

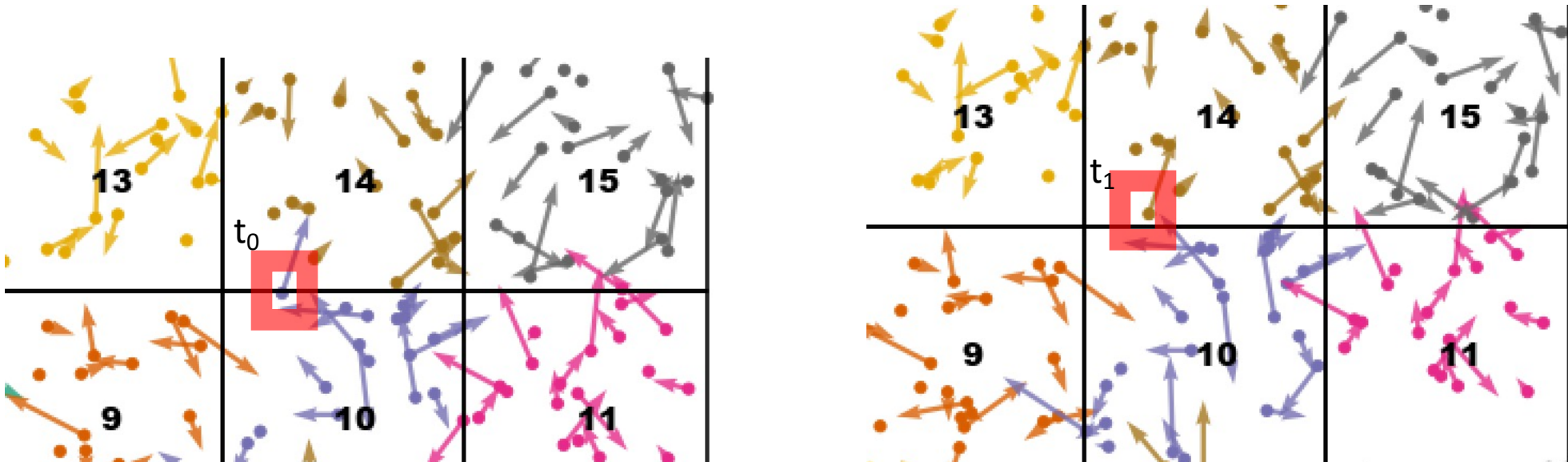
Particle simulations on the IPU

Show and tell

1 Feb

The problem:

- N-body/particle systems don't seem like they'd work in Poplar because:
 - I need more complex data structures (e.g. oct-trees)
 - I need to pass variable amounts of data to unknown destinations depending on particles



Investigation: More flexible data access

I want to

- Have dynamically sized data
- Represent data structures like vectors, hashmaps, oct-trees etc.
- Have heterogenous data (bools, ids, offsets/ptrs, Particle structs) etc.
- Use pointer memory more efficiently (256KiB = only need 18 bits for a pointer. Could load 3 in one 64-bit load etc.)
- Share these data structures between different Vertex invocations
- Share this with my 6 workers / other threads

Why is this hard in Poplar?

- Programming abstraction is at three levels
 - Host program
 - Data transfers, graph declaration & compilation, execution etc.
 - Compute graph
 - Declare and map Tensors (like PGAS global arrays)
 - Instantiate operations (Vertexes) and map to cores
 - Wire up Vertexes and Tensors
 - **Schedule communication between distributed cores**
 - **Always the same amount of data between the same cores (statically compiled)**
 - Vertexes
 - Only access local memory
 - Implement operation
 - No comms here

