SDAV
Scalable Data Management, Analysis, and Visualization

U.S. DEPARTMENT OF ENERGY | Office of Science

PORTABLE PARALLEL HALO AND CENTER FINDERS FOR HACC

Christopher Sewell, LANL
Katrin Heitmann, ANL
Ollie Lo, LANL
Salman Habib, ANL
Jim Ahrens, LANL

OLCF User Meeting
July 23, 2014

LA-UR-14-25437

Outline

Background: SDAV, PISTON, VTK-m, data-parallel abstractions, and halo analysis

Algorithms: Halo and center finding

Results: Moonlight (LANL), Stampede (TACC), and Titan (ORNL)

Extensions: in-situ integration and Poisson center finder

Future needs: current bottlenecks, expected trends, possible hardware and software solutions

Background

SDAV, PISTON, VTK-m, Data-Parallel Abstractions, and Halo Analysis

LA-UR-14-25437

SDAV VTK-m Frameworks

Objective: Enhance existing multi/many-core technologies in anticipation of in situ analysis use cases with LCF codes

Benefit to scientists: These frameworks will make it easier for domain scientists' simulation codes to take advantage of the parallelism available on a wide range of current and next-generation hardware architectures, especially with regards to visualization and analysis tasks

Projects

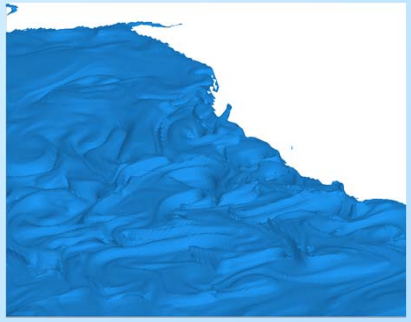
- EAVL, Oak Ridge National Laboratory
- Dax, Sandia National Laboratory
- DIY, Argonne National Laboratory
- PISTON, Los Alamos National Laboratory

Work on integrating these projects with VTK is on-going, in collaboration with Kitware

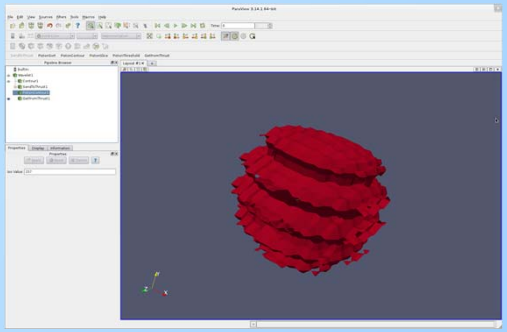
Algorithms for Science Applications Using VTK-m

- The PISTON component of VTK-m focuses on developing data-parallel algorithms that are portable across multi-core and many-core architectures for use by LCF codes of interest
- PISTON consists of a library of visualization and analysis algorithms implemented using Thrust, as well as a set of extensions to Thrust
- PISTON algorithms are integrated into LCF codes in-situ either directly or though integration with ParaView Catalyst

PISTON isosurface with curvilinear coordinates



Ocean temperature isosurface generated across four GPUs using distributed PISTON



PISTON integration with VTK and ParaView

SDAV
Scalable Data Management, Analysis, and Visualization

LA-UR-14-25437

U.S. DEPARTMENT OF ENERGY | Office of Science

Brief Introduction to Data-Parallelism and Thrust

What algorithms does Thrust provide?

- Sorts
- Transforms
- Reductions
- Scans
- Binary searches
- Stream compactions
- Scatters / gathers

Challenge: Write operators in terms of these primitives only

Reward: Efficient, portable code

```
input                      4  5  2  1  3
-----------------------------------------
transform(+1)              5  6  3  2  4
inclusive_scan(+)          4  9 11 12 15
exclusive_scan(+)          0  4  9 11 12
exclusive_scan(max)        0  4  5  5  5
transform_inscan(*2,+)     8 18 22 24 30
for_each(-1)               3  4  1  0  2
sort                       1  2  3  4  5
copy_if(n % 2 == 1)        5  1  3
reduce(+)                                15

input1                     0  0  2  4  8
input2                     3  4  1  0  2
-----------------------------------------
upper_bound                3  4  2  2  3
permutation_iterator       4  8  0  0  2
```

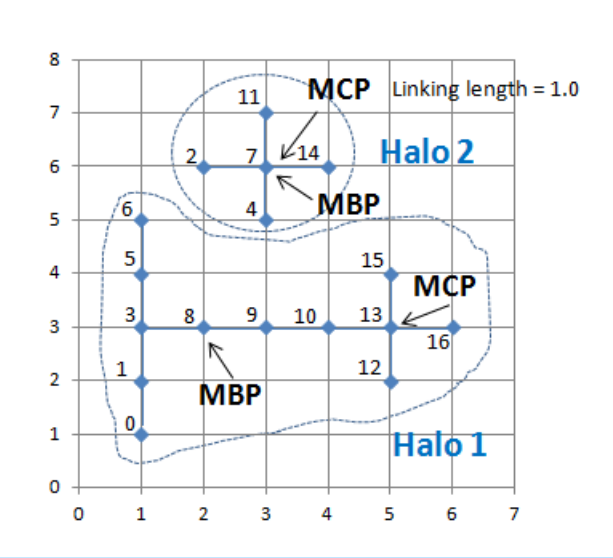
Halo Analysis

- An important and time-consuming analysis function within HACC is finding halos and the centers of those halos
- HACC is written as an MPI+X code, designed to be easy to port and optimize for different architectures
- It would be very time-consuming to optimize the full cross-product of all analysis operators and all architectures
- Data-parallelism, using a framework such as PISTON, allows a single implementation of an analysis operator to take advantage of all supported architectures
- Total implementation work is O((# of analysis operators) **+** (# of architectures)) rather than O((# of analysis operators) ***** (# of architectures))

