IBM Research

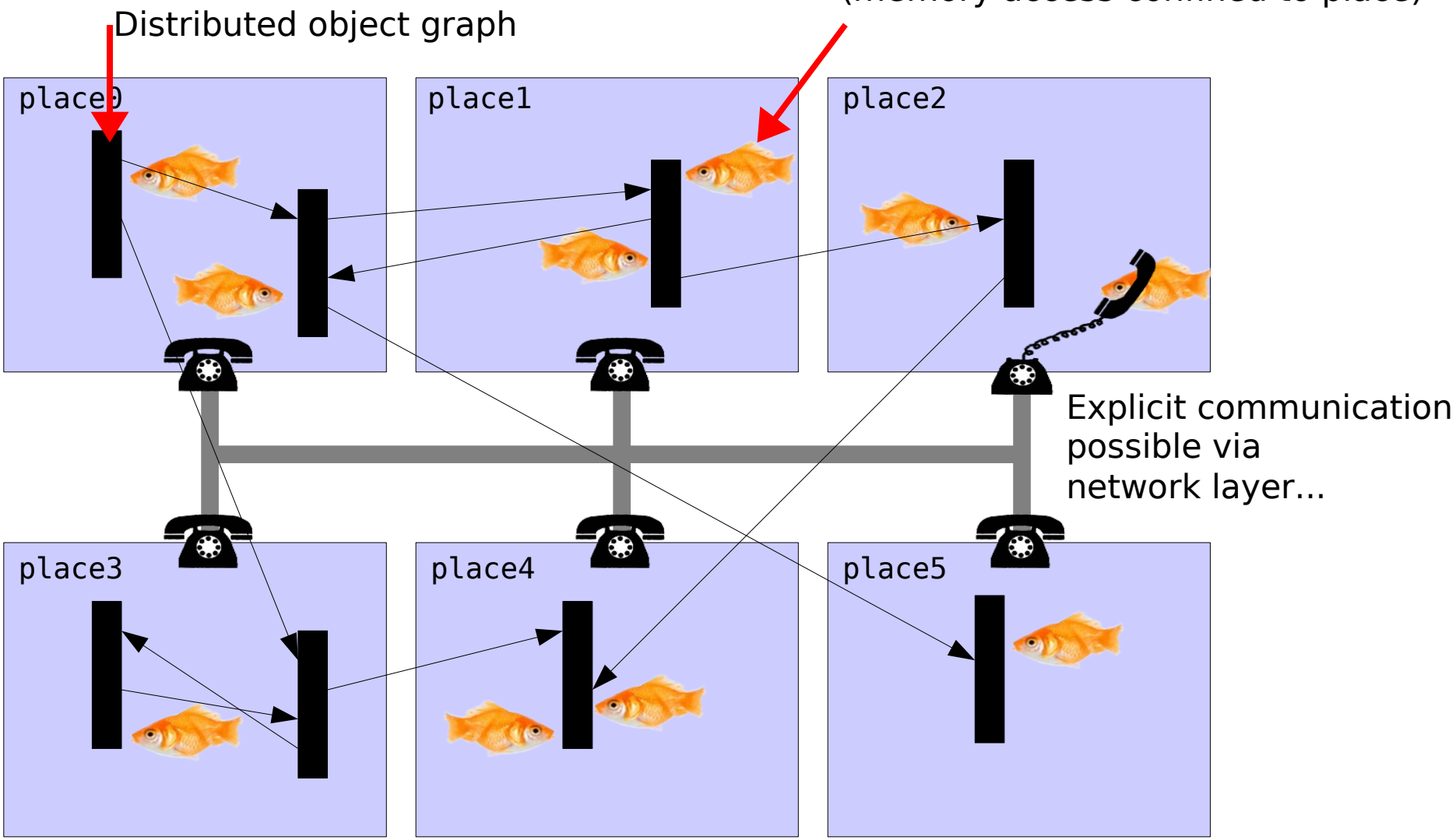
K-means clustering using the APGAS library

Dave Cunningham
Sayantan Sur
George Almasi
Vijay Saraswat
Calin Cascaval

Outline

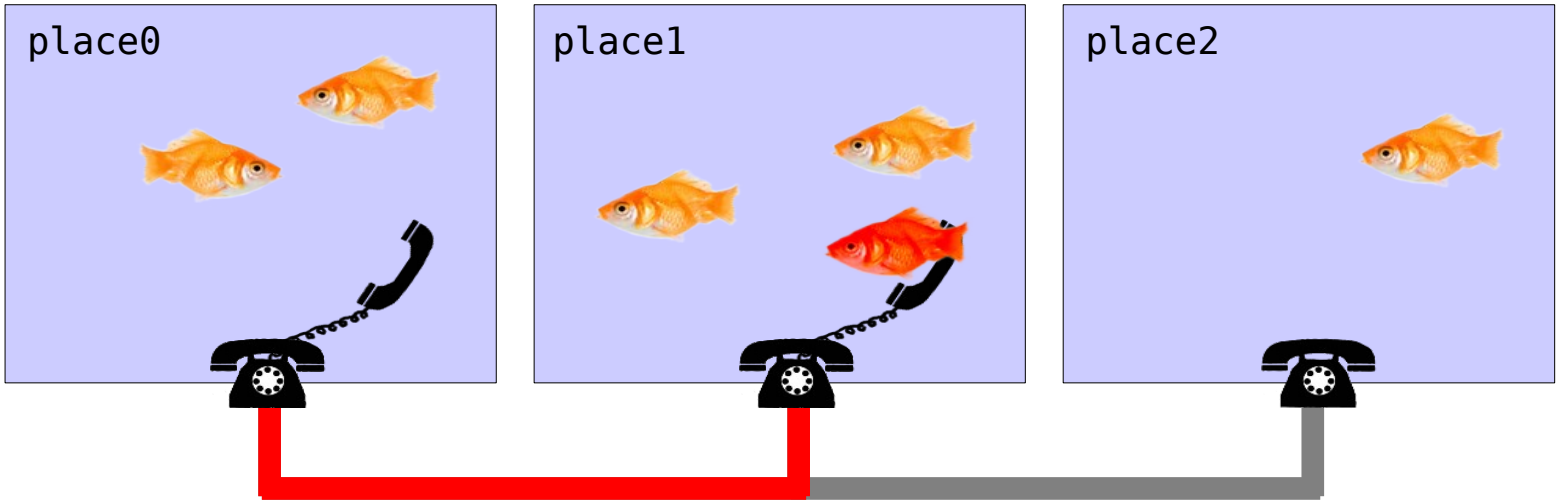
- Overview of APGAS model & library
- APGAS + Accelerators
- Description of k-means problem
- Description of Lloyd's Algorithm
- Parallel Lloyd's Algorithm with APGAS library
- Performance results
  - CPU cluster (P570)
  - Cell (QS21 blades)
  - GPU (NVIDIA Tesla S1070)

