A Distributed Algorithm for 3D Radar Imaging

Patrick Li
Simon Scott

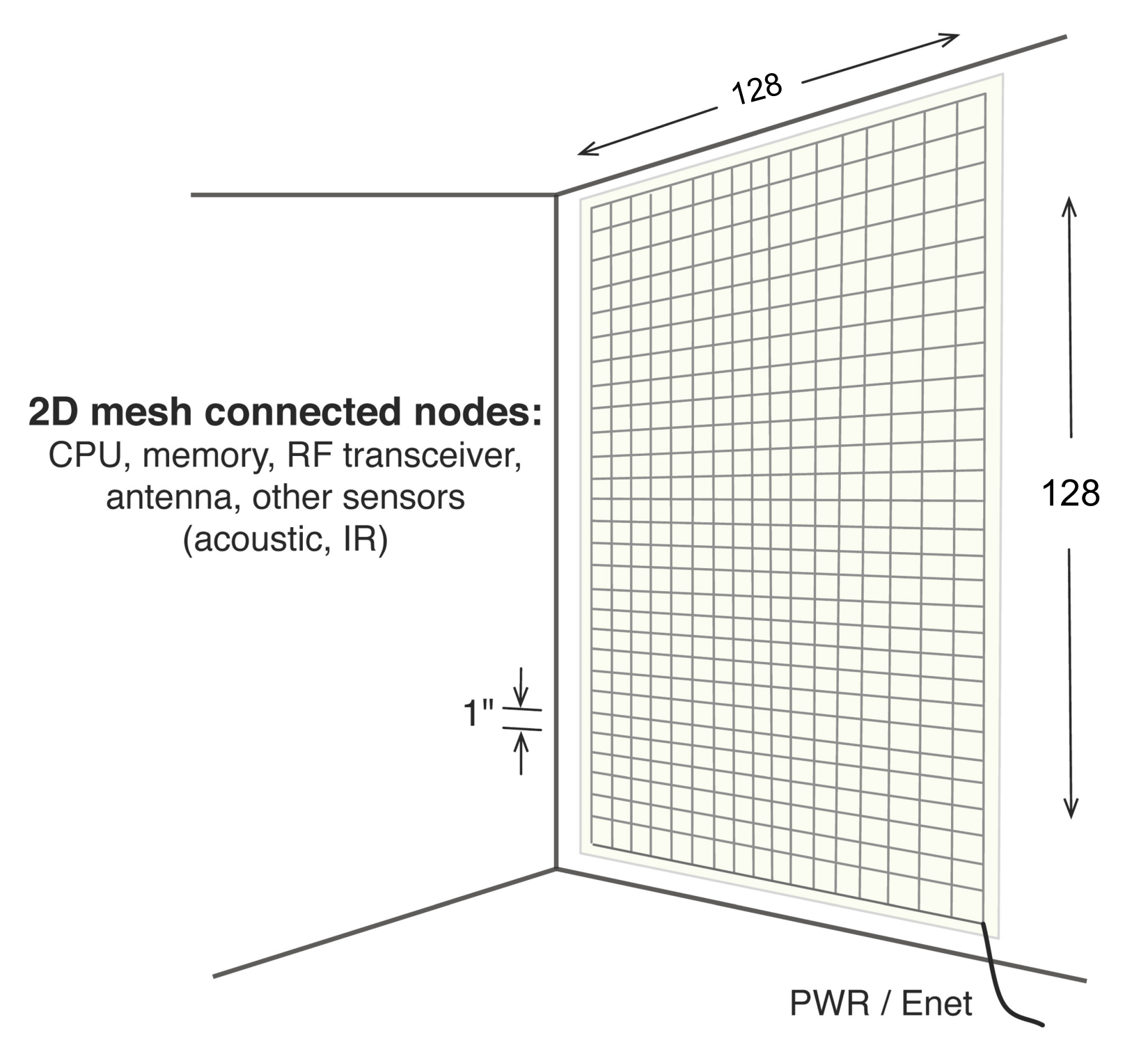
eWallpaper Imaging

eWallpaper: a smart wallpaper with thousands of embedded, low-power processors, connected in a 2D mesh. Each processor has its own radio transceiver.

Application: use the radio transceivers to perform 3D radar imaging of the room.

Algorithm: each radio transmits pulses and listens to the response. mm-wave synthetic-aperture radar (SAR) imaging techniques are used to form an image from the combined responses.