Images of the matter density field, produced by the HACC simulation on Titan
Image credits: Joe Insley and Silvio Rizzi

Definitions

- Friend-of-friends (FOF) halo: connect each particle to all “friends”, i.e., all other particles within a specified “linking length” of it; two particles will end up in the same “halo” if there exists any chain of “friends” between them
- Most connected particle (MCP) center: the particle within a halo with the most “friends”
- Most bound particle (MBP) center: the particle within a halo with the lowest potential, where the potential for a given particle is computed as the sum over all other particles of the negative of mass divided by distance



MBP center:

$$\arg\min_{i \in H} \left(- \sum_{j \in H,\, j \neq i} \frac{m_j}{d_{ij}} \right)$$

Algorithms

Halo and Center Finding

LA-UR-14-25437

Distributed Parallel Halo Finder

- Particles are distributed among processors according to a decomposition of the physical space
- Overload zones (where particles are assigned to two processors) are defined such that every halo will be fully contained within at least one processor
- Each processor finds halos within its domain
- At the end, the parallel halo finder performs a merge step to handle “mixed” halos (shared between two processors), such that a unique set of halos is reported globally
- Developed primarily by Pat Fasel at LANL, with the serial halo finder running on each individual processor using a KD-tree based algorithm designed by C.H. Hsu

Distributed Parallel Halo Finder

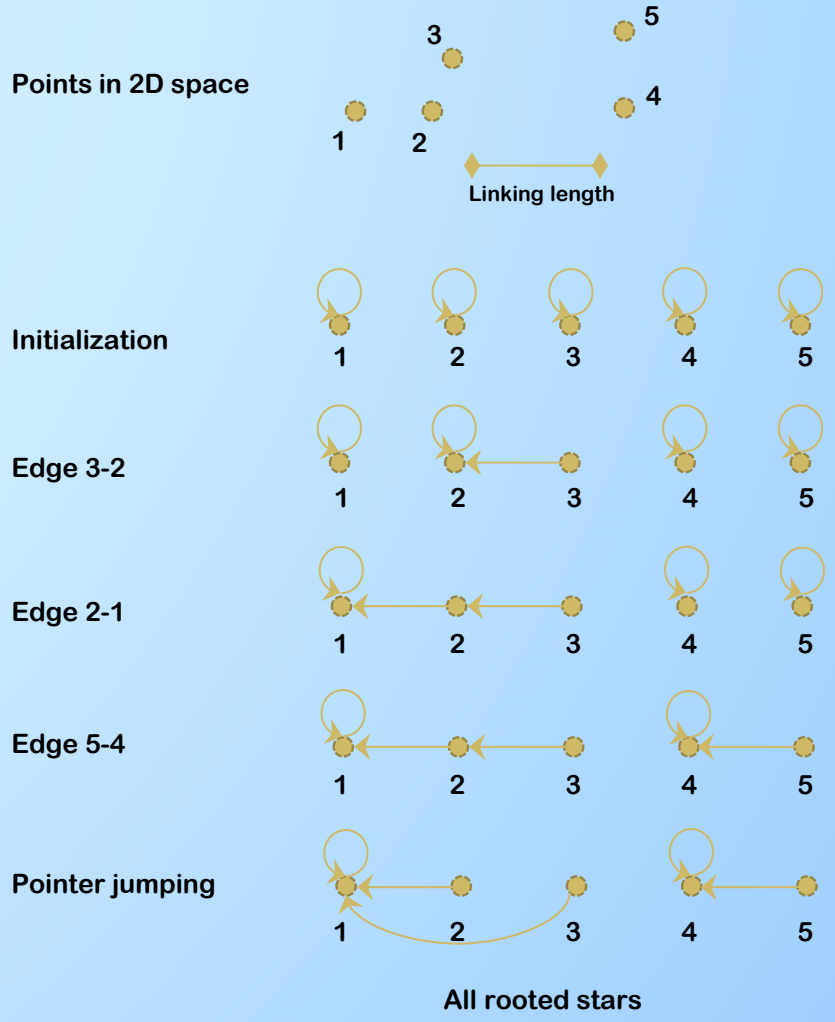
- Particles are distributed among processors according to a decomposition of the physical space
- Overload zones (where particles are assigned to two processors) are defined such that every halo will be fully contained within at least one processor
- Each processor finds halos within its domain: Drop in PISTON multi-/many-core accelerated algorithms
- At the end, the parallel halo finder performs a merge step to handle “mixed” halos (shared between two processors), such that a unique set of halos is reported globally

Halo Finder Algorithm: Connected Components

```
for all vertices i D(i):=i
while (true) {
  // Graft trees onto smaller
  // vertices of other trees
  for all (i,j)∈E pardo
    if (D(i)=D(D(i)) and D(j)<D(i))
      set D(D(i)):=D(j)

  // If all the vertices are in
  // rooted stars, then exit
  for all vertices i pardo
    set star(i):=true
  for all vertices i pardo
    if (D(i)≠D(D(i))
      set star(i),star(D(i)),
          star(D(D(i))):=false
  for all vertices i pardo
    set star(i):=star(D(i))
  if (star(i) for all i) break

  // Pointer jumping on each vertex
  for all i pardo set D(i):=D(D(i))
}
```