Poplar Vertex limitations

- No malloc (because no heap)
- No fancy members
- No access to another Vertex's data
- Only very simple types in interface
- Input, Output, InOut semantics
- Only access own core memory
- Can't schedule data comms at this level

```
using MyOwnStructType = struct {float potato; int pumpkin;};
```

```
class ExampleVertex : public Vertex{  
public:
```

```
    ✓ Input<bool> aSimpleType; // OK
```

```
    ✗ Input<MyOwnStructType> aComplexType; // NOPE
```

```
    // Error: Field 'ExampleVertex.aComplexType' has unsupported field type 'MyOwnStructType'
```

```
    ✓ Input<Vector<char>> aList; // OK
```

```
    ✓ Vector<Input<Vector<bool>>> aListOfLists; //OK
```

```
    ✗ Vector<Vector<Input<Vector<float>>>> moreThan2D; // NOPE
```

```
    ✓ unsigned int someSimpleInitialValue; // OK
```

```
    ✗ unsigned int someArray[100]; // NOPE
```

```
    //Error: Field 'ExampleVertex.someArray' has unsupported field type 'unsigned int [100]'
```

```
    ✗ MyOwnStructType something; //NOPE
```

```
    // Error: Field 'ExampleVertex.something' has unsupported field type 'MyOwnStructType'
```

```
    ✗ char *buffer1; //HELL NO
```

```
    //Error: Field 'ExampleVertex.buffer1' has unsupported field type 'char *'
```

```
    ✓ static float *buffer; // YES! But not very useful without malloc
```

```
    bool compute() {
```

```
    ✗ //buffer = malloc(sizeof(Potato));
```

```
    ✗ // Can't find malloc in stdlib
```

```
    ✗ buffer = new float[10000000];
```

```
//     terminate called after throwing an instance of 'poplar::link_error'
```

```
//     what():  undefined symbol _Znwm on tile 0
```

```
    ✓ auto thisIsNice = reinterpret_cast<::MyOwnStructType *>(&aList[0]);
```

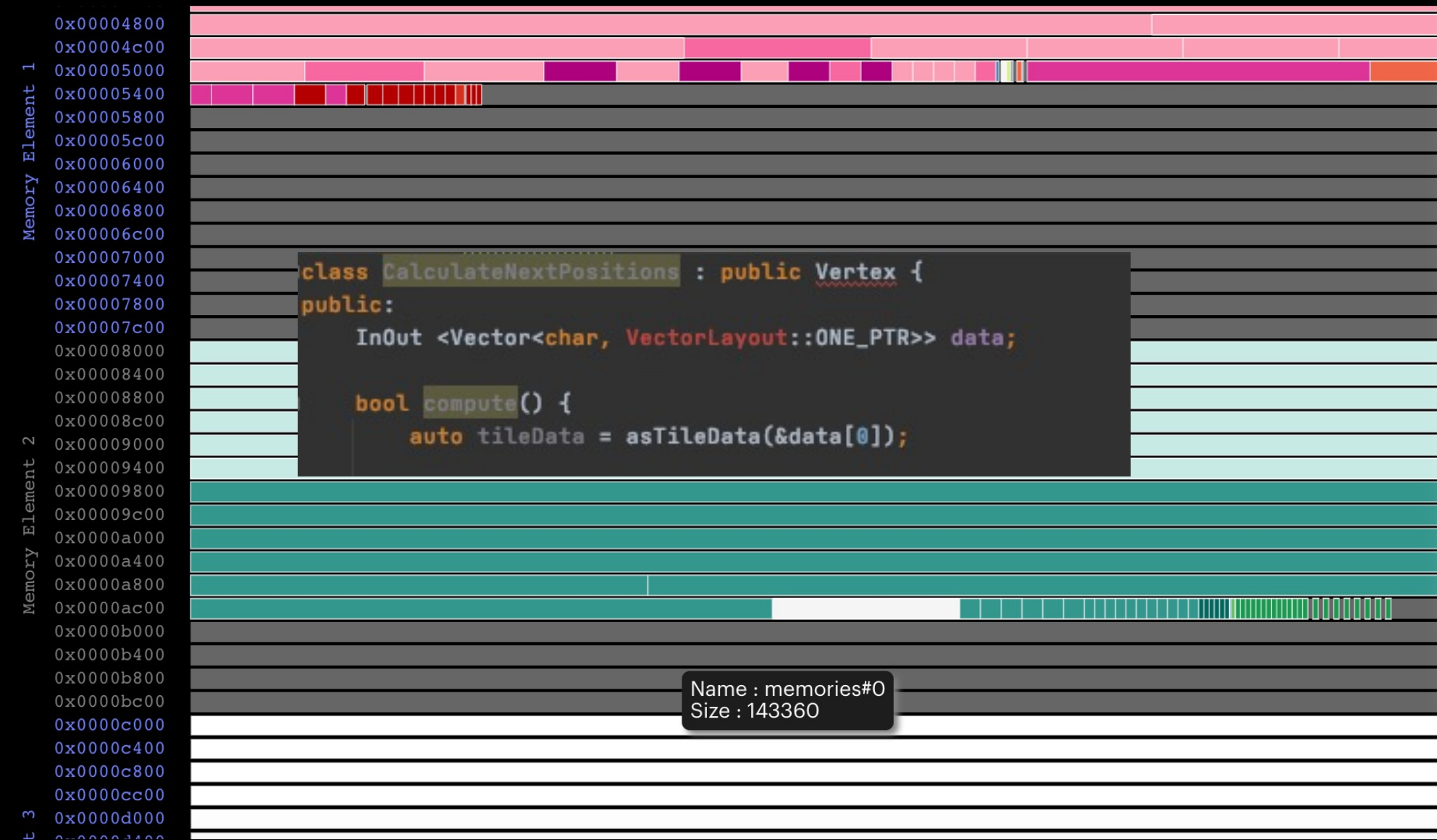
```
    // Skanky, but it will have to do
```

```
    return true;
```

```
};
```

```
};
```