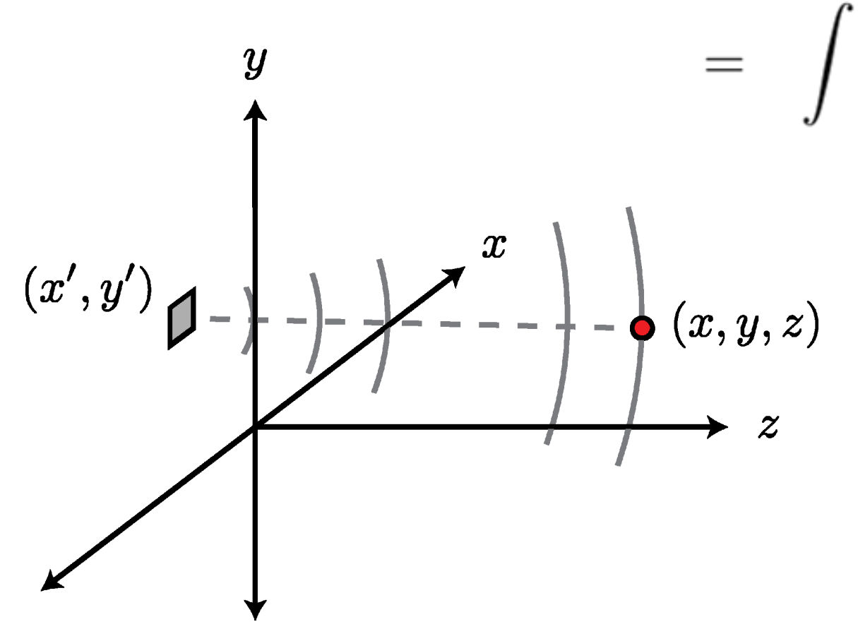
Challenges: distribution of the response amongst the 16 000 processors, the restrictive mesh topology and the limited local memory for each processor.


128
2D mesh connected nodes:
CPU, memory, RF transceiver, antenna, other sensors (acoustic, IR)
128
1"
PWR / Enet

Synthetic Aperture Radar

The antennas record the reflected wavefield at position, z=0, for each frequency, w, s(x,y,z=0,w). To reconstruct the wavefield at depth z, we downwards continue the recorded wavefield by multiplying by an appropriate phase offset, and integrate over the signals received at each antenna.

$$\begin{aligned} s(x,y,z) &= \int s(x,y,z,\omega)d\omega \\ &= \int (\int\int s(x',y',z=0,\omega)e^{-j\frac{2\omega}{c}||x'-x,y'-y,z||}dx'dy')d\omega \end{aligned}$$



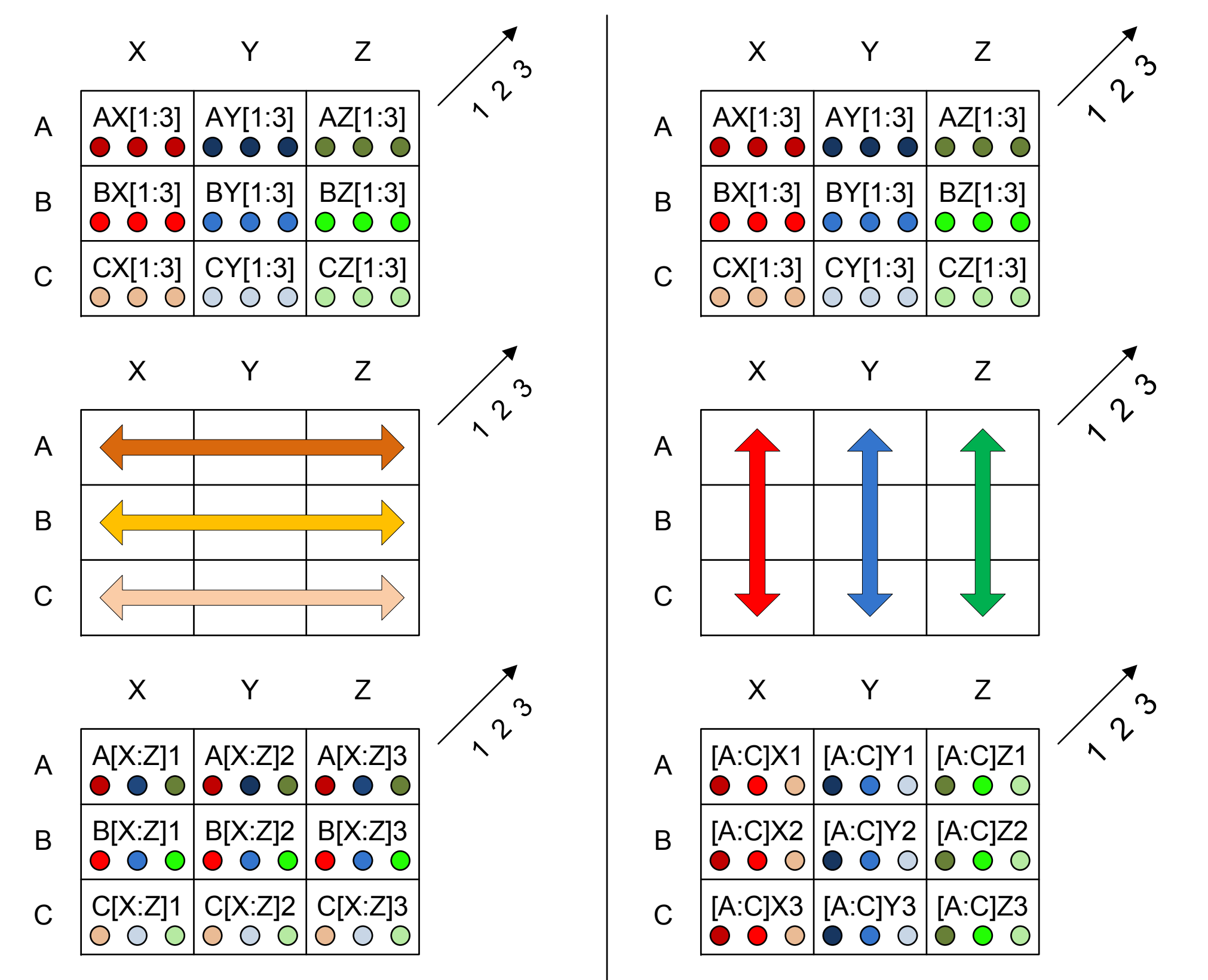
To perform the above operation efficiently, we can avoid the double integral by pre-transforming the recorded signal to the frequency domain. This also allows us to compute the wavefield at every depth z, using Stolt interpolation, and an inverse Fourier transform.