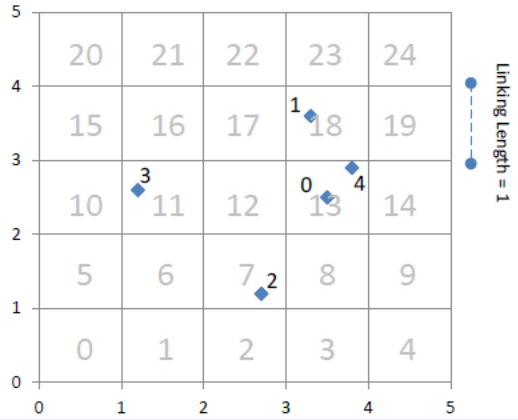


Reference: *An Introduction to Parallel Algorithms*, Joseph JáJá, 1992

Halo Finder Algorithm: Computing Edges on the Fly

- If we define an edge to exist between two particles if and only if their distance is less than the linking length, the connected components solution is the FOF halos
- However, it could take $O(n^2)$ time and $O(n^2)$ memory to directly compute and store all edges
- Instead, partition the domain into bins with edge length equal to the linking length
  - Friends can only exist in its own bin or one of 27 neighbor bins
  - Compute edges on the fly in the algorithm, comparing each particle to each other particle in its bin and its neighbor bins to see which edges exist
  - The number of bins (most empty) may be too large to store pointer to each, so instead store only pointers to neighbor bins for each particle
  - Neighbor bins will be located in 1D vector in groups of three, so 9 not 27 pointers needed to neighbors for each particle

Halo Finding: Binning Example



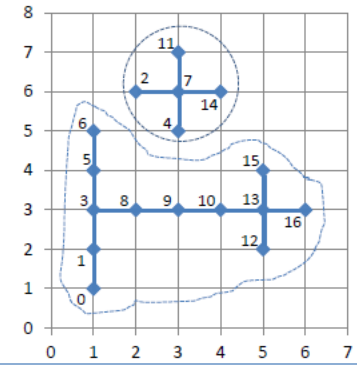

```
I          0        1        2        3        4
X        3.5      3.3      2.7      1.2      3.8
Y        2.5      3.6      1.2      2.6      2.9
for_each(cnt_itr(0), cnt_itr(0)+n, compute_bins(X, Y, B))
B         13       18        7       11       13
sort_by_key(B.begin(), B.end(), zip(X, Y, I))
B          7       11       13       13       18
I          2        3        0        4        1
for_each(cnt_itr(0), cnt_itr(0)+n, compute_neighbor_range_start(B, R))
R    1   6 11   5 10 15   7 12 17   7 12 17 12 17 22
lower_bound(B.begin(), B.end(), R.begin(), R.end(), N1.begin())
N1   0   0   1   0   1   4   0   2   4   0   2   4   2   4   5
for_each(cnt_itr(0), cnt_itr(0)+n, compute_neighbor_range_end(R))
R    3   8 13   7 12 17   9 14 19   9 14 19 14 19 24
upper_bound(B.begin(), B.end(), R.begin(), R.end(), N2.begin())
N2   0   1   4   1   2   4   1   4   5   1   4   5   4   5   5
```

Two vectors are computed, N1 and N2, which contain the beginning and ending of three ranges (nine in 3D) in the sorted particle vectors for which each particle will need to search for its potential friends.

Center Finding Algorithms

- Most connected particle can be found easily by counting the number of friends for each particle during any iteration through the virtual edge list, and then using a max-reduce to find the maximum
- Similarly, an approximation of DB scan can exclude any particle with too few friends from all edges
- Most bound particle can be found by computing the potential for each particle in a highly parallel brute force approach
- Centers for all halos can be computed simultaneously using segmented vectors

MBP Center Finding Example




```
I       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
X       1   1   2   1   3   1   1   3   2   3   4   3   5   5   4   5   6
Y       1   2   6   3   5   4   5   6   3   3   3   7   2   3   6   4   3
D       0   0   2   0   2   0   0   2   0   0   0   2   0   0   2   0   0
sort_by_key(D.begin(), D.end(), zip(I, X, Y)
D       0   0   0   0   0   0   0   0   0   0   0   0   2   2   2   2   2
I       0   1   3   5   6   8   9  10  12  13  15  16   2   4   7  11  14
inclusive_scan_by_key(D.begin(), D.end(), cnt_itr(0), H1.begin(), min)
H1      0   0   0   0   0   0   0   0   0   0   0   0  12  12  12  12  12
sequence(H2.begin(), H2.end(), 0)
H2      0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
inclusive_scan_by_key(D.rbegin(), D.rend(), H2.rbegin(), H2.rbegin(), max)
H2     11  11  11  11  11  11  11  11  11  11  11  11  16  16  16  16  16
for_each(cnt_itr(0), cnt_itr(0)+n, compute_potential(X, Y, H1, H2, P))
P    -4.01 -5.22 -5.77 -5.22 -4.01 -6.02 -5.83 -5.93 -4.84 -6.01 -4.83 -4.46 -2.91 -2.91 -4.00 -2.91 -2.91
inclusive_scan_by_key(D.begin(), D.end(), P.begin(), Pmin.begin(), min)
Pmin -4.01 -5.22 -5.77 -5.77 -5.77 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -2.91 -2.91 -4.00 -4.00 -4.00
inclusive_scan_by_key(D.rbegin(), D.rend(), Pmin.rbegin(), Pmin.rbegin(), min)
Pmin -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -6.02 -4.00 -4.00 -4.00 -4.00 -4.00
transform(cnt_itr(0), cnt_itr(0)+n, C.begin(), equals_min_pot(I, P, Pmin)
C       0   0   0   0   0   8   0   0   0   0   0   0   0   0   7   0   0
inclusive_scan_by_key(D.begin(), D.end(), C.begin(), C.begin(), max)
C       0   0   0   0   0   8   8   8   8   8   8   8   0   0   7   7   7
inclusive_scan_by_key(D.rbegin(), D.rend(), C.rbegin(), C.rbegin(), max)
C       8   8   8   8   8   8   8   8   8   8   8   8   7   7   7   7   7
sort_by_key(I.begin(), I.end(), zip(D, C))
I       0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
C       8   8   7   8   7   8   8   7   8   8   8   7   8   8   7   8   8
```