- ☒ **CODE**
 - ☒ Control Table
 - ☒ Control Code
 - ☒ Vertex Code
 - ☒ Internal Exchange Code
 - ☒ Host Exchange Code
 - ☒ Global Exchange Code
- ☒ **VERTEX DATA**
 - ☒ Vertex Instance State
 - ☒ Copy Descriptor
 - ☒ Vector List Descriptor
 - ☒ Vertex Field Data
- ☒ **TEMPORARY VARIABLES**
 - ☒ Message
 - ☒ Host Message
 - ☒ Global Message
 - ☒ Rearrangement
 - ☒ Output Edge
- ☒ **MISC**
 - ☒ Multiple
 - ☒ Control ID
 - ☒ Host Exchange Packet Header
 - ☒ Global Exchange Packet Header
 - ☒ Stack
 - ☒ Instrumentation Results
 - ☒ Shared Code Storage
 - ☒ Shared Data Storage
 - ☒ Shared Structure State

InOuts<Vector<char>> for everyone

- Sharing a char * to memory you don't own between different threads seems like a great idea. **What could go wrong?**
- Rules
 - Never wire up any of this memory to vertexes on other cores
 - Threads only touch their 'own' sub-areas
 - Use Graph steps to sync access
 - Track worker indexes separately
 - Every vertex gets the flattened **InOut<Vector<char>>** member and must use the same alignments, memory bank definitions etc.
- Icky. But if it works, maybe Graphcore can be convinced to support this better.

An aside:

- Using this technique, we could get much better performance on the previous structured grid work too!
- Better weak scaling – less rearrangement from halo exchanges/alignment
- Simpler vertex code

Investigation: Sending variable amounts of data

Idea: Vary loop iterations not amount of data

- Graph element

```
class RepeatWhileTrue: public poplar::program::Program
```

```
#include <Program.hpp>
```

A program that evaluates the condition program, and if the predicate tensor is false it exits the loop.

If predicate tensor is true it evaluates the body program, and then loops to re-evaluate the condition program. This is like a C while statement.

