

## Introduction

Chaparral is a library designed to solve radiation heat transfer problems across large, three-dimensional meshes. A major portion of this calculation involves comparing each pair of faces in the mesh to find the View Factor (VF) matrix.

The calculation of each element in the VF matrix can be easily distributed because each view factor depends only on the two faces it corresponds to (except for possible obstructions). However, in the current implementation, the data for the full mesh must be loaded into each process. We explore an alternative scheme using Charm++ for distributing the mesh data that will eliminate the need for replicating the mesh.

## View Factors and Chaparral

**View Factor (VF):** Describes how much of the heat radiated from one facet impacts another. For the sake of simplifying this study, we assume there are no obstructions between the facets. The view factor between facet  $i$  and facet  $j$  is  $F_{ij}$ :

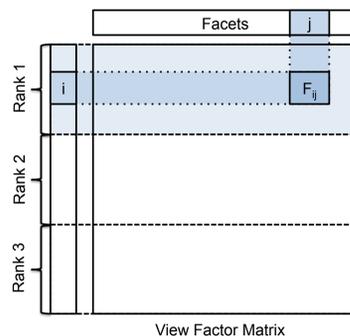
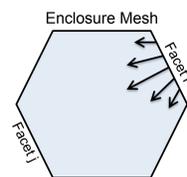
$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{(\cos \theta_i * \cos \theta_j)}{\pi r^2} dA_j dA_i$$

**VF Matrix:** A matrix describing the view factor for each pair of faces in the mesh. The memory required for this matrix is at least  $O(n^2)$ , regardless of the approach. This overhead could be decreased by writing the results out to a file as they become available. Because of this, our analyses concentrate on additional memory.

**Parallelization:** Chaparral's algorithm for calculating the VF matrix requires that the data be copied to every process. That is, the total memory usage across the entire VF matrix calculation excluding the VF matrix itself is:

$$O(n * p) \quad \begin{matrix} n \text{ facets} \\ p \text{ processes} \end{matrix}$$

**Project goal:** Avoid replicating data.



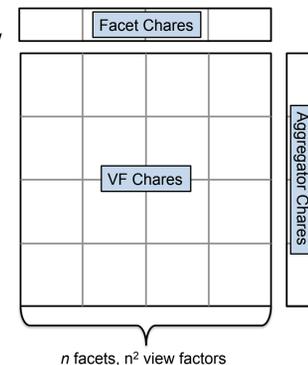
## View Factors with Charm – Chare Types

### How can the VF matrix calculation take advantage of over-decomposition?

There are two entities involved in Chaparral's calculation of the view factor matrix: facets and view factor scalars. The design for both of our approaches is based around these entities.

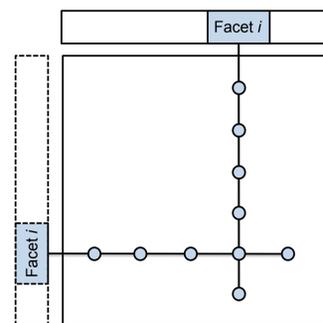
#### Three types of chares:

- **Facet chare** – one-dimensional array, each element corresponding to a facet in the mesh.
- **VF chare** – two-dimensional array, each element corresponding to the view factor between two of the facet chares.
- **Aggregator chare** – merges each row of VF chares into a single array.



In this implementation, each chare corresponds to a single face or a single VF. An improved but more complex implementation would instead have each chare responsible for a patch of facets and VFs.

## View Factors with Charm – Naïve Approach



- **Facet chares:** Read in a facet and send the facet data to every VF chare that needs it.
- **VF chares:** When the data for both facet chares are received, calculate the view factor for that pair.
- **On startup:** Each facet chare sends  $2 * n$  messages, each message containing the data for its face.

These messages are queued in Charm++'s RTS until a processor is available to calculate the VF for the corresponding faces.