APGAS – memory & threads



Communication primitives 1 – Active message

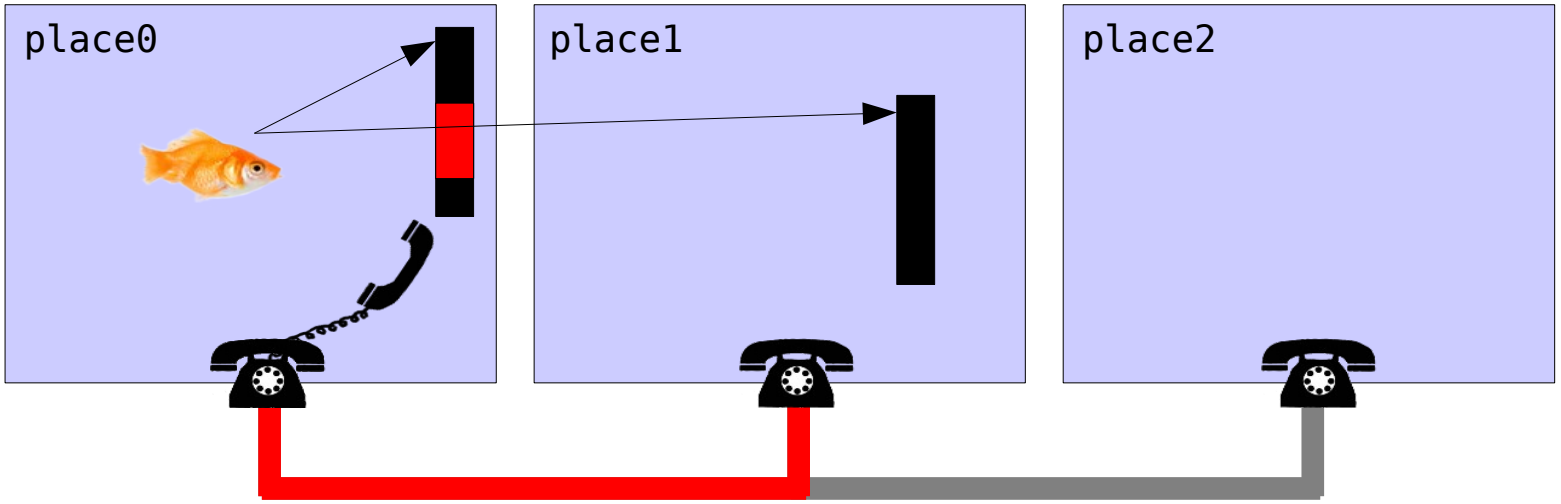
Execute code at a particular place:



1) Thread initiates AM
2) An idle thread at destination receives message
3) Thread at destination executes code

Communication primitives 2 – DMAs (put/get)

Copy memory from place to place:



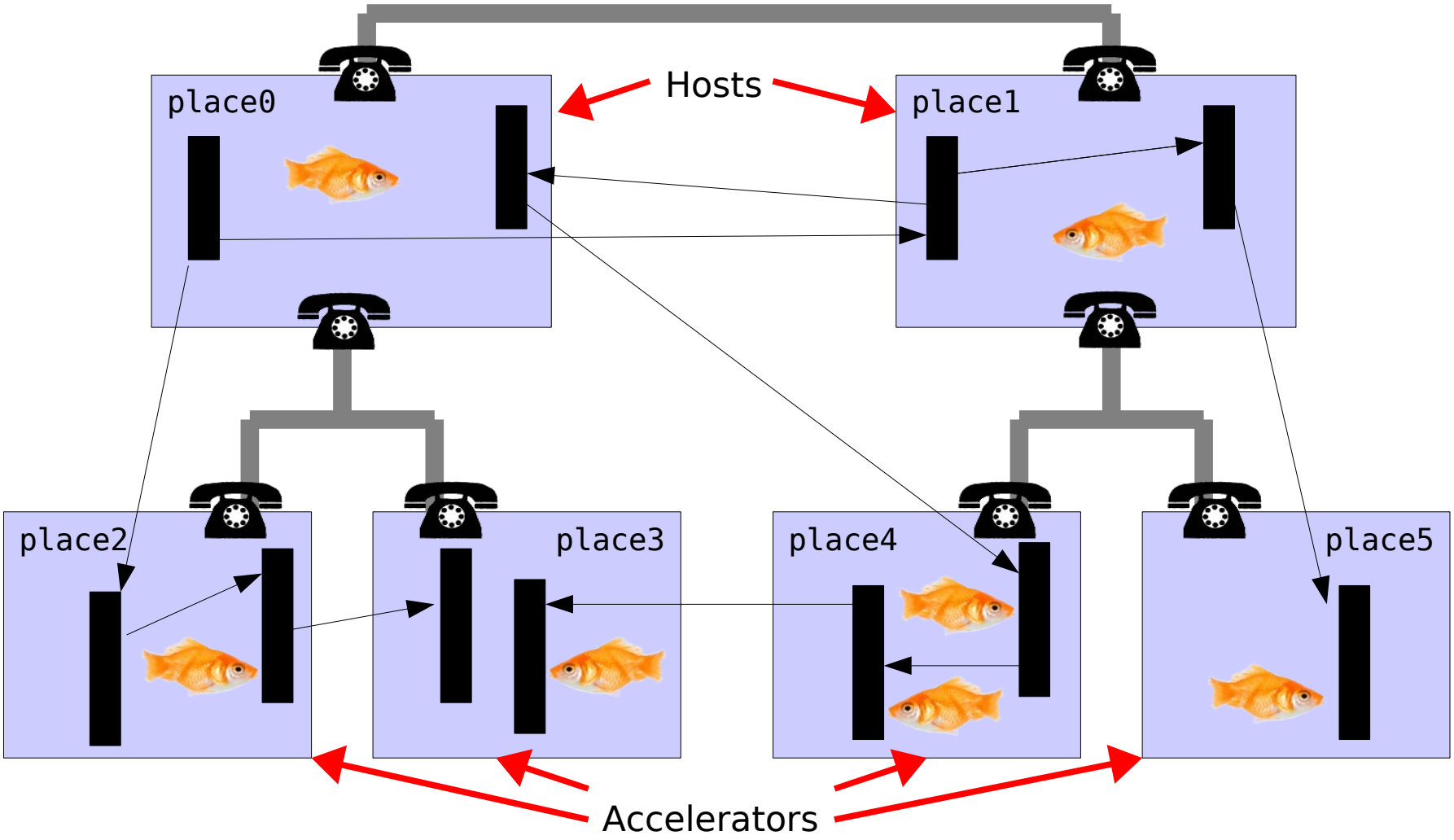
1) Thread initiates DMA (in this case, a 'put')
2) DMA asynchronously completes

Communication primitives 3 – The rest

- Fence (proceed when local ops completed)
- MPI Collectives
  - Barrier (proceed when all places are ready)
  - Gather
  - Scatter
  - Allreduce
  - Broadcast
  - etc

Accelerators

Refine notion of places into hierarchy:



Accelerator operations

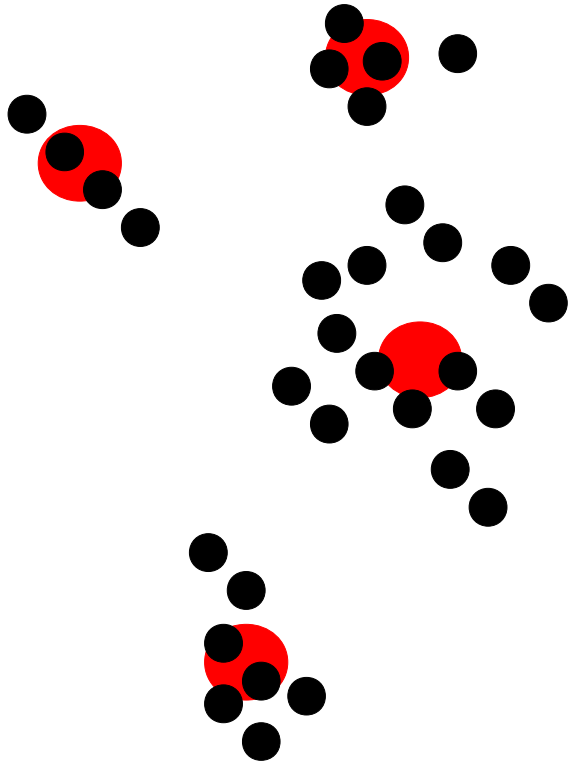
- DMA (get/put)
  - SPE: supported
  - CUDA: must be initiated from host
  - Routing (not implemented yet)
- Active message
  - SPE: supported
  - CUDA: GPU cannot send active messages
  - Routing (not implemented yet)
- Barrier / collectives
  - Host only

Case study: K-means clustering

- Implemented our vision of APGAS (as a C library).
- Does it help solve a real problem?
- To what extent is:
  - The solution scalable?
  - The code portable (to cell/GPU/CPU)?
  - The code readable/maintainable?
  - The performance portable?