Public Functions

```
RepeatWhileTrue(const Program &cond, Tensor predicate, const Program &body,  
const DebugContext &debugContext = {})
```

Construct a repeat while true program.

Parameters

- cond: The program evaluated before the body is evaluated.
- predicate: The scalar tensor that determines whether to execute the body.
- body: The body to execute when the predicate is true.
- debugContext: Optional DebugId and program name.

Keep "exchanging a particle" until nobody has a particle to exchange

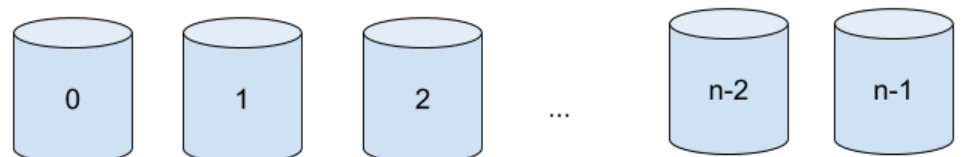
- Tensors “coreHasParticleToOffer” and “particleToOffer”, mapped 1 elem per core
- + map concatenation of Neighbours’ slices
- Reduction over “coreHasParticleToOffer” to find out whether to keep looping

[illegible]

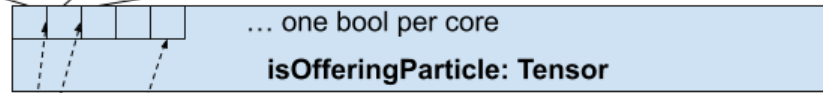
```

for (auto tileNum = 0; tileNum < NUM_PROCESSORS; tileNum++) {
    auto neighbours = findNeighbours(tileNum);
    assert(!neighbours.empty());
    Tensor particleToShedSlices = particleToShed[neighbours[0]].flatten();
    Tensor isOfferingSlices = hasParticlesToShed[neighbours[0]].flatten();
    for (auto i = 1; i < neighbours.size(); i++) {
        particleToShedSlices = poplar::concat(particleToShedSlices, particleToShed[neighbours[i]].flatten());
        isOfferingSlices = poplar::concat(isOfferingSlices, hasParticlesToShed[neighbours[i]].flatten());
    }
    auto v = graph.addVertex(csAccept, vertexType: "AcceptParticles",
                            connections: {
                                { field: "data", memories[tileNum]},
                                { field: "potentialNewParticles", particleToShedSlices},
                                { field: "isOfferingParticle", isOfferingSlices},
                            });
    graph.setInitialValue(field: v["numNeighbours"], neighbours.size());
    graph.setCycleEstimate(v, cycles: 100);
    graph.setTileMapping(v, tileNum);
}
exchangeParticles.add(Execute(csAccept));

```



Actual distributed
per-core
memories



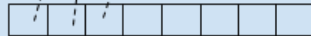
Tensors in graph
(like PGAS global
arrays)

```
auto hasParticlesToShed = graph.addVariable(
    poplar::BOOL,
    shape: {NUM_PROCESSORS}, debugContext:
    "hasParticlesToShed");
mapNPPerTile( & hasParticlesToShed, n: 1);
```