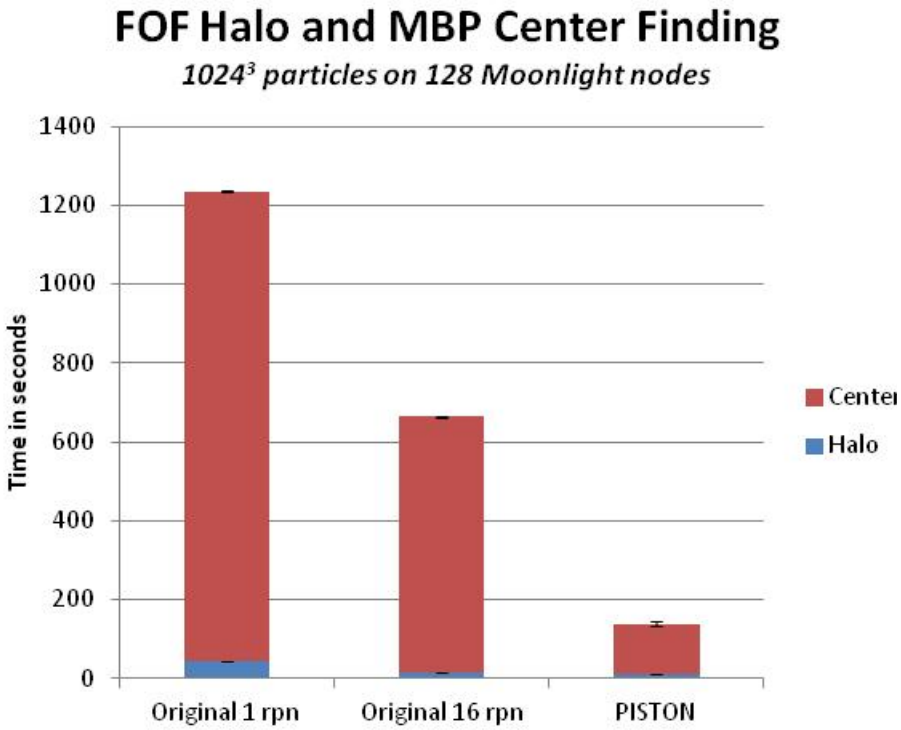
The inputs are the particle ids (I), coordinates (X, Y), and halo id (D, found using the halo finding algorithm). The output is a vector C containing for each particle the id of the MBP center for its halo.