$$S(k_x,k_y,z) = \int S(k_x,k_y,z=0,k_z)e^{-jk_z z}dk_z = \mathrm{IFT}_{k_z}\{S(k_x,k_y,z=0,k_z)e^{-jk_z z_0}\}$$

where z0 is the minimum distance to the target. Thus the complete algorithm is

$$s(x,y,z) = \mathrm{IFT}_{k_x,k_y}\{\mathrm{IFT}_{k_z}\{\mathrm{Stolt}\{\mathrm{FT}_{x,y}\{s(x',y',z=0,\omega)\}\}e^{-jk_z z_0}\}\}.$$

The Algorithm



Row-wise Transpose **Column-wise Transpose**

Each Node's Operations for the Imaging Algorithm

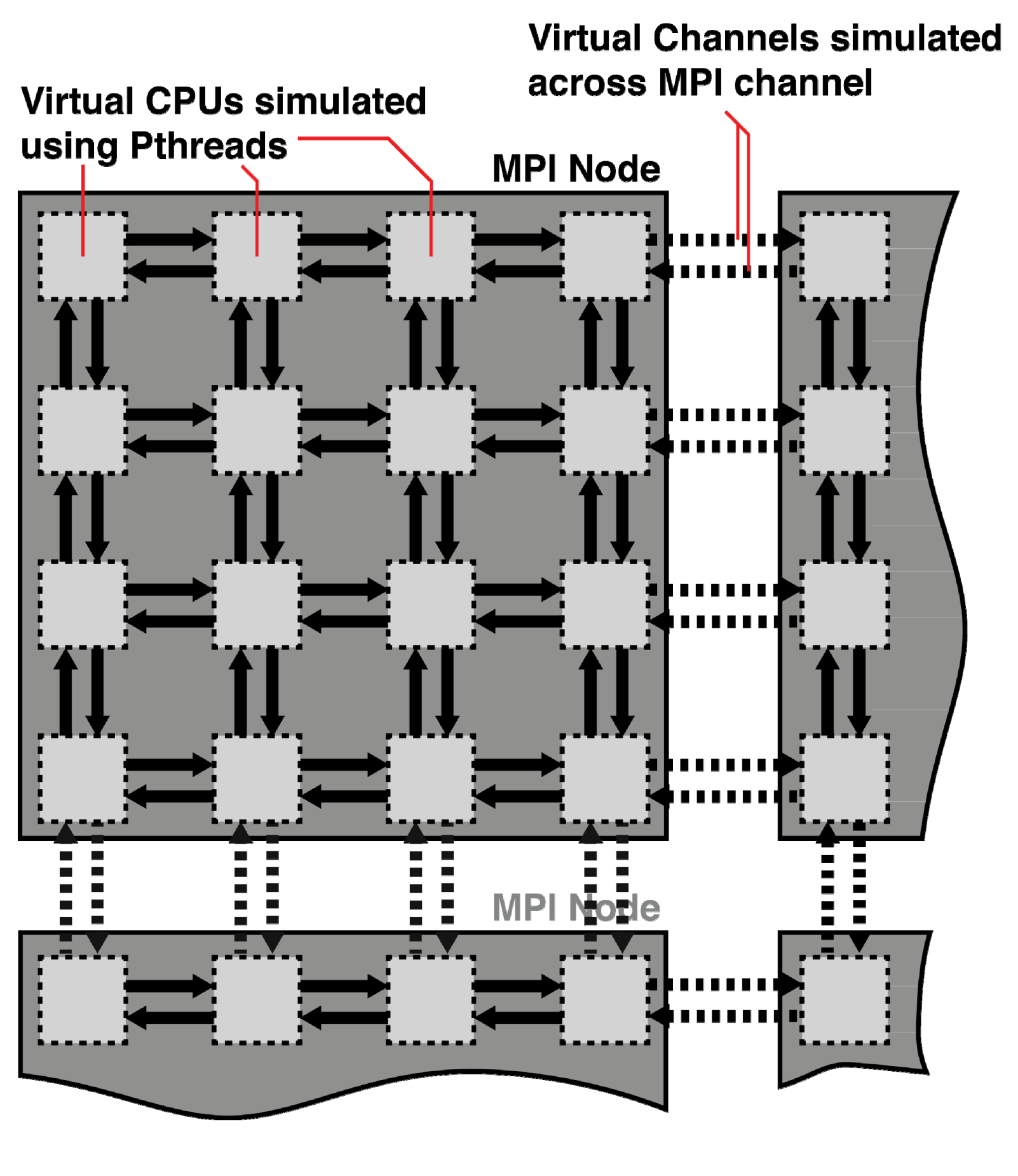
- Row-wise Transpose
- 2x 1D FFT (128 point)
- Column-wise Transpose
- 2x 1D FFT (128 point)
- Row-wise Transpose
- Point-wise vector multiply (256 elements)
- Stolt linear Interpolation (256 elements)
- 1x 1D IFFT (128 point)
- Column-wise Transpose
- 2x 1D IFFT (128 point)
- Row-wise Transpose
- 2x 1D IFFT (128 point)
- Column-wise Transpose

