NALU Assembly – Prototyping the NGP Transition

Christian Trott, Anthony Agelastos, Victor Brunini, Stefan Domino, Travis Fisher, Simon Hammond, Paul Lin
Where Sandia’s Codes Are Now

- Largely C++ Code Base
  - True C++: Object Oriented, Inheritance, Templates, Virtual Functions
- Some Fortran Still Around
- MPI-Only
  - Effectively Zero Threading in Production
- Solvers are mostly application specific
- Trilinos developed new algorithms, but little production use
- Little threading experience in code groups
  - Need to build expertise over time
The Transition To NGP

- Mandate to Run Applications on Trinity (Xeon Phi) and Sierra (IBM Power8+GPU)
- Need portable codes with good (enough) performance
- Both machines benefit from threading, for Sierra it is required
- Candidate Models: OpenMP and Sandia’s Kokkos
  - OpenACC is effectively single-vendor model
  - OpenCL does not support C++ in kernels, most vendors seem to have it on low priority, and it’s the most complicated syntax from all models
- Decision was made to go with Kokkos
  - OpenMP not proven on GPUs
  - OpenMP does not currently handle memory hierarchies well (e.g. there is no support for KNLs two level memory system)
  - OpenMP 4 does not behave well in a class based environment when targeting GPUs
Timetable for Transition

- **Trinity Phase I**:
  - Serial
  - OpenMP
  - Kokkos Min. CPU X

- **Trinity Phase II**:
  - Optimized OpenMP
  - Kokkos Min. CPU

- **Sierra**:
  - Kokkos Min. GPU
  - K-Opt 1
  - K-Opt 2
  - K-Opt 3

- **Crossroads**:
  - RAJA Optimized CPU
  - RAJA GPU/Optimized GPU

- **Timeline**:
  - 2015-2021:
    - OpenMP
    - CUDA
    - C++20 Language Specification
NALU: a CFD Application as Prototype

- www.github.com/spdomin/Nalu
- Patterns, and data structures shared with IC production apps
- Unstructured meshes
- Uses Trilinos Solver stack
- Scaling demonstrated to >500k cores

2D/3D Sliding/Overset Mesh

- www.github.com/crtrott/Nalu
- Prototype Kokkos assembly
- Used to harden threaded solver stack
- Get Nalu ready for Trinity Phase II demonstration

MultiPhysics CHT
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++ ) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++ ) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for (ib = 0; ib < num_buckets; ib++) {
  Bucket b = buckets[ib];

  std::vector<double> local_data(b.data_size());
  b.gather_data(local_data);

  for (elem = 0; elem < b.num_elems(); elem++) {
    std::vector<double> elem_contribute(b.topo_size());
    b.material_compute(elem, elem_contribute, local_data);
    matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
  }
}
```

Elements are grouped in buckets
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```c++
for( ib = 0; ib < num_buckets; ib++ ) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num elems(); elem++ ) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```
The Assemble Algorithm

- **High level algorithm:**

```cpp
for( ib = 0; ib < num_buckets; ib++ ) {
    Bucket b = buckets[ib];
    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);
    for ( elem = 0; elem < b.num_elems(); elem++ ) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++ ) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++ ) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for (ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for (elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```

Compute matrix contributions
The Assemble Algorithm

- High level algorithm:

```cpp
define
for (ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for (elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```
The Assemble Algorithm

- High level algorithm:

```cpp
for (ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for (elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```

Add contributions into matrix
The Assemble Algorithm

- High level algorithm:

```cpp
for( ib = 0; ib < num_buckets; ib++) {
    Bucket b = buckets[ib];

    std::vector<double> local_data(b.data_size());
    b.gather_data(local_data);

    for ( elem = 0; elem < b.num_elems(); elem++) {
        std::vector<double> elem_contribute(b.topo_size());
        b.material_compute(elem, elem_contribute, local_data);
        matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
    }
}
```
Potential threading issues

- Allocates temporary arrays (std::vector)
  - *Use pre allocated scratch space*
- Limited parallelism on each level
  - Typical number of elems per bucket 512
  - Typical number of buckets per node 512
  - *Use hierarchical parallelism to expose both levels*
- Potential write conflicts for matrix sum_into
  - *Use Atomic accumulation*
- Mesh data too large to fit into high bandwidth memory
  - *Use high bandwidth memory for scratch space only*
- Divergence due to different material models
  - *Use uniform buckets (i.e. elements within a bucket are same type)*
  - This is the legacy approach anyway
**The Kokkos Version**

*Add Hierarchical Parallelism: One thread team per bucket*

```cpp
TeamPolicy<> policy(b.num_buckets, AUTO());

parallel_for(policy, [=] (TeamPolicy<>::member_type& team) {
    ib = team.league_rank();

    Bucket b = buckets[ib];
    std::vector<double> local_data(b.data_size());

    b.gather_data(local_data);

    parallel_for(TeamThreadRange(team, 0, b.num_elems()),
                 [&] (const int& elem) {

            std::vector<double> elem_contribute(b.topo_size());

            b.material_compute(elem, elem_contribute, local_data);

            matrix_A.sum_into(b.contrib_list(elem), elem_contribute);
        });
```
The Kokkos Version

*Use scratch space for temporary allocations*

```cpp
TeamPolicy<> policy = TeamPolicy<>((b.num_buckets,AUTO())).
  set_scratch_size(2,b.team_scr(),b.thread_scr());

parallel_for(policy, [=] (TeamPolicy<>::member_type& team) {
  ib = team.league_rank();

  Bucket b = buckets[ib];
  View<double*> local_data(team.team_scratch(2),b.data_size());
  View<double*> elem_contribute(team.thread_scratch(2),b.data_size());

  b.gather_data(local_data);

  parallel_for(TeamThreadRange(team, 0, b.num elems()),
               [&] (const int& elem) {

    b.material_compute(elem, elem_contribute, local_data);
    matrix_A.sum_into(b.contrib_list(elem),elem_contribute);
  });
});
```
Assemble Nalu-kokkos
simple heat conduction; 262,144 elements;
run on Shepard (Haswell, 16C/32T per socket)
Next Plans

- Port more assembly kernels
  - Support Open Science campaign with Nalu on Trinity

- Explore common interface layer for bucket loops
  - Potentially put something in a library

- Discuss algorithm with other IC teams
  - Do we need determinism (i.e. avoid atomics?)
  - Is the hierarchical parallelism layer acceptable?
  - Is the a-priori scratch size determination problematic?

- Move data structures to enable GPU execution
  - Need to switch from std::vector in the bucket data structure

www.github.com/spdomin/Nalu -- Master Version of Nalu
www.github.com/crtrott/Nalu -- Kokkos Version of Nalu
www.github.com/kokkos/kokkos -- Kokkos repository