K-Means: formulation of the problem

- **Given:**
  - A set of points (black dots) in *D*-dimensional space
    - (in this case $D = 2$)
  - Desired number of partitions *K*
    - (in this case $K = 4$)
- **Calculate:**
  - The K cluster locations (red blobs).
  - Minimise average distance to cluster

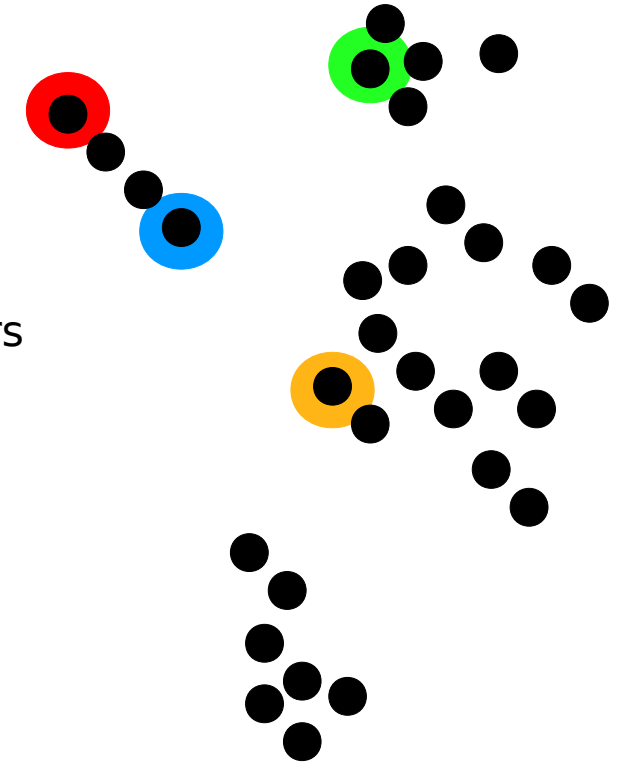


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

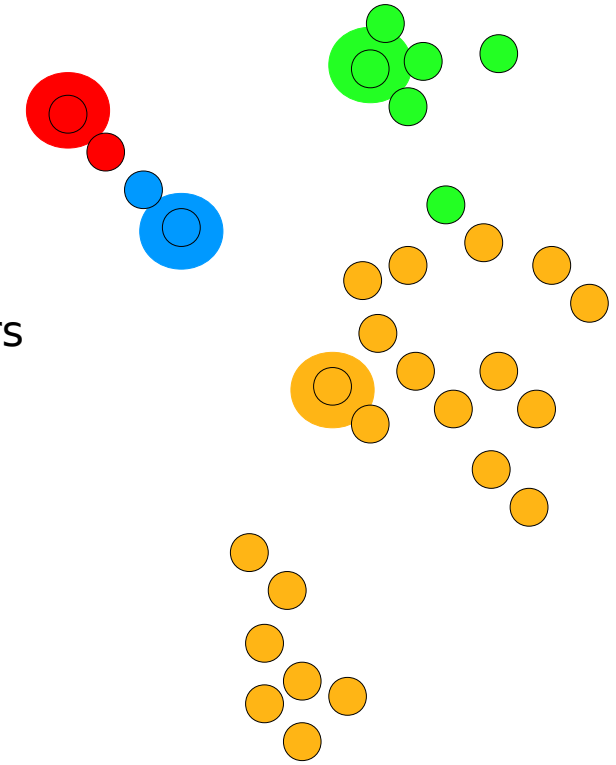


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

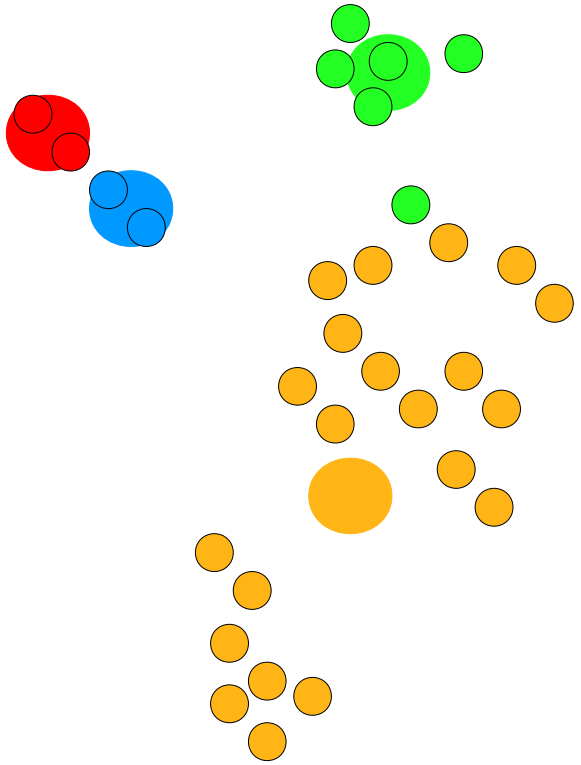


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

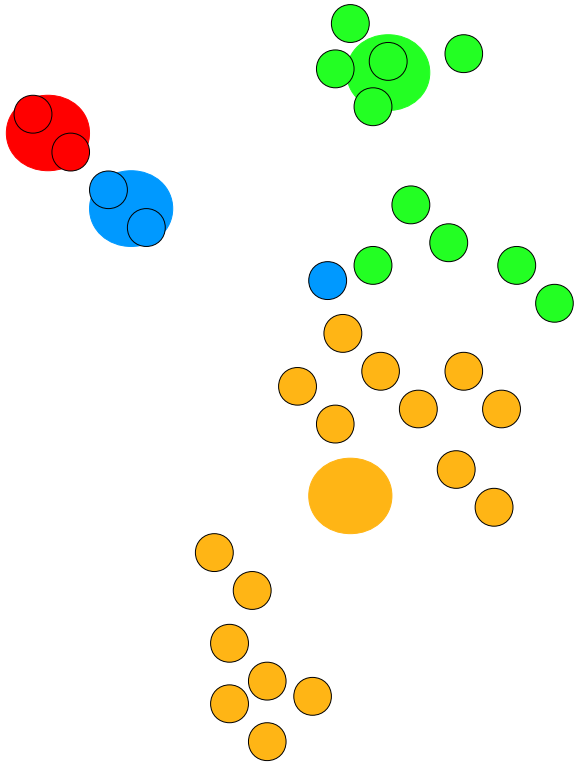


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

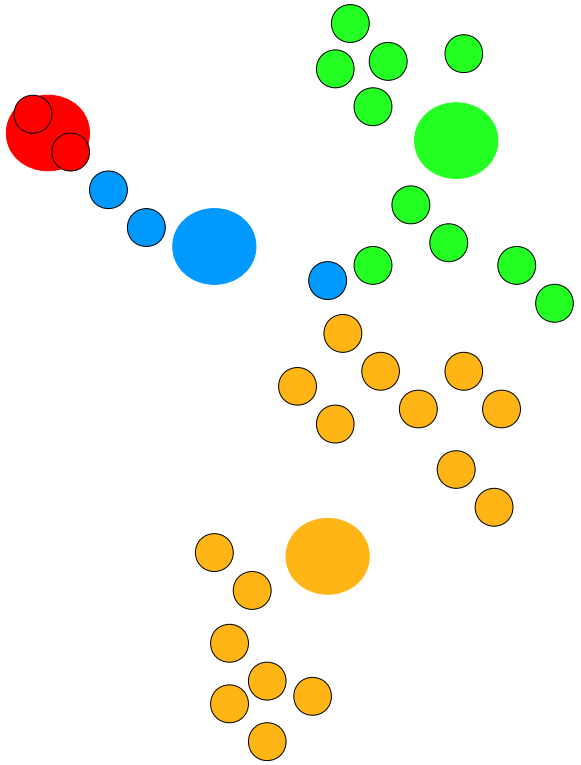


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
    classify points by nearest cluster
    average equivalence classes for improved clusters
}
```

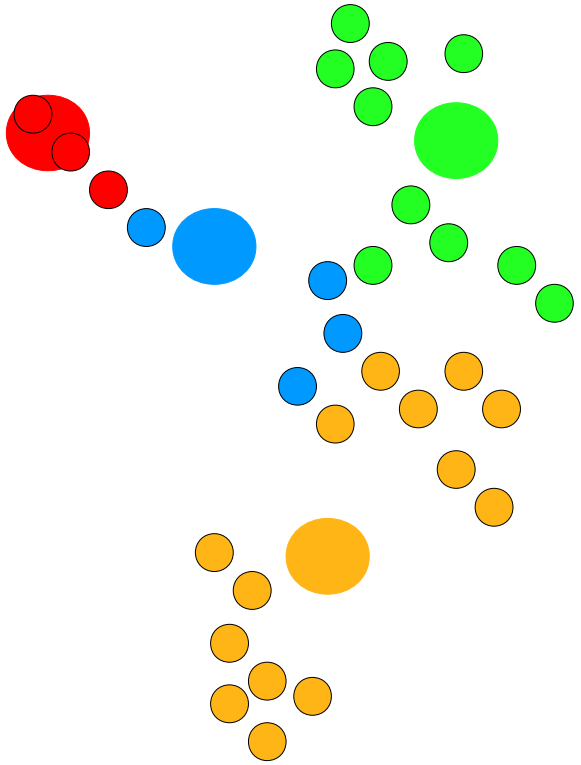


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
    classify points by nearest cluster
    average equivalence classes for improved clusters
}
```

