface
- Makes code shorter, more legible

Example:

```cpp
std::vector<Real> send, recv;
...
send_receive(dest_processor_id, send,
            source_processor_id, recv);
```
Instead of:

```c
if (dest_processor_id == libMesh::processor_id() &&
    source_processor_id == libMesh::processor_id())
    recv = send;
#endif // HAVE_MPI
else
{
    unsigned int sendsize = send.size(), recvsize;
    MPI_Status status;
    MPI_Ssendrecv(&sendsize, 1, datatype<unsigned int>(),
                  dest_processor_id, 0,
                  &recvsize, 1, datatype<unsigned int>(),
                  source_processor_id, 0,
                  libMesh::COMM_WORLD,
                  &status);

    recv.resize(recvsize);

    MPI_Ssendrecv(sendsize ? &send[0] : NULL, sendsize, MPI_DOUBLE,
                   dest_processor_id, 0,
                   recvsize ? &recv[0] : NULL, recvsize, MPI_DOUBLE,
                   source_processor_id, 0,
                   libMesh::COMM_WORLD,
                   &status);
}
#endif // HAVE_MPI
```
## ParallelMesh Data Structure

### std::vector fails
- Not sparse
- $O(N_E)$ storage cost

### std::map
- "mapvector" interface provides iterators
- $O(\log(N_E/N_P))$ lookup time without std::hash_map
- $O(1)$ lookup time with std::hash_map

### Hybrid data structure?
- Dense vector for most elements
- Sparse data structure for new elements
Round Robin Communications

- Processor $P$ sends to processor $P + K$ while receiving from $P - K$
- New data is operated on and old data discarded
- $K$ is incremented “round robin” from 1 to $N_P - 1$
Round Robin Communications

Pros
- $O(N_G/N_P)$ memory usage - only one data exchange at a time
- Straightforward to code
- Reliable
Inter-Processor Communication

Round Robin Communications

**Cons**

- Communications loop over non-neighboring processors
- $O(N_G)$ execution time
- Multiple, synchronous communications
Adaptivity Issues

Adaptivity and ParallelMesh

Challenges

- Reindexing elements, nodes, DoFs
- Synchronization of ghost objects
- Load balancing, Repartitioning
Parallel Global Indexing

One-pass indexing

- Index processor $P$ from $\sum_{1}^{P-1} N_{Ep}$
- Pass $\sum_{1}^{P} N_{Ep}$ to processor $P + 1$
- $O(N_E)$ work
- $O(N_E)$ execution time
Adaptivity Issues

Parallel Global Indexing

Two-pass indexing

- Count processor $P$ indices from 0
- Gather $N_{Ep}$ on all processors
- Re-index processor $P$ from $\sum_{1}^{P-1} N_{Ep}$
- Double the work
- $O(N_{E}/N_{P})$ execution time
Parallel Synchronization

What can lose sync?

- Refinement flags
- New child elements, nodes
- New degrees of freedom
- Hanging node constraint equations
- Repartitioned elements, nodes
Adaptivity Issues

Parallel Synchronization

Round-Robin Complications

- Refinement flags must obey consistency rules
- New ghost nodes may have unknown processor ids
- Constraint equations may be recursive
- Hanging node constraint equations
- Repartitioned elements, nodes
Debugging

- Regression tests
- Precondition, postcondition tests
- Unit testing
- Parallel debuggers
- Low $N_P$ test cases
Summary

Parallelism tradeoffs

- Efficiency vs. ease of programming/debugging
- Latency vs. redundant work
- "Premature optimization" mistakes vs. bad assumptions

and guidelines

- Reuse existing code/algorithms
- Build incrementally
- Test extensively