The Fast Multipole Method in Pictures

Brad Chamberlain, Cray Inc.
A number of point sources on the surface of an object
Finest octree-style discretization of space
The non-empty finest cells of interest
This will be the cell we focus on in this example
(as well as its parents)
= “outer” signature function we need to calculate
(field induced outside cell due to sources inside cell)
compute outer signature function directly from sources in cell
(repeat for all cells)
we’re done with the point sources for now
interpolate the outer signature functions for use at the next level
jump to next level of hierarchy
need to compute outer signature functions for these cells
do this using an outer-to-outer translation, combining child values to get parent’s
repeat for all cells
(done with finest level for now)
Interpolate signature functions for use at next level
go to next level of hierarchy (the coarsest for this example)
need to compute these outer signature functions
again, do so using children
repeat for all cells
ignore middle level for now; next we’ll start to compute inner signature functions (field inside cube due to stuff outside)
compute inner signature functions using outer-to-inner translation

= these cells are too close for the transform to be valid (adjacent)

= a contribution from one cell to another (non-neighboring)
repeat for all cells
return to middle level; now we want to compute inner signature functions for these cells
propagate parent inner values to children using inner-to-inner translation for starters…
repeat for all children…
filter resulting signature functions to be appropriate at this level
then use outer-to-inner translation to get contributions from cells whose parents were too close (but are not too close themselves)
repeat for all cubes at this level
we’re done with the coarse level
now repeat for the finest level
use inner-to-inner translation to propagate from parents to children
repeat for all children
filter inner signature functions to be appropriate at finest level
use outer-to-inner translation to take care of cells whose parents were previously too close

This is effectively a 216- (or perhaps 190)-point stencil: $6 \times 6 \times 6 (-3 \times 3 \times 3 + 1)$
repeat for all cells
now we’re done with the FMM
we have computed the inner expansion for all cells at the finest level
Compute the far-field at the source points in each cell at the finest level
FMM Data Structures

\[ \text{var OSgfn, ISgfn: [lvl in Levels] [SpsCubes[lvl]] [Sgfns[lvl]] [1..3] complex;} \]

1D array over levels of the hierarchy
FMM Data Structures

```chapel
var OSgfn, ISgfn: [lvl in Levels] [SpsCubes[lvl]] [Sgfns[lvl]] [1..3] complex;
```

1D array over levels of the hierarchy

...of 3D sparse arrays of cubes (per level)

...of 2D discretizations of spherical functions, (sized by level)

...of complex values

...of 1D vectors
FMM Parallelism

Note that here, we want to parallelize over spatial arrays.

Whereas here, we want to parallelize over the signature functions.
FMM Distributions

Distributing these is harder (recursive bisection technique?)

Distributing these is easy (Block works well)
FMM: Supporting Declarations

\[
\text{var} \ OSgfn, \ ISgfn: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ [\text{SpsCubes(lvl)}] \ [\text{Sgfns(lvl)}] \ [1..3] \ \text{complex};
\]

previous definitions:

\[
\begin{align*}
\text{var} & \ n: \ \text{int} = \ldots; \\
\text{var} & \ \text{numLevels}: \ \text{int} = \ldots; \\
\text{var} & \ \text{Levels}: \ \text{domain}(1) = [1..\text{numLevels}]; \\
\text{var} & \ \text{scale}: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ \text{int} = 2^{\text{lvl}-1}; \\
\text{var} & \ \text{SgFnSize}: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ \text{int} = \text{computeSgFnSize(lvl)}; \\
\text{var} & \ \text{LevelBox}: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ \text{domain}(3) = [(1,1,1)..(n,n,n)] \ \text{by} \ \text{scale(lvl)}; \\
\text{var} & \ \text{SpsCubes}: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ \text{sparse subdomain}(	ext{LevelBox}) = \ldots; \\
\text{var} & \ \text{Sgfns}: \ [\text{lvl} \ \text{in} \ \text{Levels}] \ \text{domain}(2) = [1..\text{SgFnSize(lvl)}, 1..2*\text{SgFnSize(lvl)}];
\end{align*}
\]
FMM: Computation

`var OSGfn, ISgfn: [lvl in Levels] [SpsCubes(lvl)] [Sgfns(lvl)] [1..3] complex;`

**outer-to-inner translation:**

```chapel
for lvl in 1..numLevels-1 by -1 {
    ...
    forall cube in SpsCubes(lvl) {
        forall sib in out2inSiblings(lvl, cube) {
            const Trans = lookupXlateTab(cube, sib);

            atomic ISgfn(lvl)(cube) += OSGfn(lvl)(sib) * Trans;
        }
    }
    ...
}
```
Fast Multipole Method: Summary

- Chapel code captures structure of data and computation far better than sequential Fortran/C versions (to say nothing of the MPI versions)
  - cleaner, more succinct, more informative
  - rich domain/array support plays a big role in this

- Parallelism shifts at different levels of hierarchy
  - Aided by global-view programming and nested parallelism

- Boeing FMM expert was able to find bugs in my implementation when seeing Chapel for the first time

- Yet, I’ve elided some non-trivial code (the distributions)