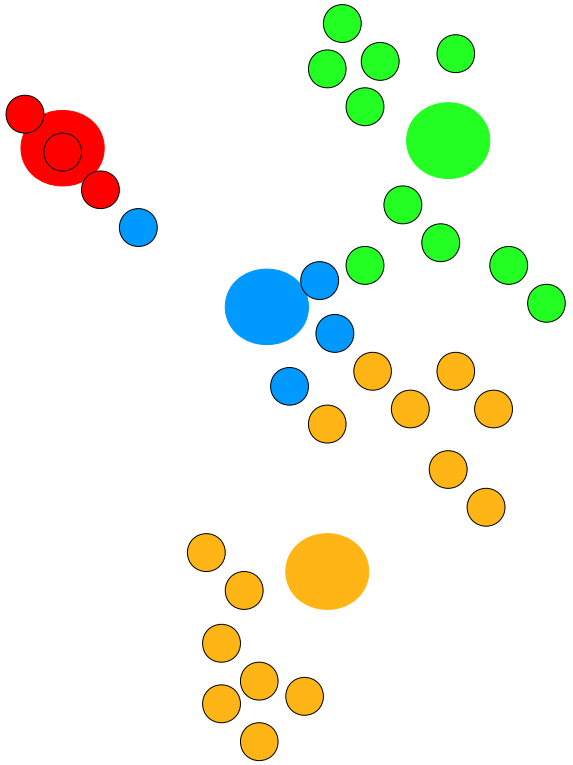


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

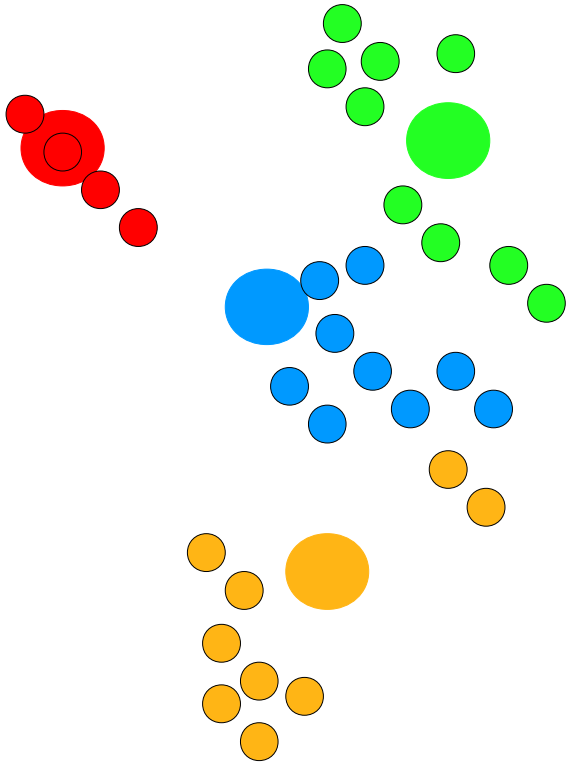


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

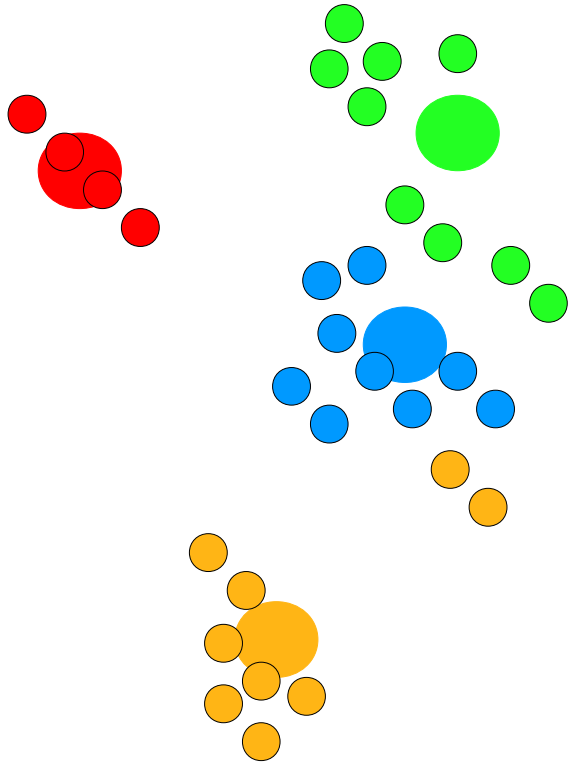


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

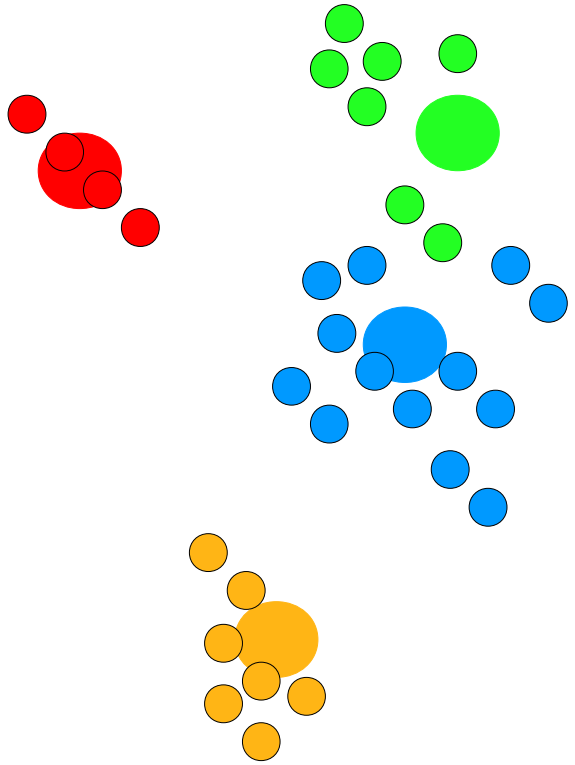


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

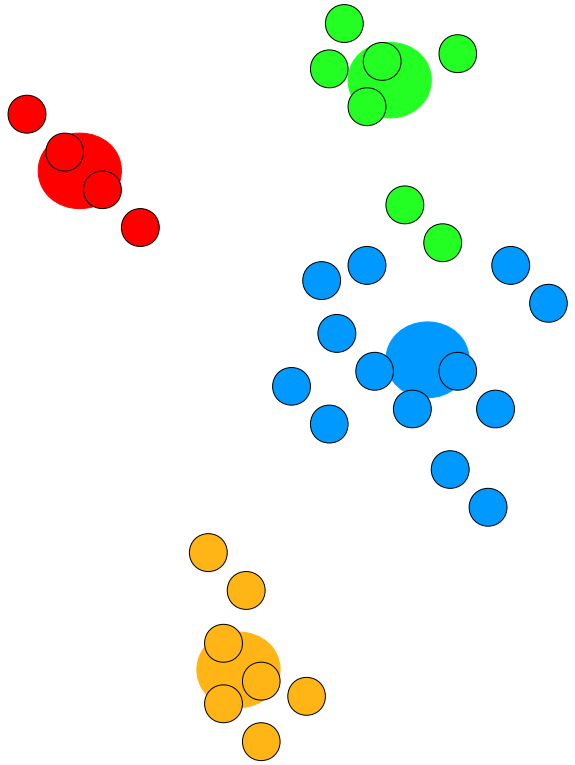


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

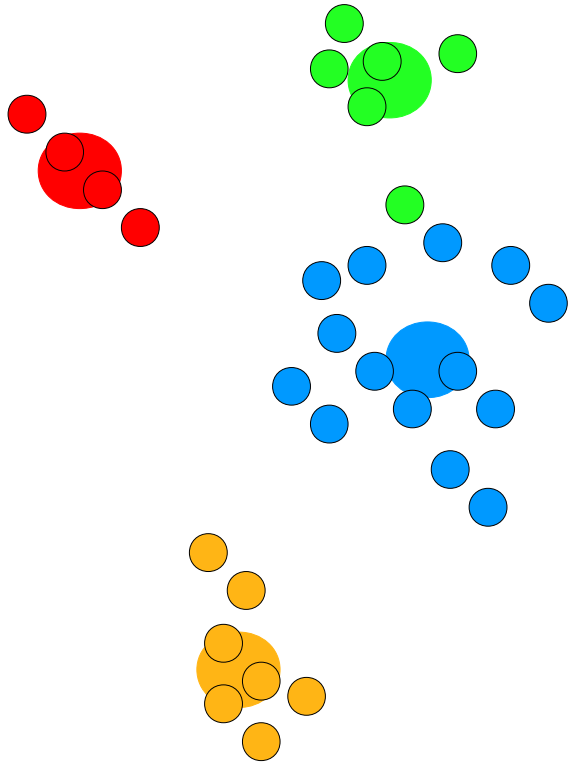


Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```



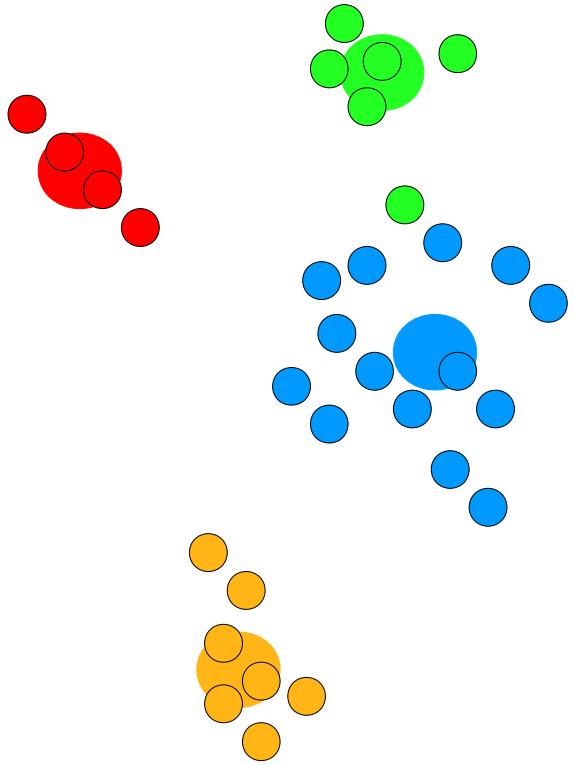
Lloyd's Algorithm

- Computes good (not best) clusters.
- Initialise with a guess for the K positions.
- Iteratively improve until we reach fixed point.
- (Can also stop early if we get bored.)

Algorithm:

```
while (!bored) {
   classify points by nearest cluster
   average equivalence classes for improved clusters
}
```

Converges quickly.



Lloyd's algorithm: code

```
void classify (const float point[N][D],
               const float cluster[K][D],
               int nearest[N])
{
  // classify points based on nearest cluster

  for (int p=0 ; p<N ; p++) { // POINTS

    int k_near = -1;
    float mindist2 = MAX_FLT;

    for (int k=0 ; k<K ; k++) { // CLUSTERS
      float dist2 = 0;
      for (int d=0 ; d<D ; d++) { // PYTHAGORAS
        diff = (point[p][d]-cluster[k][d]);
        dist2 += diff * diff;
      }
      if (dist2 <= mindist2) {
        k_near = k;