Kernels (Vertices) don't know about
tensors and only use local mem

AcceptParticles: Vertex

isNeighbourOfferingParticle: Vector<bool>



```
class AcceptParticles : public Vertex {
public:
    Input <Vector<float>> potentialNewParticles;
    Input <Vector<bool>> isOfferingParticle;
    InOut <Vector<char, VectorLayout::ONE_PTR>> data;
    int numNeighbours;

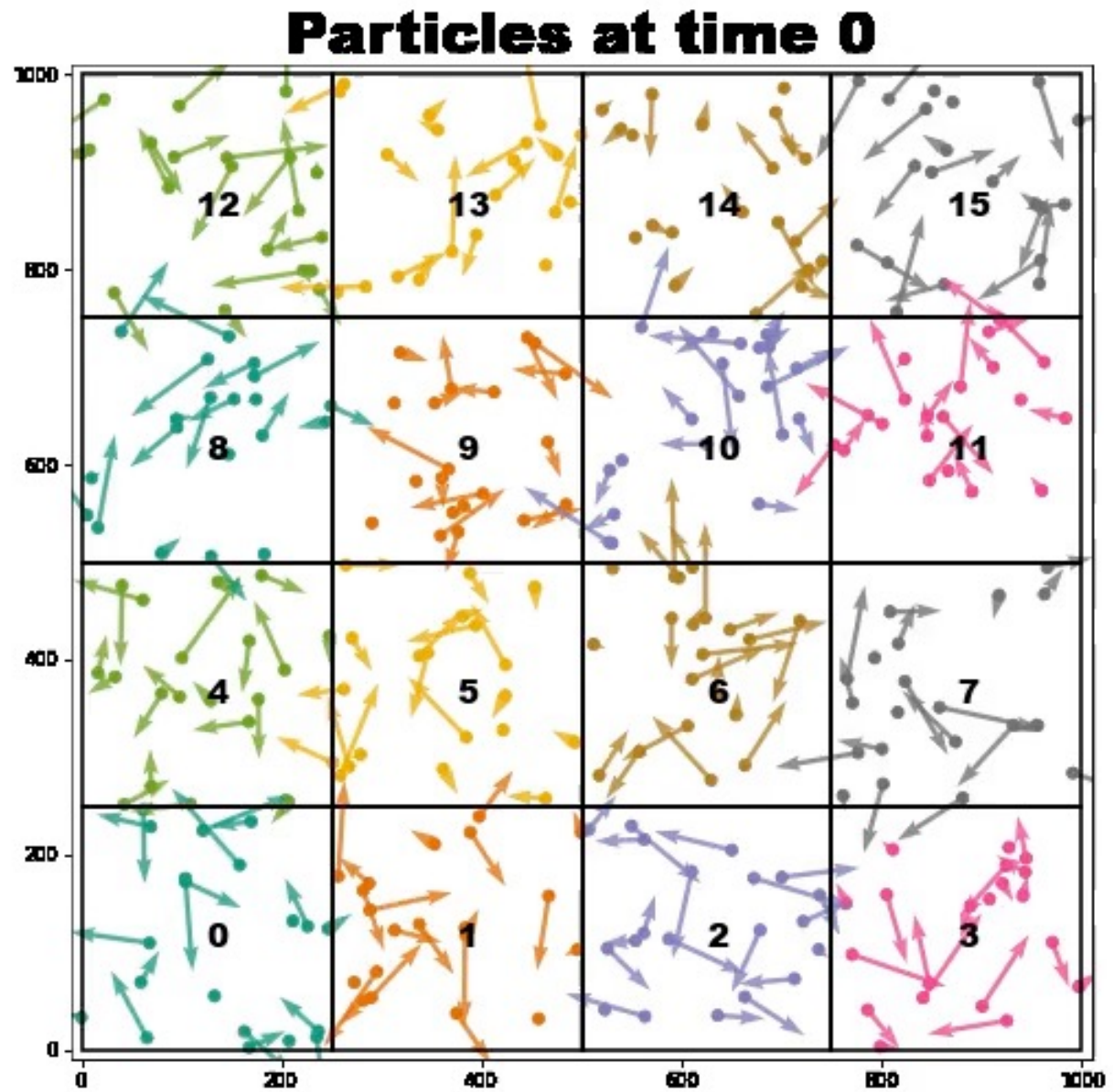
    bool compute() {
        auto tileData = asTileData(&data[0]);
        const auto particles = currentParticleBuffer(*tileData);
        auto maybeParticles = reinterpret_cast<Particle *>(&potentialNewParticles[0]);
```

```
for (auto tileNum = 0; tileNum < NUM_PROCESSORS; tileNum++) {
    auto neighbours = findNeighbours(tileNum);
    assert(!neighbours.empty());
    Tensor particleToShedSlices = particleToShed[neighbours[0]].flatten();
    Tensor isOfferingSlices = hasParticlesToShed[neighbours[0]].flatten();
    for (auto i = 1; i < neighbours.size(); i++) {
        particleToShedSlices = poplar::concat(particleToShedSlices, particleToShed[neighbours[i]].flatten());
        isOfferingSlices = poplar::concat(isOfferingSlices, hasParticlesToShed[neighbours[i]].flatten());
    }
    isOfferingSlices.dumpRegions();
    auto v = graph.addVertex(csAccept, vertexType "AcceptParticles",
        connections: {
            { field: "data", memories[tileNum]},
            { field: "potentialNewParticles", particleToShedSlices},
            { field: "isOfferingParticle", isOfferingSlices},
        });
    graph.setInitialValue( field: v["numNeighbours"], neighbours.size());
    graph.setCycleEstimate(v, cycles: 100);
    graph.setTileMapping(v, tileNum);
}
```