Computation in yellow, communication in grey

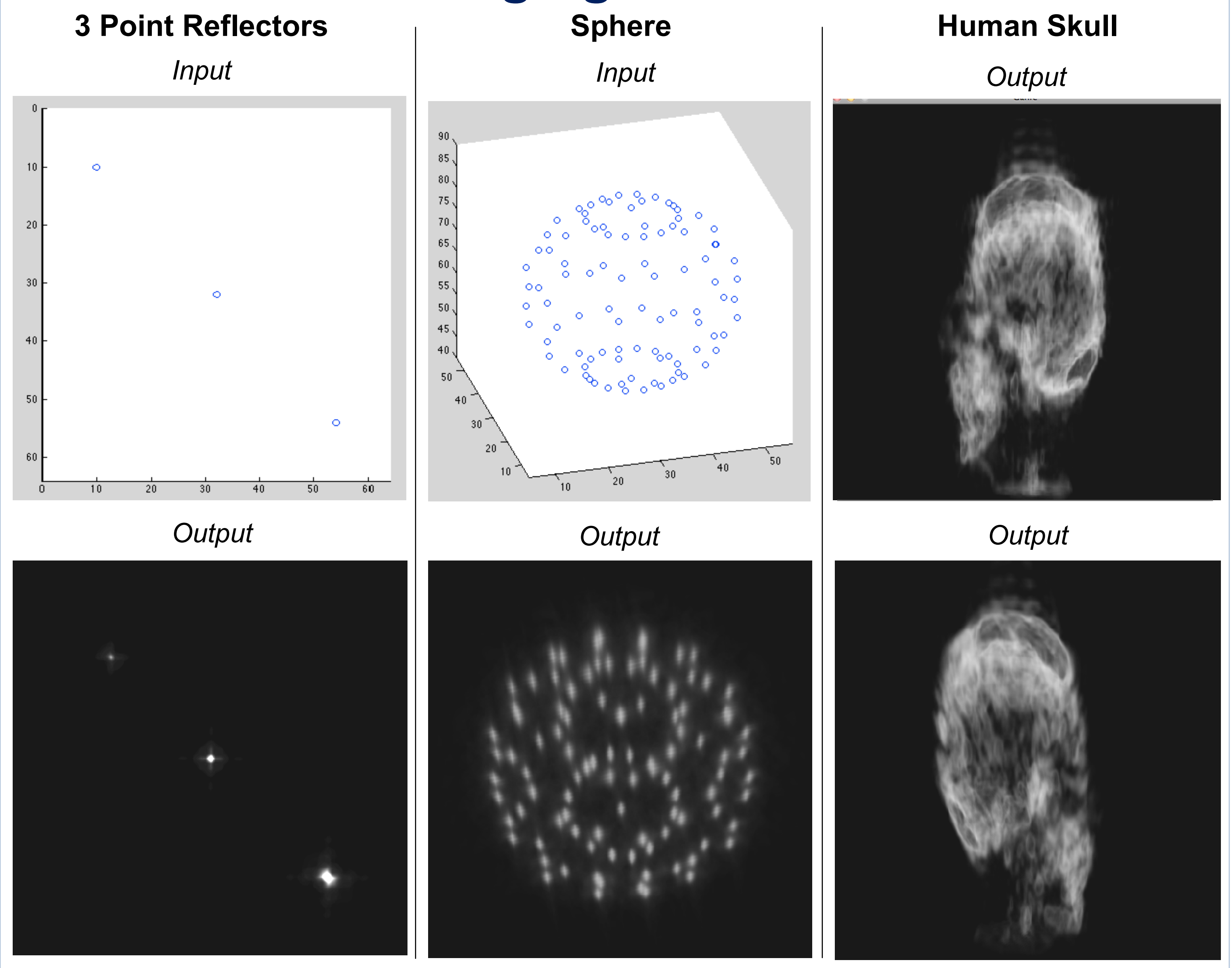
- Diagram shows 3 x 3 processor grid, each with 3 local values.
- Exactly two hop-delays are required to perform row- or column-wise transpose.
- eWallpaper contains 128 x 128 processors, each with 256 values.