        mindist2 = dist2;
      }
    }

    nearest[p] = k_near;

  }
}
```

O (N x K x D)

```
void reaverage (const float point[N][D],
                const int nearest[N],
                float cluster[K][D],
                int   counts[K])
{
  // recalculate clusters
  // (assume 0-initialised)

  for (int p=0 ; p<N ; p++) { // POINTS

    int k_near = nearest[p];

    for (int d=0 ; d<D ; d++) // DIMENSIONS
      cluster[k_near][d] += point[k_near][d];

    counts[k_near]++;
  }
}

...

  for (int k=0 ; k<K ; k++)
    for (int d=0 ; d<D ; d++)
      cluster[k][d] /= counts[k];
```

O (N x D)

Parallel/Distributed/Accelerated Lloyds:

```
int main (...)
{
    ... here() ... hosts() ... children(h) // inspect the global hierarchy

    dma_put(child,...); // load this host's subset of points, dma to accelerators (if any)

    bcast(...); // pick initial clusters (and replicate)

    while (!bored) {

        // compute new clusters from current clusters and local points
        for (int child=0 ; child<children ; ++child)
            amsend(child,...);
        // can also do some work locally...
        classify(...) ; reaverage(...);
        fence();

        allreduce(clusters,OP_ADD);
        allreduce(counts,OP_ADD);

        // perform division
        for (int k=0 ; k<K ; k++)
            for (int d=0 ; d<D ; d++)
                cluster[k][d] /= counts[k];
    }
}
```

Different kinds of children (Cell/CUDA) will handle with their own code written in their own specialist languages...