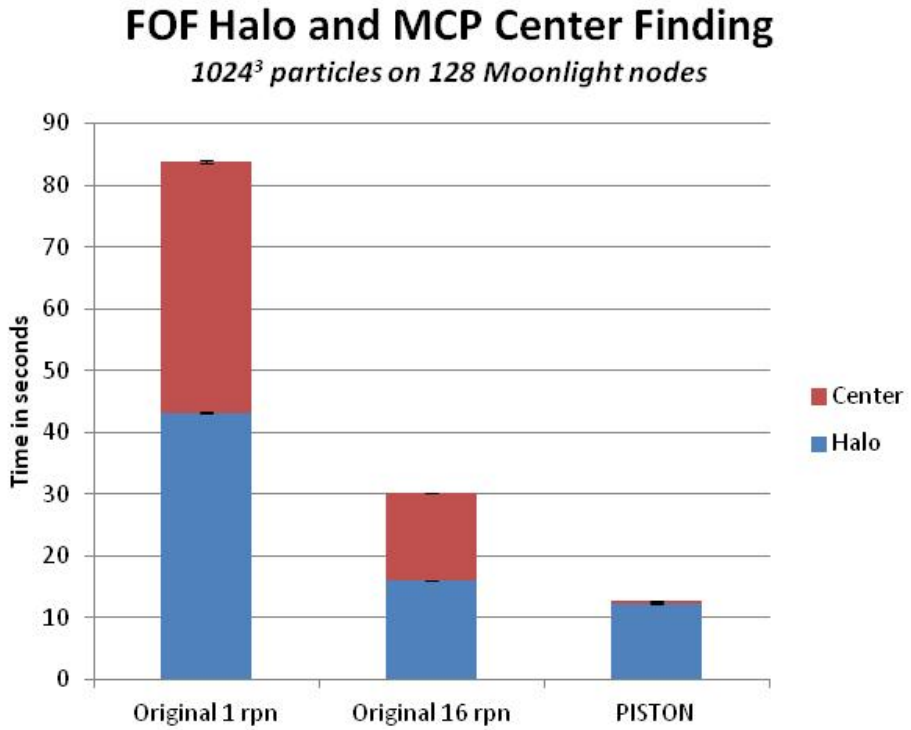
Results

Moonlight, Stampede, and Titan

LA-UR-14-25437

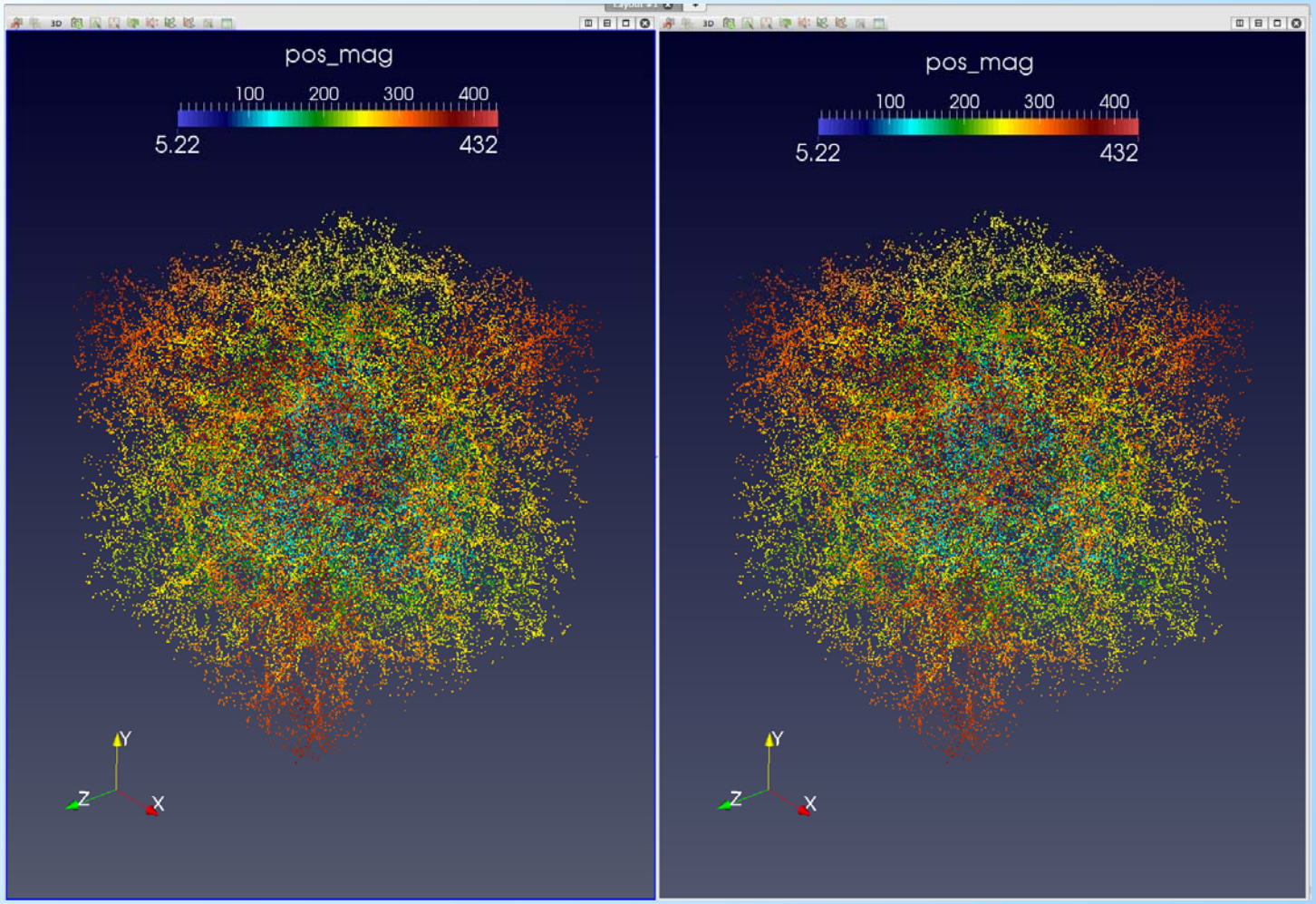
Results: Moonlight





FOF + MBP: PISTON ~4.9x faster than original with 16 rpn
FOF + MCP: PISTON ~2.5x faster than original with 16 rpn