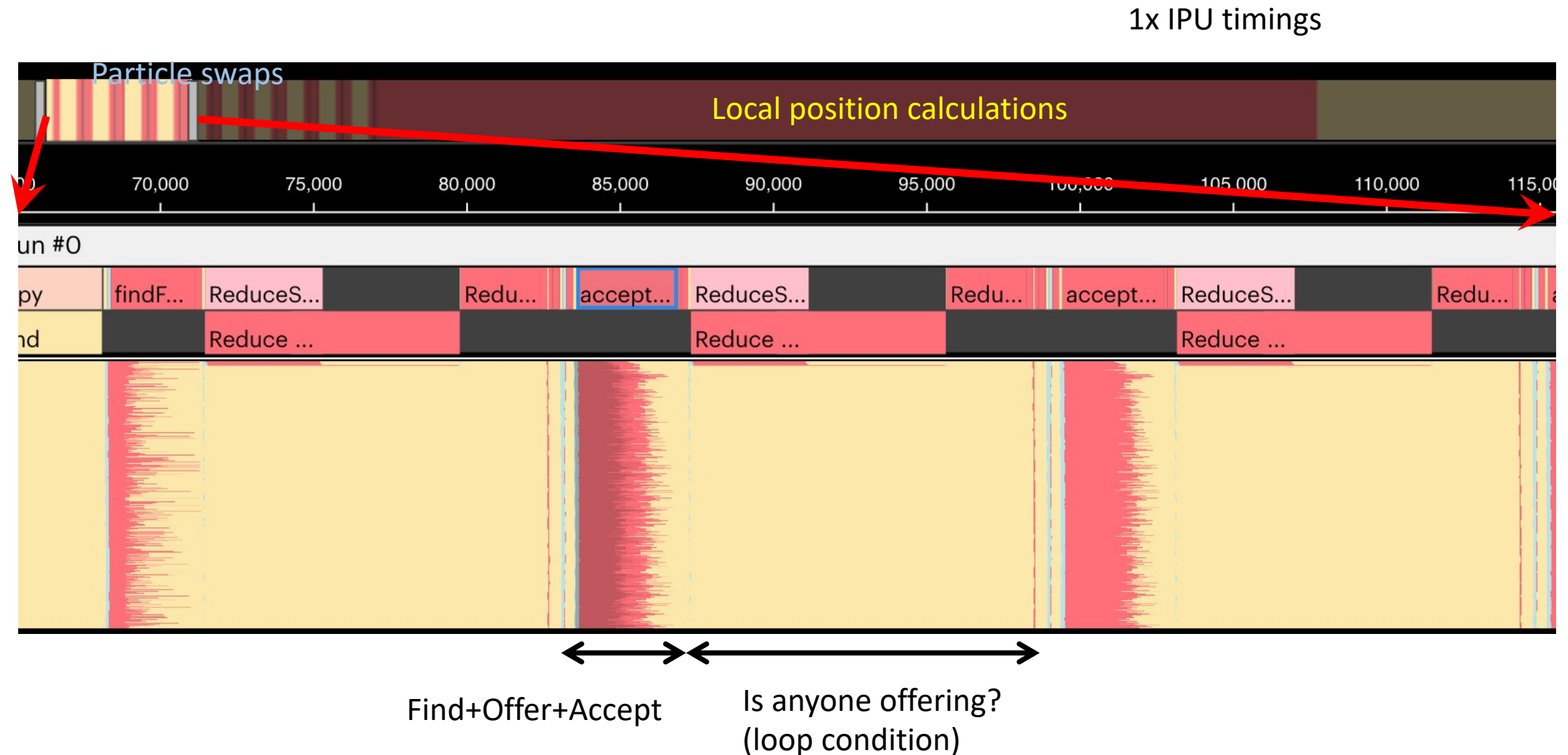
Wiring up remote
tensor slices to
vertexes
generates
comms