CUDA Caveats

Handle the active message on the HOST!

- DMA to push clusters onto GPUs
- Run compute-intensive 'classify' kernel on the GPUs
- DMA to fetch 'nearest' vector (can overlap with kernel)
- Run irregular 'reaverage' kernel on the CPU (4x multicore+SMT)

Kernel Stuff (for the CUDA geeks out there):

- Cache clusters in shared memory (16kb) not constant cache (8kb)
- Outer loop (N): Strip-mine (treat GPU as massive vector unit)
- Middle loop (K): Manually unroll 20 times (#pragma didn't work!)
- Inner loop (D): Compiler automatically unrolled fully (D is a macro)
- Write kernel to support any configuration!
- Automatically configure to maximise performance WRT local constraints:
  - Maximise occupancy (SMT) to achieve peak flops
    - despite dependent instructions
  - Prefer more blocks to bigger blocks (per MP)

Lloyd's algorithm: recap

```
void classify (const float point[N][D],
               const float cluster[K][D],
               int nearest[N])
{
  // classify points based on nearest cluster

  for (int p=0 ; p<N ; p++) { // POINTS

    int k_near = -1;
    float mindist2 = MAX_FLT;

    for (int k=0 ; k<K ; k++) { // CLUSTERS
      float dist2 = 0;
      for (int d=0 ; d<D ; d++) { // PYTHAGORAS
        diff = (point[p][d]-cluster[k][d]);
        dist2 += diff * diff;
      }
      if (dist2 <= mindist2) {
        k_near = k;
        mindist2 = dist2;
      }
    }

    nearest[p] = k_near;

  }
}
```

O (N x K x D)

```
void reaverage (const float point[N][D],
                const int nearest[N],
                float cluster[K][D],
                int   counts[K])
{
  // recalculate clusters
  // (assume 0-initialised)

  for (int p=0 ; p<N ; p++) { // POINTS

    int k_near = nearest[p];

    for (int d=0 ; d<D ; d++) // DIMENSIONS
      cluster[k_near][d] += point[k_near][d];

    counts[k_near]++;
  }
}

...

  for (int k=0 ; k<K ; k++)
    for (int d=0 ; d<D ; d++)
      cluster[k][d] /= counts[k];
```

O (N x D)

Cell Caveats

Do everything on the SPEs:

- PPE is too slow
- No need to send 'nearest' vector from SPE to host
- Vector is thinner (4) than GPU (32)
- More local memory

Kernel details:

- DMA clusters to SPEs
- Fuse classify + reaverage loops to simplify code
  - Reduces amount of global memory DMA code within kernel
- No need to overlap DMA with computation!
- DMA new clusters back to host (for reduction across hosts)

- Outer loop (N): Explicit strip-mine (yuck) :(
- Middle loop (K): Manually unroll 16 times

Power5 Caveats

Loops:

- No Altivec on Power5, but can still unroll:
- Outer loop (N): unroll manually
- Middle loop (K): unroll manually

Memory Access:

- Prefetch from points array
- SMT further hides latency
- Ultimately we are bound by bandwidth
  - Required: 4 +4/UR Bytes/cycle
  - Advertised: 6 Bytes/cycle
  - In practice: 3 Bytes/cycle

K-Means Performance Experiment

Hardware	Hosts	Accelerators	Peak GF
Power5 Cluster	8 x 16 x P570	-	970
2xQS21 (PowerXCell 8i)	2 x PPE	16 x SPE	819
Tesla S1070	2 x 4 x Xeon	2 x G200	2500

Points	K	Dims
50M	400	4
25M	400	8
50M	400	4
25M	400	8

Problem configuration chosen to:

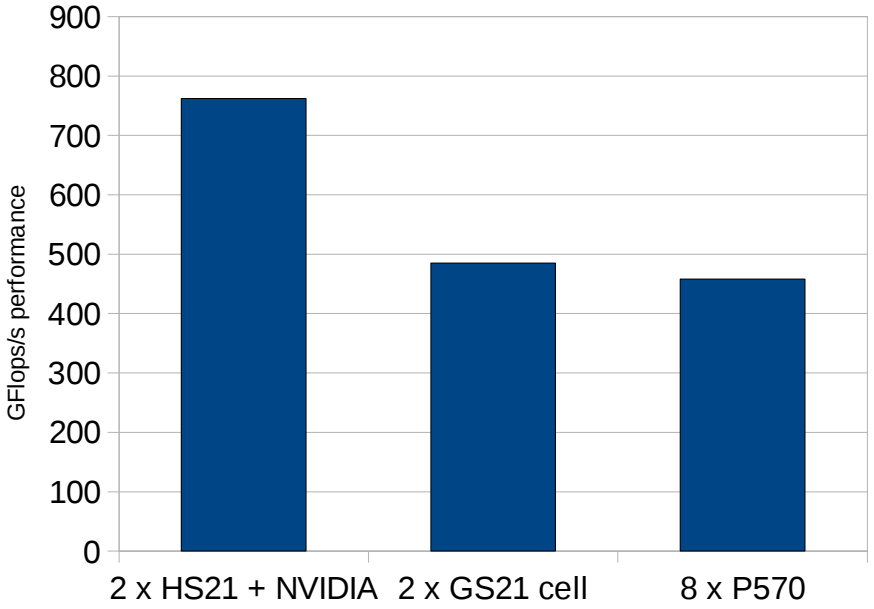
- Be reasonably realistic
- Take a reasonable amount of time
- Be appropriate for accelerators:
- Avoid clusters overflowing
  - SPE local storage (256K)
  - CUDA shared memory (16K)

We expect library to scale to more complicated kernels

For scaling K-means see:

Wu, Ren; Zhang, Bin; Hsu, Meichun "GPU-Accelerated Large Scale Analytics"

Absolute performance comparison

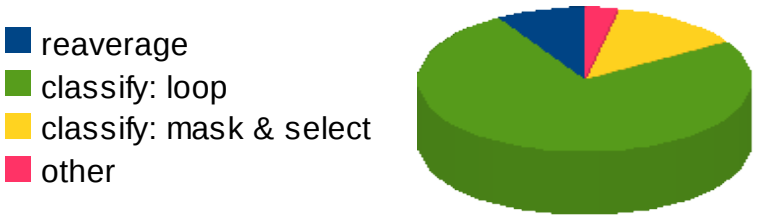


- Same problem size!
  - N=25,000,000, K=400, D=8

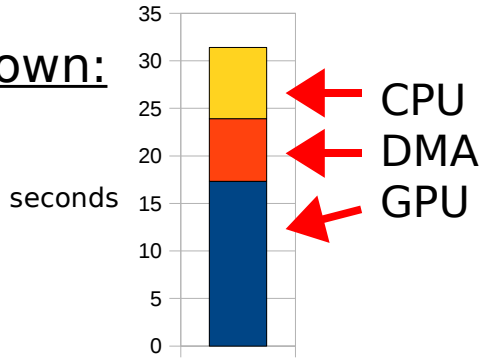
- P570: 50% peak
- QS21: 57% peak
- Tesla: 31% peak

Tesla score hurt by CPU reaverage
With DMA hiding, Tesla score is 39%

Cell breakdown:



GPU breakdown:



Conclusions

To what extent is:

- The solution scalable?
- The code portable (to cell/GPU/CPU)?
- The code readable/maintainable?
- The performance portable?