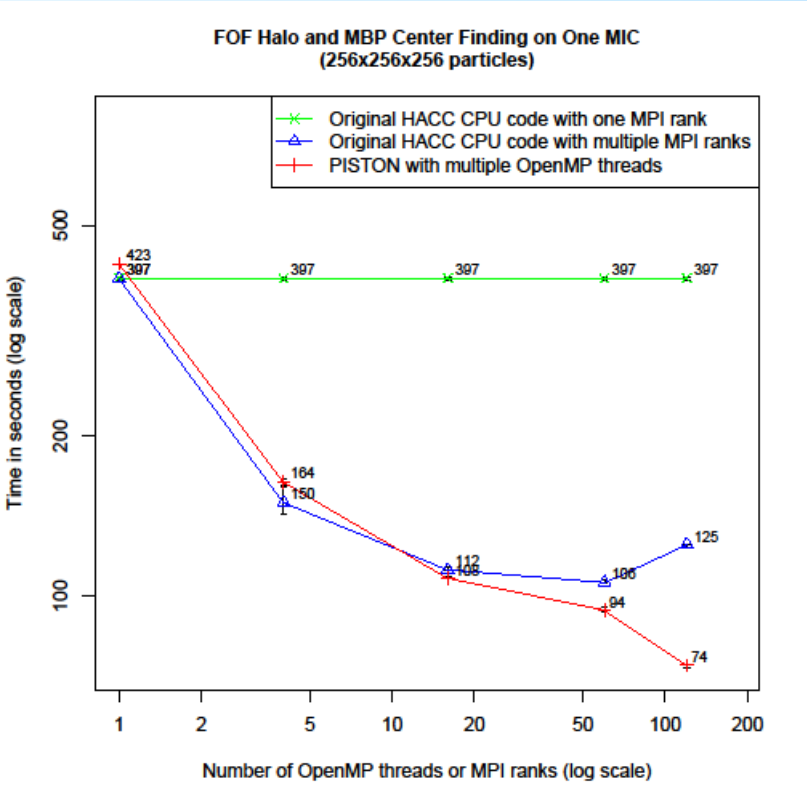
Results: Visual comparison of halos



Original Algorithm **VTK-m Algorithm**

LA-UR-14-25437

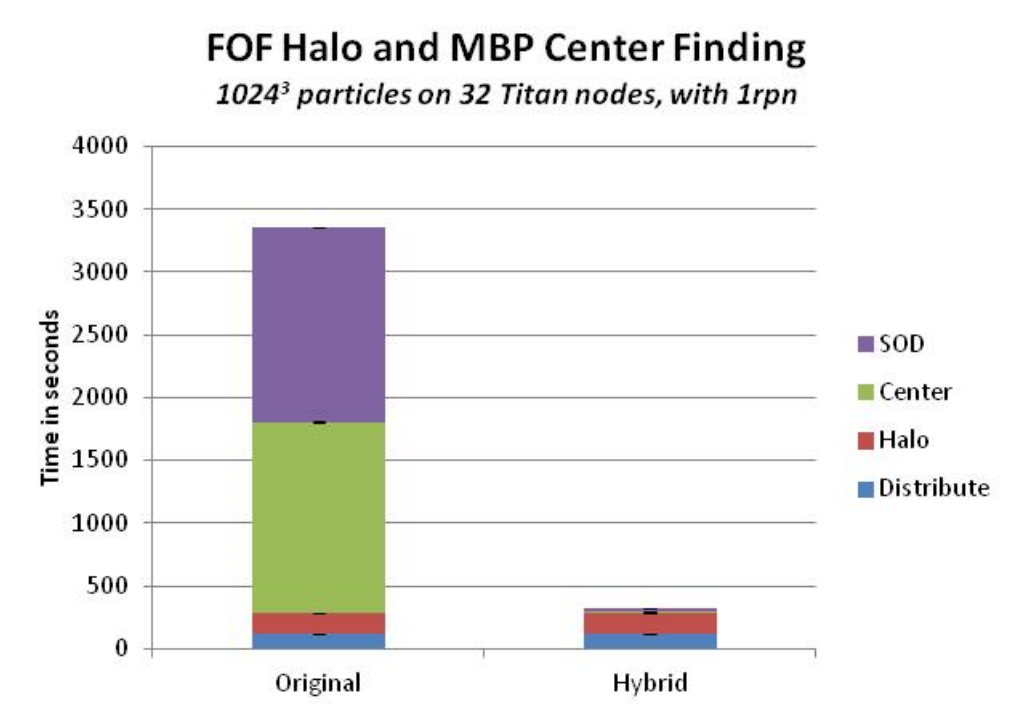
Results: Xeon Phi (MIC) on Stampede



- *To demonstrate the portability of our algorithms, the same code was compiled to the Thrust OpenMP backend (including our own OpenMP implementation of scan) and run on a 256^3 particle data set on an Intel Xeon Phi SE10P (MIC) Coprocessor on a single node of Stampede at TACC*
- *PISTON version scales to more cores than running the existing serial algorithms with multiple MPI processes*

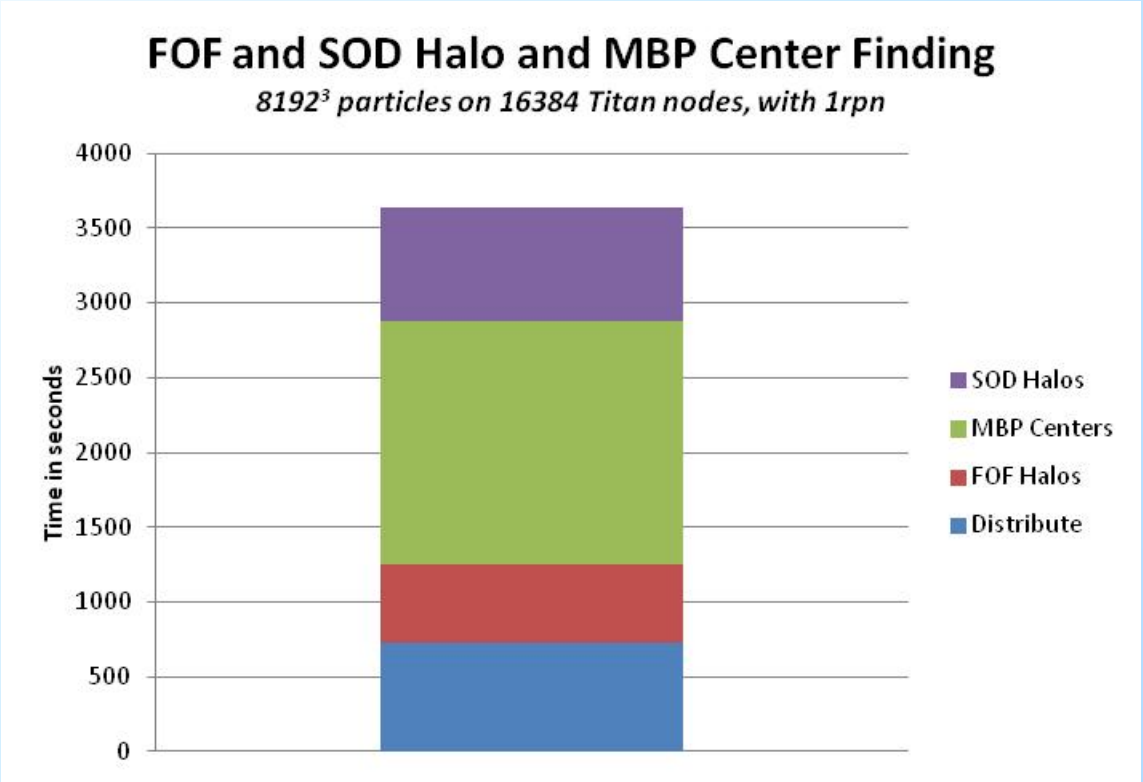
LA-UR-14-25437

Results: Titan



- *This test problem has ~90 million particles per process.*
- *Due to memory constraints on the GPUs, we utilize a hybrid approach, in which the halos are computed on the CPU but the centers on the GPU.*
- *The PISTON MBP center finding algorithm requires much less memory than the halo finding algorithm but provides the large majority of the speed-up, since MBP center finding takes much longer than FOF halo finding with the original CPU code.*