Functional Simulator

- MPI based functional simulator enables fast prototyping and debugging for eWallpaper applications.
- eWallpaper applications are written in SPMD style. One instance of the program is launched for each eWallpaper CPU.
- Each virtual CPU is simulated in its own thread, with blocking sends and receives implemented via mutexes.
- Across MPI node boundaries, virtual channels are implemented via MPI_Isend and MPI_Irecv functions.
- The number of virtual CPUs per MPI node can be adjusted according to computational intensity and is transparent to the simulated application.

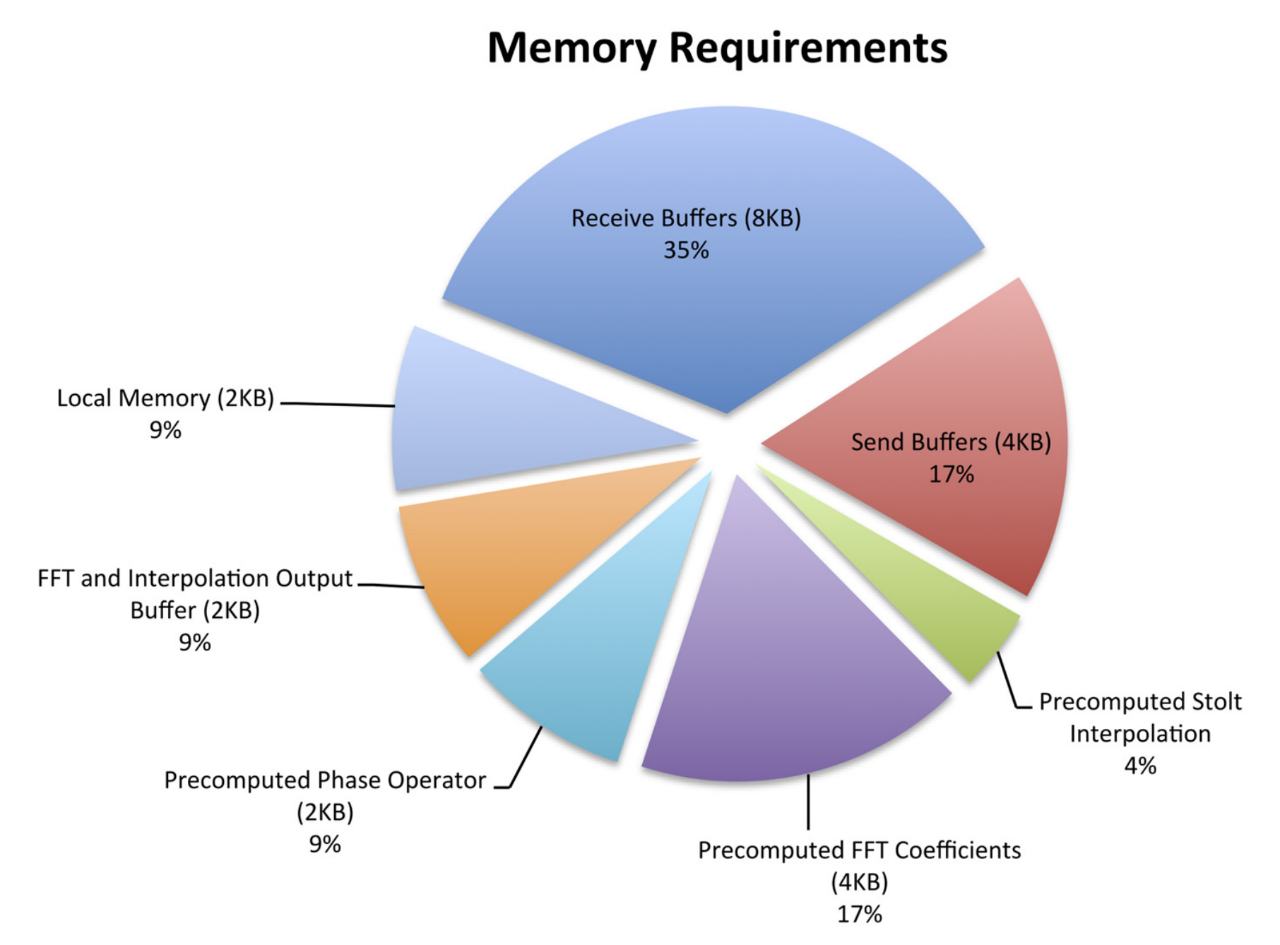

Virtual Channels simulated across MPI channel
Virtual CPUs simulated using Pthreads
MPI Node
MPI Node