Significant fraction of peak flops obtained on all systems.

Could handle as much as we could throw at it...

1 model to rule them all (APGAS)
1 library
Same top-level glue code

Had to write kernels for each arch
- Code duplication...

Had to hand-tune each kernel
- Time-consuming...
- Error-prone...

However kernels were very similar

Similar tricks used to optimise:
- Strip-mining
- Unrolling

What now?

Some obvious stuff:

- More nodes?
- More clusters/dimensions?
- Brand new problems?

Easier things:

- Collectives for accelerators?

Harder things:

- Common language for kernels? (X10?/UPC?)
- The dream:
  - Kernel is written once.
  - Compiler can generate optimised code for any architecture.
  - Annotations to guide optimiser but no code duplication.
  - ***PLEASE*** no more manual unrolling or vectorisation!

K-means performance gating factors

Assumptions: K ~= 1% of N, D <= 10

- P5 node:
  - 16 SMT nodes
  - 64 GBytes RAM
  - 121 Gflops/s peak
- Largest problem: $N < \frac{64\,GBytes}{D * sizeof(float)} \approx 10^9$
- Expected runtime with N= 10^9: $T = \frac{total\ ops}{peak\ flop\ rate} = \frac{N \times K \times D}{121\,GFlops/s} \approx 10^6\ s/iter$

- NVIDIA C1060:
  - 240 processors
  - 4 GBytes RAM
  - 933 Gflops/s peak
- Largest problem: $N < \frac{4\,GBytes}{D * sizeof(float)} \approx 10^8$
- Expected runtime with N= 10^8: $T = \frac{total\ ops}{peak\ flop\ rate} = \frac{N \times K \times D}{933\,GFlops/s} \approx 10^4\ s/iter$

Algorithm gated by compute power, not memory.
More compute power through: **parallelism** and **accelerators**

Common problem size for all architectures

- Aim for ~ 1 second/iteration, i.e. 1 Tflop/iteration
  - Classify kernel: N * K * (3*D+1) flops/iteration
  - Reaverage kernel: (N +K) * D flops/iteration
- Gated by available physical memory:
  - 64 GB on P570 node; 16 GB on Tesla; 2 GB on QS21 PPU
  - N * K * D * sizeof (float) < 2 GBytes
- Desire to fit centroids into NVIDIA shared memory

- Common configurations:

N=50,000,000
K=400
D=4

N=25,000,000
K=400
D=8

  - **260 GFlops/iteration**
  - **Memory: 0.8 Gbytes**
  - **Centroids: 13 Kbytes**

QS21 implementation: distributing across SPEs

```
classify_reavg (const float point[N][D],
          const float cluster[K][D],
          float c2[K][D])
{
  for (int p=0; p<N; p+=44) {
    ...
    for (int k=0; k<K; k++) {
      ...
      for (int d=0; d<D; d++)
      {
        vector diff0 = (point[p][d]-cluster[k][d]);
        vector diff1 = ...;
        ...
        vector dist0 += diff * diff;
        vector dist1 += ...
      }
      ...
    }

   /* update c2 */
    c2[spu_extract(kmin_v0,0) += spu_extract(vpp[0],
0);
    ...
  }
}
```

strip-mine

vectorize

unroll

fuse reaverage

- Limited local store
  - 256 KB local store in each SPE
  - Copy-in chunks of memory, execute
  - Double buffering found to be unnecessary
- Manual SIMDization
  - SIMDization + unrolling
  - Strip-mine/unroll “p” loop
  - Unroll “k” loop
- Fused classify/reaverage
  - Reaverage executed in

P570 implementation details: `classify` kernel dominates runtime

```
classify (const float point[N][D],
          const float cluster[K][D],
          int nearest[N])
{
  for (int p=0; p<N; p+=8) {
    ...
    for (int k=0; k<K; k++) {
      ...
      for (int d=0; d<D; d++)
      {
        __dcbt (p+pointv);
        diff0 = (point[p][d]-cluster[k][d]);
        diff1 = (point[p+1][d] ... ;
        ...
        dist0 += diff * diff;
        dist1 += ...
      }
      ...
    }
    nearest[p] = idx0;
    nearest[p+1] = ...
  }
}
```

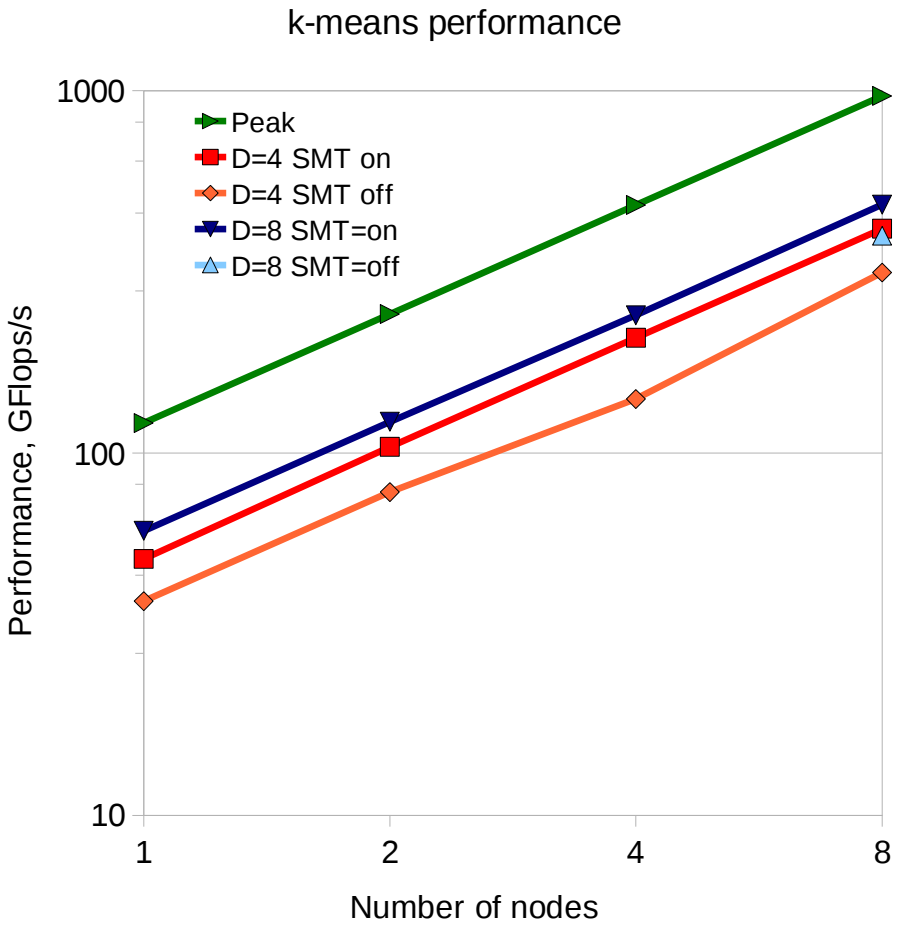
strip-mine

prefetch

unroll

- Innermost loop: 3 flops
  - 1 fsub, 1 fma
  - Only 75% of peak flop rate!
- Unrolling hides latency
  - Only 32 FP registers: limits amount of unrolling
  - SMT helps to further hide latency
  - No Altivec on P5
- Best loop to unroll is “p” loop
  - Compiler cannot stripmine & unroll “p” loop without user