Results: Large Run on Titan



- *Because the memory requirements increase with the number of MPI processes due to overload regions, the HACC simulation with this data can only use one MPI process per node, along with the associated GPU, given the memory available on Titan.*
- *Thus, there is an even greater potential for speed-up by utilizing the GPUs.*
- *The performance improvements using PISTON on GPUs allowed halo analysis to be performed on a very large 8192^3 particle data set across 16,384 nodes on Titan for which analysis using the existing CPU algorithms was not feasible.*

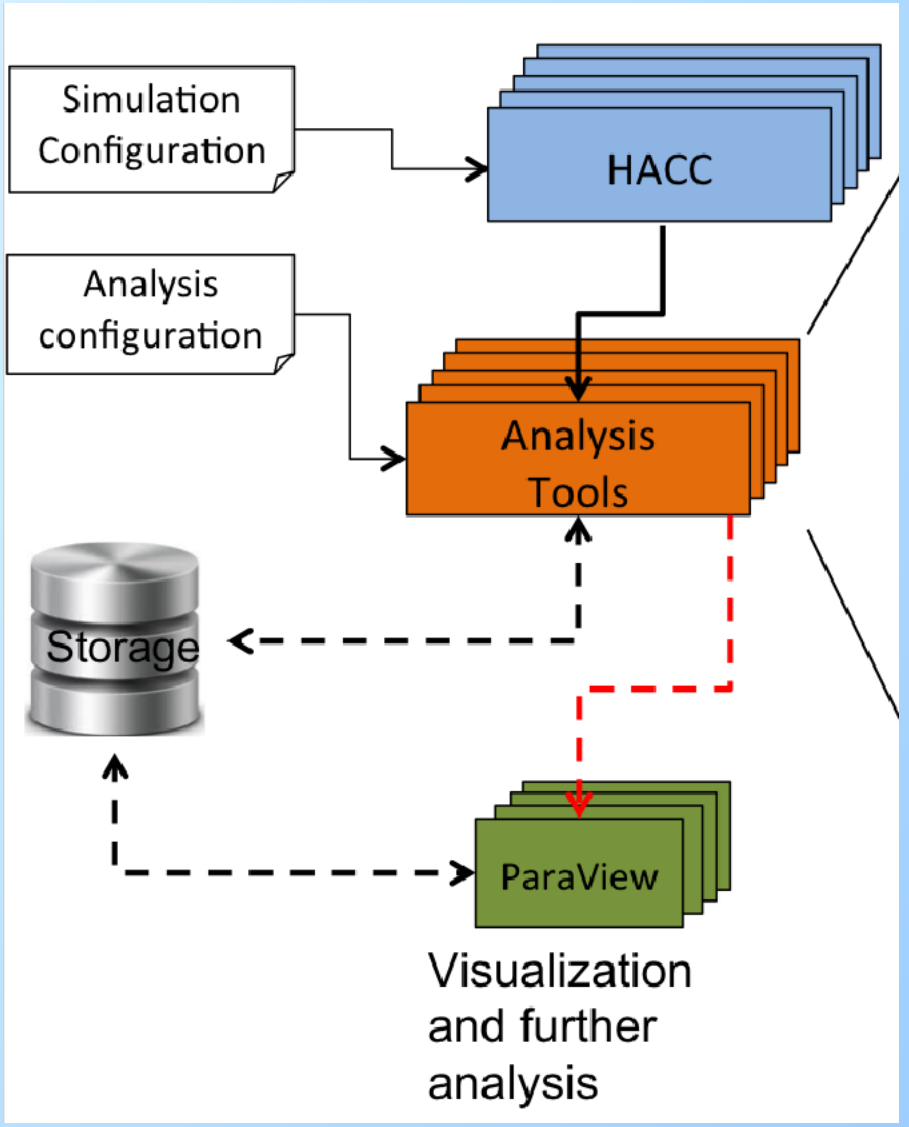
Extensions

In-situ Integration and Poisson Center Finder

LA-UR-14-25437

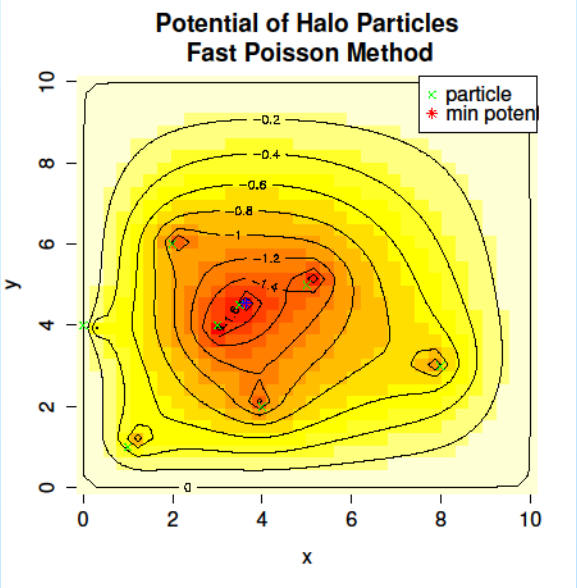
In-situ Integration in HACC

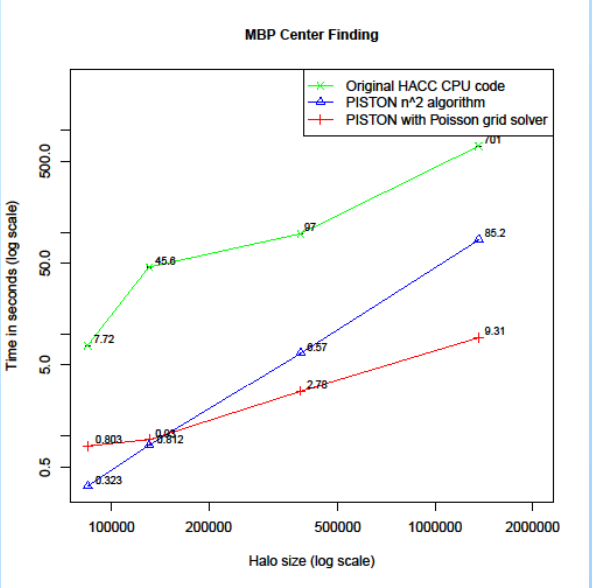
- Successfully ran 500 time-step, 512^3 particle simulation on Moonlight using our GPU halo and center finders integrated with HACC in-situ
- Integrated with CosmoTools