Imaging Results

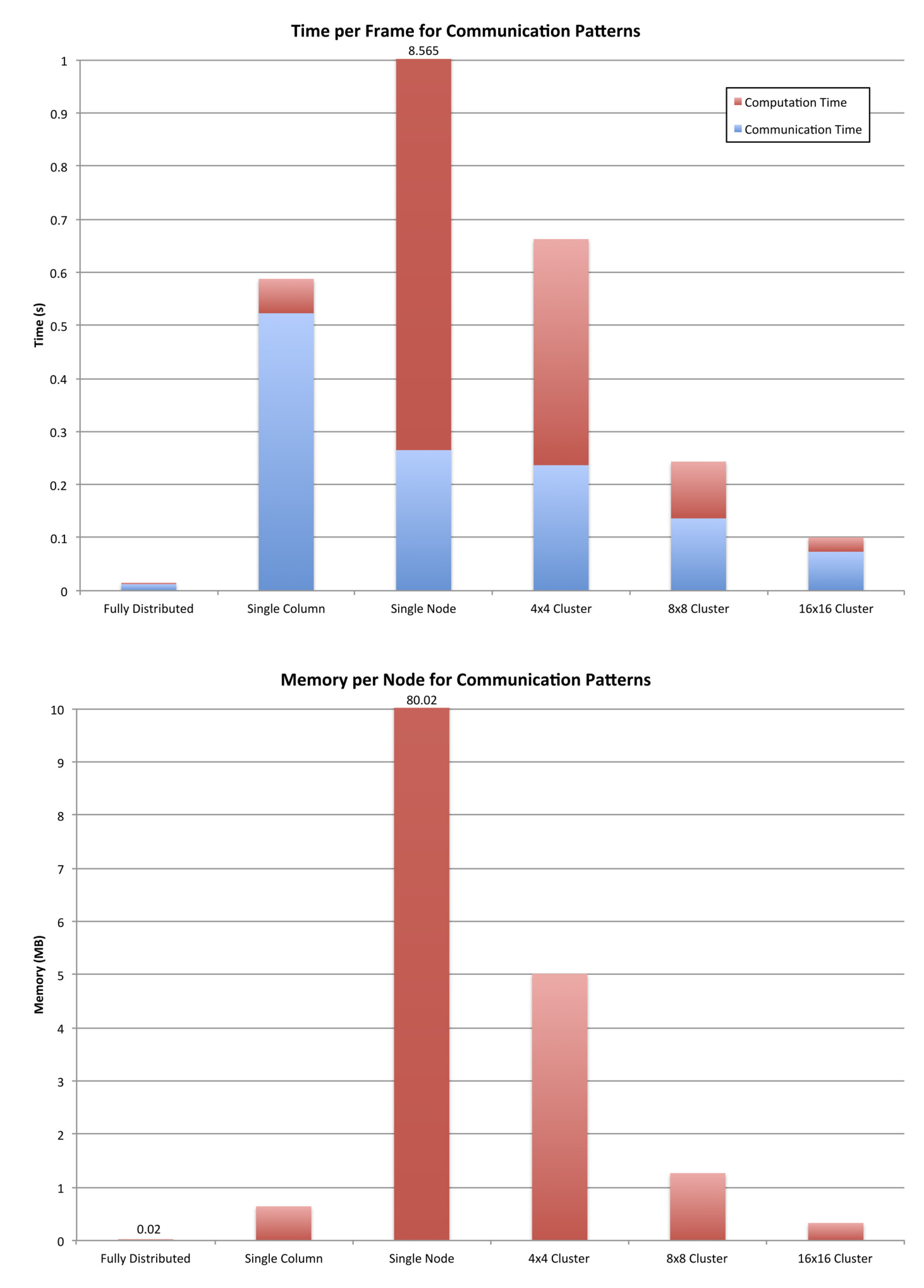


Timing Model and Network Simulator

- The application code that ran on the functional simulator was analyzed and a timing model was developed.
- The model showed that each processor spends > 90% of its time communicating.
- A Python-based discrete-event simulator was therefore built to accurately simulate network traffic on the eWallpaper.
- The components of the inter-processor communication protocol are accurately modeled, using packet transmission, reception and collision discrete events.
- Adjustable parameters for bandwidth and latency enable accurate modeling of the eWallpaper network.
- Memory usage for the imaging application was analyzed and is shown alongside.