It works!

- Simple demo



In practice: reduction is the expensive bit

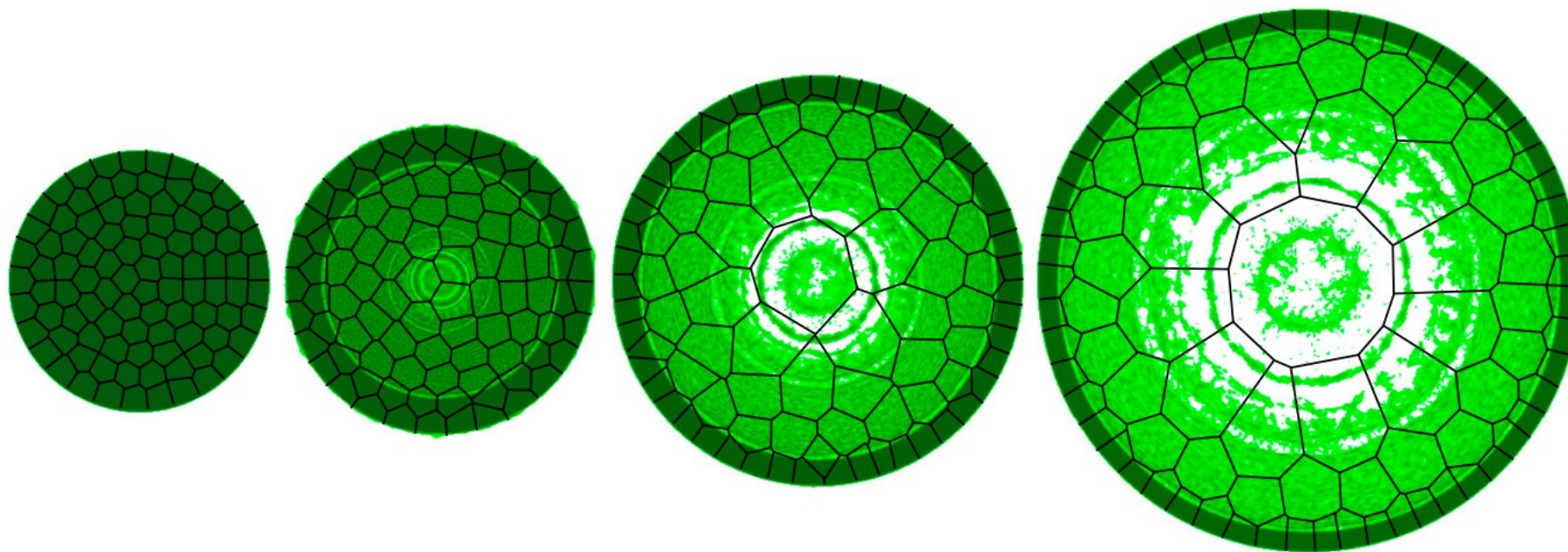


Can do the same for force interaction calcs:
Every core 'offers' a particle, every core calculated
contribution for particle

Opportunities for parallelism

- Calculate forces for N_p particles per loop-step (>120k on 16 IPUs in parallel)
- Do time integration for 120k in parallel
- Also able to vectorise (max 2 F32s or 4 F16s per op)
- Even just streaming the embarrassingly parallel ops (update particle pos)

Better load balancing



Short-range forces (e.g. Lennard-Jones)

$$V_{\text{LJ}} = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right]$$

- Can ignore particles further than, say 2sigma
- I.e. only nearest neighbour comms!