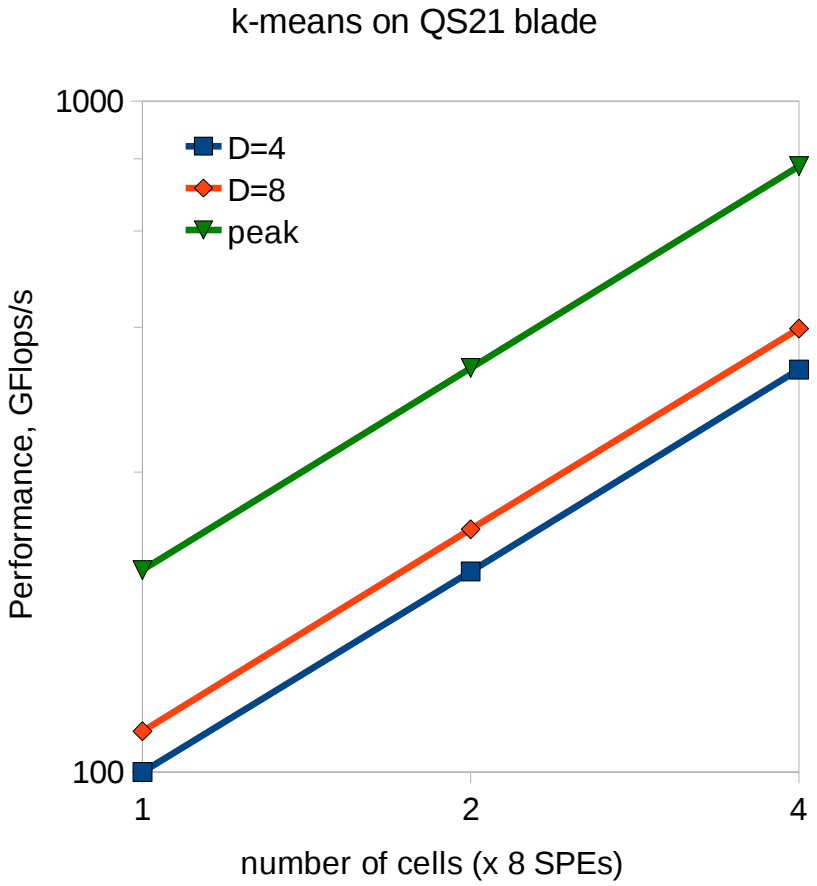
P570 cluster implementation: performance



- **Measured** Performance
  - D=4: 32% w/o SMT, **43%** SMT
  - D=8: 41% w/o SMT, **50%** SMT
- No scaling issues
- No superlinear bump

- **Expected** performance:
  - Peak * ¾ (flops)
  - Peak * 3/4.5 Bytes/cycle
    - Unroll factor = 8
  - Expect ¾ * 3/4.5 =

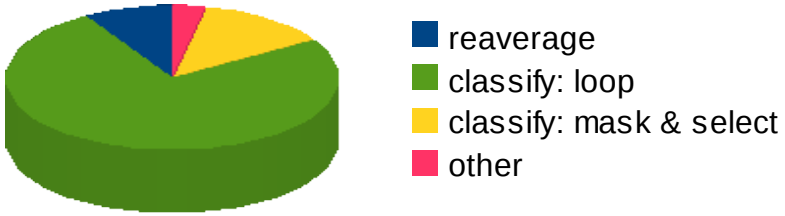
QS21 implementation: performance



- **57 %** peak when D=8
- **49 %** peak when D=4

- "Classify" close to ¾ peak
- Performance lost in:
  - Loop overheads
  - Non-vectorizable code

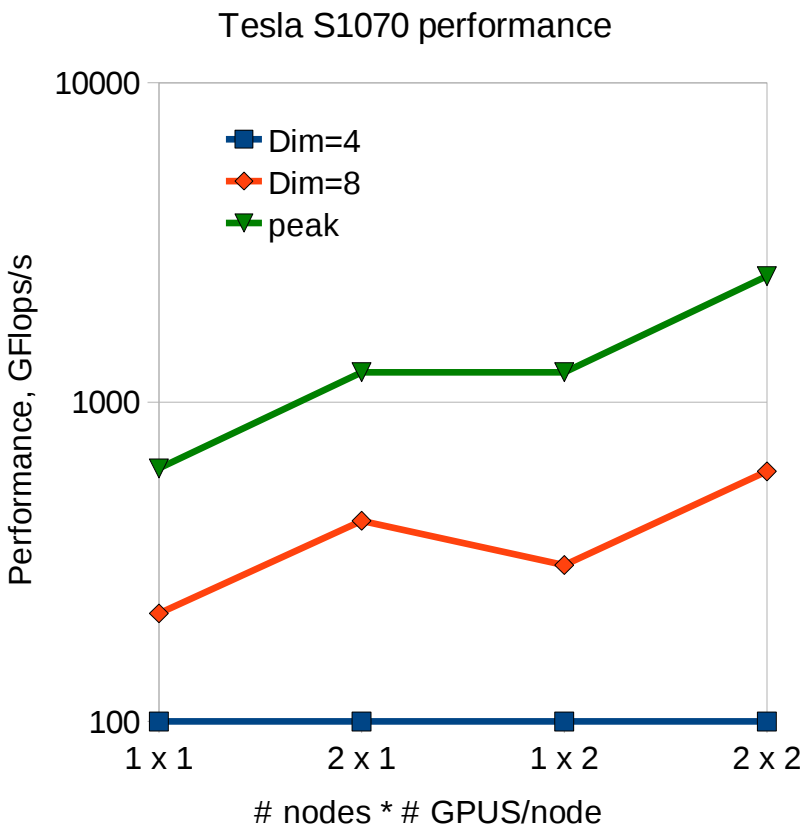
Iteration breakdown for D=4



Tesla S1070 implementation: Issues

- Used “logical layer” of APGAS
  - Classify kernel runs in GPU
  - Reaverage runs in CPU
- Classify kernel:
  - Outermost “p” loop parallelized by CUDA compiler (not manually unrolled)
  - “k” loop unrolled by hand up to 25 times
  - “d” loop unrolled by compiler
- Reaverage kernel:
  - Significan contributor to total execution time!
  - Has to run on CPU: not enough parallelism for GPU or else reduction not worth the cost
  - Performance improvement possible if using hyperthreads on CPU.
- Communication between CPU and GPU a significant

NVIDIA S1070 cluster: performance



- Running at **25%** to **33%** of Tesla real peak performance:
  - Classify kernel runs at ¾ peak performance!
  - Communication time
    - Fraction increases when using both GPUs on a node
  - Significant time spent in reaverage

Did APGAS help?

- APGAS “glue” is the same on all architectures
  - Collective communication primitives identical
  - Hide differences in active message handlers
- APGAS DMA used to transfer data to/from accelerators
  - Future: Intra-accelerator collectives?
- Everything is asynchronous
  - Overlap host / accelerator / DMA
  - Fence to get consistency