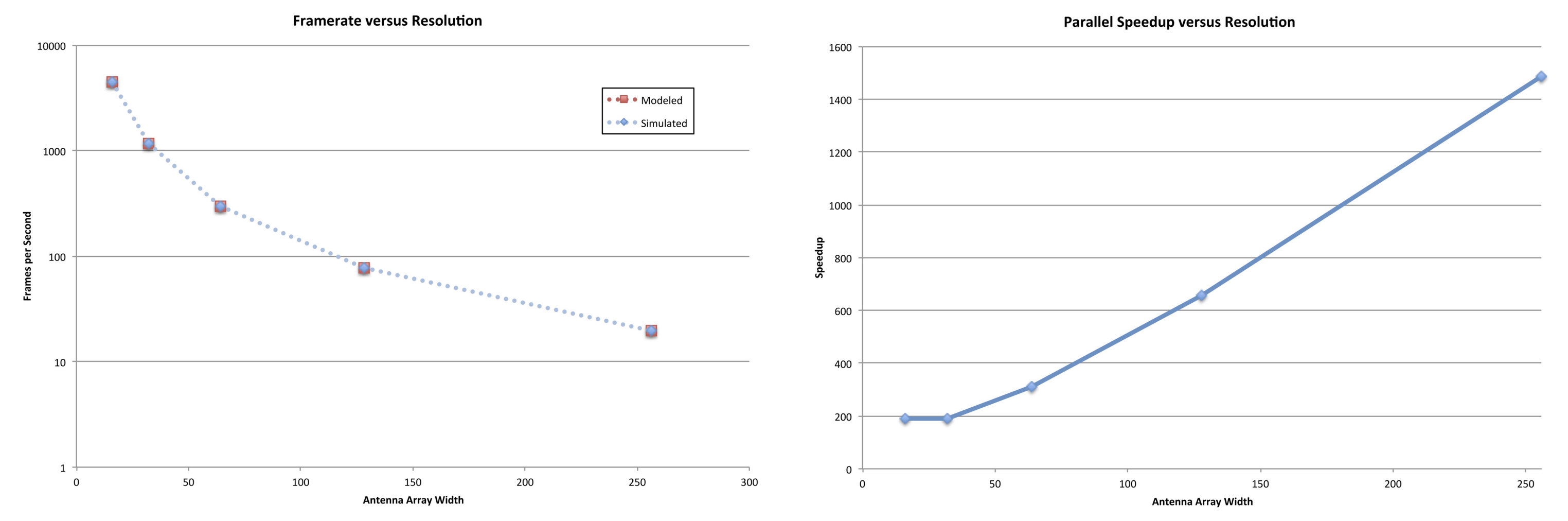


Investigation of Communication Patterns

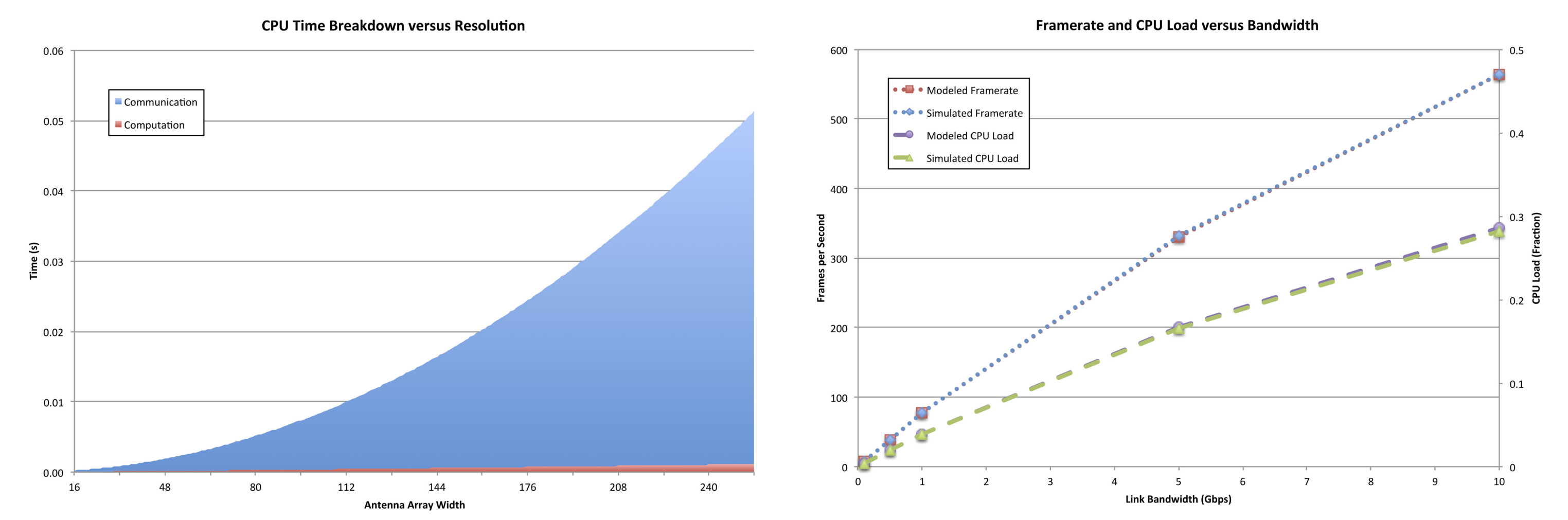


- Alternative communication patterns were simulated to determine if higher framerates could be achieved.
- **Fully distributed:** The previously described algorithm, where the work is evenly distributed amongst nodes.
- **Single Column:** Each processor forwards its data to the left-most node in its row. The nodes in this column do all the processing.
- **Single Node:** Each processor forwards its data to a single processor, which does all the processing.
- **Cluster:** The processors forward their data to a small cluster of nodes in the center.
- The graphs show that the fully-distributed pattern is the fastest and requires the least memory.
- The single column, single node, and 4x4 cluster and 8x8 cluster patterns are not viable, as they exceed the amount of available memory per node (100KB).

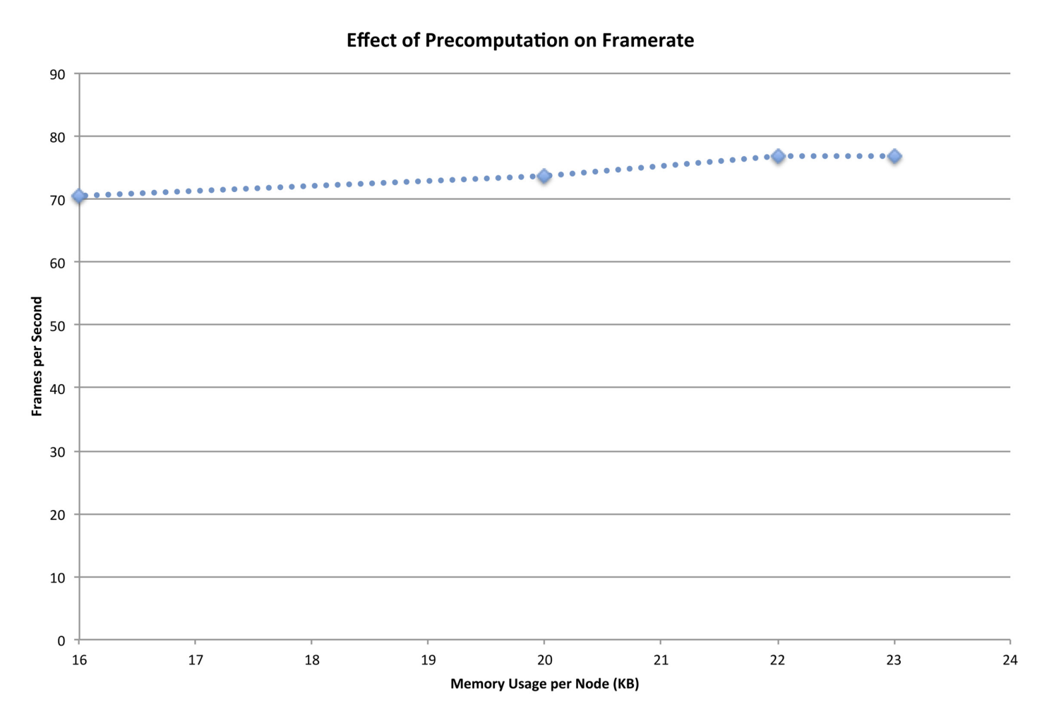
Performance Results



At our planned resolution of 128 x 128 antennas, the fully-distributed algorithm achieves a framerate of 75 fps. It is 600 times faster than the naïve serial implementation (the single node communication pattern described above). The left curve shows that even at a resolution of 256 x 256, realtime video framerates can be achieved. The framerate predicted by the timing model matches the simulated results.



The left curve shows that as the resolution increases, the execution time becomes dominated by communication costs. At the proposed link bandwidth of 1Gbps, the achieved 75 fps results in a CPU load of 0.03. As the bandwidth increases, the communication time decreases, resulting in a higher framerate and better CPU utilization. A minimum bandwidth of 250Mbps is required to achieve realtime framerates (25 fps).



The above graphs assume that the Fourier transform roots of unity, the downward continuation operator and the Stolt interpolation indices are precomputed. If these are instead calculated as required, the memory usage can be decreased to 16KB per node, with a small decrease in framerate.

Conclusions and Future Work

We developed an imaging algorithm for the eWallpaper that achieves realtime framerates with feasible memory and bandwidth requirements. We are currently building the first hardware prototype using FPGAs.