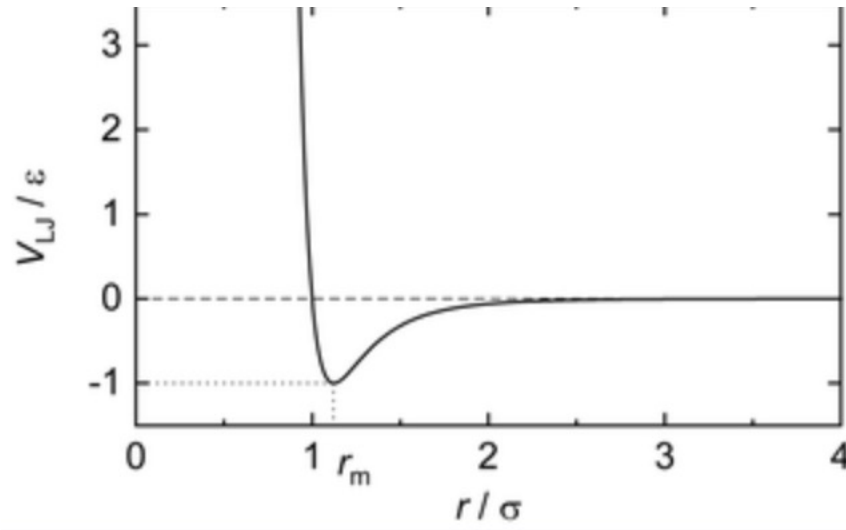


Figure 1. Graph of the Lennard-Jones potential function: Intermolecular potential energy V_{LJ} as a function of the distance of a pair of particles. The potential minimum is at $r = r_m = 2^{1/6}\sigma$.

Long-range forces

- E.g. Gravity, charge


$$\mathbf{F}_i = \sum_{j=1, j \neq i}^N \frac{G m_i m_j (\mathbf{x}_j - \mathbf{x}_i)}{|\mathbf{x}_j - \mathbf{x}_i|^3}$$

$$\mathbf{F}_{ij} = \frac{q_i q_j}{4\pi\epsilon_0} \frac{(\mathbf{x}_j - \mathbf{x}_i)}{|\mathbf{x}_j - \mathbf{x}_i|^3}$$

- Use quad/oct-tree approach to make NlogN
- Harder to scale well for IPU (need to exchange and consider N_p cores)

Time integration

- How to determine next position from forces acting on particle
- We store mass m , speed ($v.s$), direction($v.theta$), location $\underline{\mathbf{x}}$
- $\underline{\mathbf{F}}=m\underline{\mathbf{a}}$
- We have a way to calculate $\underline{\mathbf{F}}$ (Lennard Jones/Gravity/Charge/whatevs), so we can get $\underline{\mathbf{a}}$
- Leapfrog/Verlet

$$\mathbf{v}_i \left(t + \frac{3}{2} \delta t \right) \approx \mathbf{v}_i \left(t + \frac{1}{2} \delta t \right) + \delta t \mathbf{a}_i(t + \delta t)$$


- $\mathbf{x}(t + dt) = \mathbf{x}(t) + \mathbf{v} dt$

Collisions

- Within core
- Can use space filling curve (e.g. Morton ordering, Peano-Hilbert) to order particles for fast lookup and collision detection
- Or quadtree
- Can parallelise to 6 workers/core without more memory