Extension: Potential Field Algorithm

- Optimization for MBP center finder: for each halo…
  - Superimpose a grid over the (extended) extents of the halo
  - Estimate particle density on the grid using binning
  - Solve Poisson equation for the potential on the grid using FFT (actually DST for zero boundary conditions approximation)
  - Find the grid point with minimum potential
  - Search around the neighborhood of the minimum potential grid for the particle with minimum potential
  - Return the position of such particle as the halo center





Future Needs

Current Bottlenecks, Expected Trends, and Possible Hardware and Software Solutions

LA-UR-14-25437

Future Needs: Memory

- Current bottlenecks
  - Main memory: 8192^3 simulation run with only 1 rpn because number of “ghost” particles increases with total number of ranks, and even 2 rpn exceeded total system memory
  - GPU memory: Halo finding performed on CPU because of GPU memory limitations
- Expected future trends
  - Growth in computational rates will outpace growth in memory capacity and bandwidth
- Possible solutions
  - Hardware: More memory
  - Software: Streaming and external memory algorithms

Future Needs: Resiliency

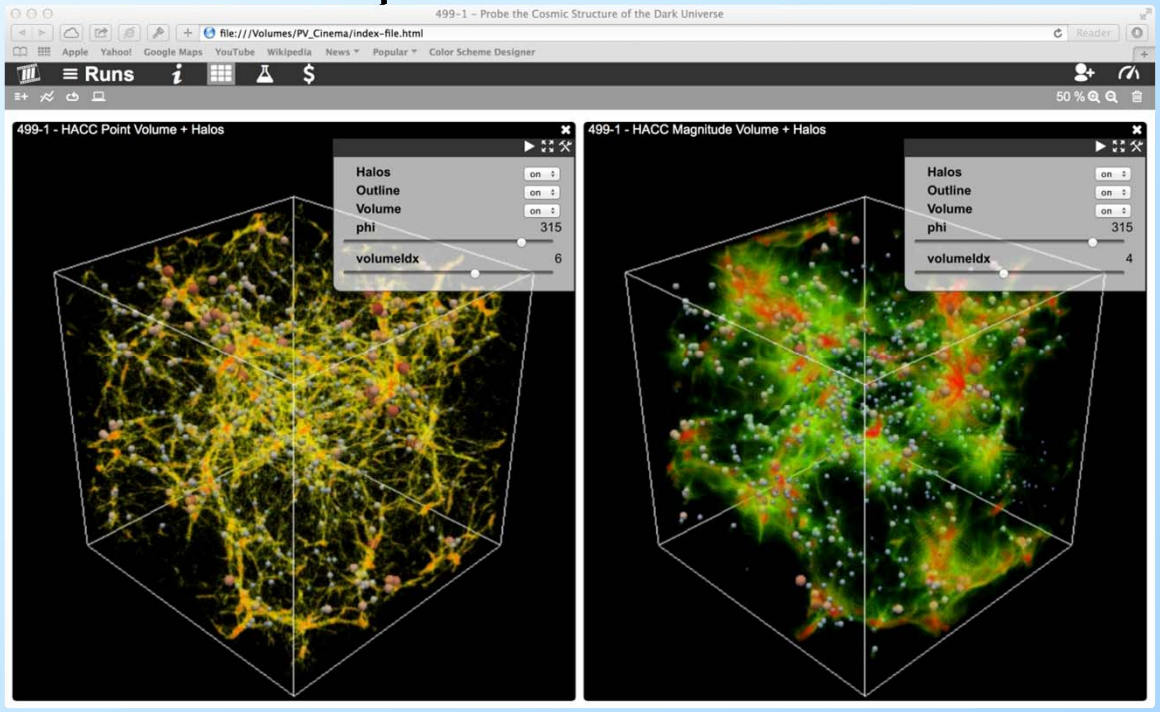
- Current bottlenecks (16k node simulation)
  - Node failures on start-up
  - Node failures while running
- Expected future trends
  - Systems will have more nodes with more cores; more things that could fail
- Possible solutions
  - Hardware: Components with longer MTBF
  - Software: Improved checkpointing (perhaps using burst buffers?); more robust run-time schedulers

Future Needs: Portability

- Current bottlenecks
  - Simulation code has to be rewritten to run, or at least to run efficiently, on new architectures
  - Some potential optimizations have been foregone because platform-specific optimizations would inhibit portability
- Expected future trends
  - More types of accelerators will come to market; individual systems will become more heterogeneous
- Possible solutions
  - Hardware: Standardize on an architecture (not likely)
  - Software: Write programs using higher-level abstractions

Future Needs: In-Situ

- Due to power and I/O constraints, we expect to need to perform analysis in-situ rather than in post-processing
- Traditional options: efficient but highly constrained in-situ, or expensive but highly interactive post-processing
- Recent work explores using image databases to allow trade-offs in the space between these extremes



Jim Ahrens, Patrick O'Leary, et. al.