- **Memory usage:** There are  $n$  facets, each of which send  $2 * n$  messages. This means that the memory usage excluding the VF matrix itself is:

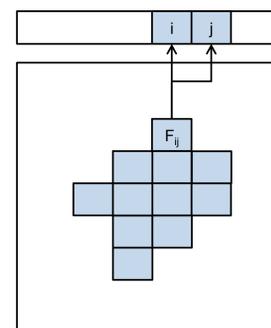
$$O(n^2)$$

## View Factors with Charm – Better Approach

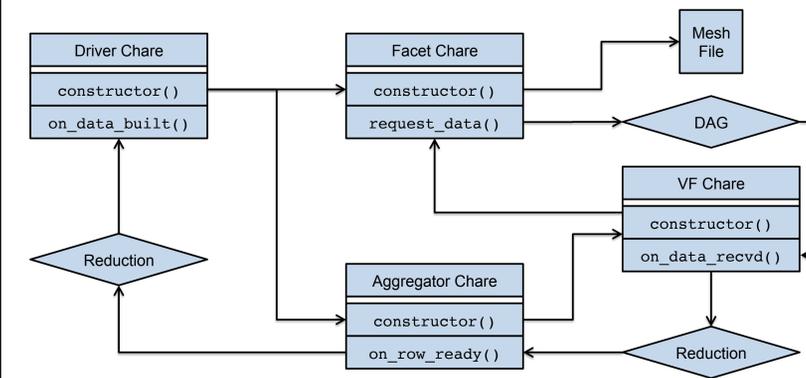
- **Facet chares:** Read in a facet and wait for requests for that data from other chares.
- **VF chares:** Request data from both facet chares. When it is received, calculate the view factor and free the memory used by the messages.
- **On startup:** The Charm++ RTS selects as many VF chares to run as there are processes. Each of these VF chares send a message to their respective facet chares, then fall asleep until those data are received. While they are asleep, the RTS selects other chares to run in their stead.

- **Memory usage:** There are  $n$  facets chares in the system. At any point in time, the RTS is transporting  $2 * p$  messages containing facet data. This means that, aside from the memory required to hold the VF matrix itself, the memory overhead is:

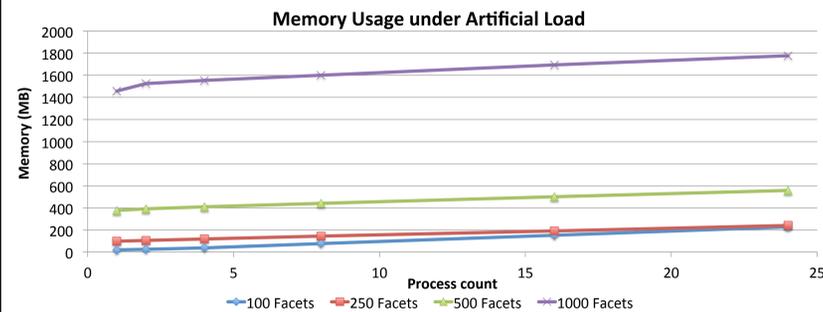
$$O(n + p)$$



## Chare Interaction



## Results



## Conclusions

In the current MPI implementation of Chaparral, the mesh file needs to be replicated to each processor. We have created a framework using Charm++ that allows distributed access to a single instance of the mesh from any process in the program, allowing for better scaling on many-core architectures.

The approaches outlined here assume each chare instance holds only one face in the mesh. While this has allowed us to experiment with the Charm++ system and improve our understanding of how the chares interact with each other, the overhead of each chare instance is overwhelming the useful work each chare does. One approach that would be useful to examine would be to have each chare responsible for a patch of faces, rather than just one. By using this patched approach, the overall overhead of the program could be decreased.

## References and Acknowledgements

- Glass, Micheal W. "CHAPARRAL: A Library for Solving Large Enclosure Radiation Heat Transfer Problems." Sandia National Laboratories
- Charm++, <http://charm.cs.illinois.edu/research/charm>

Thanks to:

- Neil Carlson (CCS-2) for his guidance with Chaparral and view factors as well as for his design suggestions,
- Jozsef Bakosi (CCS-2) for his invaluable help with learning and using Charm++,
- The ASC Integrated Codes Program for funding,
- The LANL Information Science & Technology Institute (ISTI) Parallel Computing Summer Research Internship (PCSRI).

## Charm++

**Charm++:** A parallel programming framework that encourages developers to break their problem into many small pieces that can be distributed throughout the system.

**Chare:** Lightweight, transportable objects that are responsible for computing a small portion of the problem. Communicate with each other via remote procedure calls.

**Charm++ Runtime System (RTS):** The environment in which Charm++ programs run. Manages messages between chares, decides which messages should be handled next, and manages load balancing across the physical machine.

**Over-decomposition:** The design approach of splitting a problem into as many small tasks as possible, then allowing the RTS to decide which tasks should be allowed to run. This ensures there is always something for the RTS to schedule – hides the communication cost.

