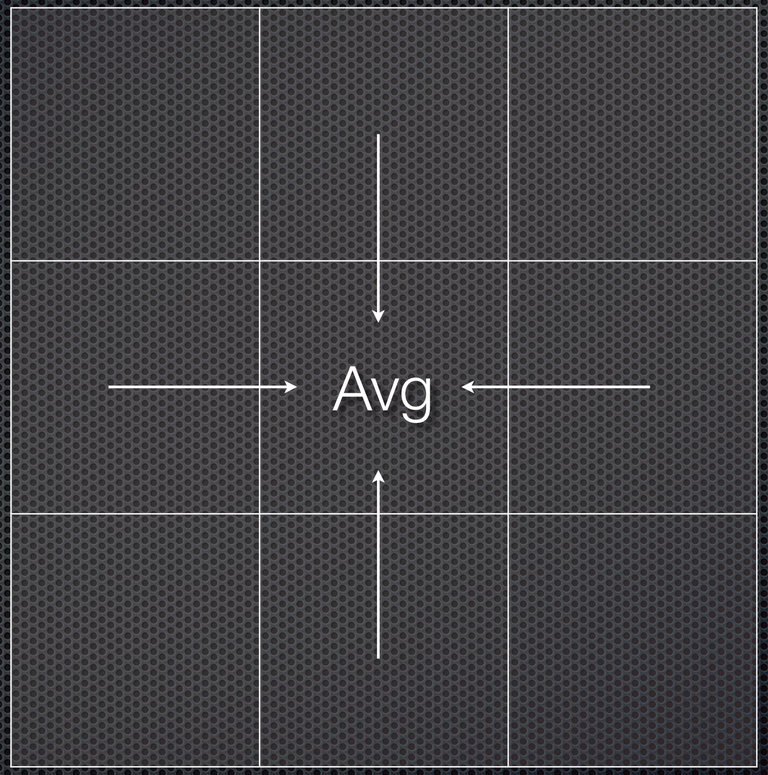
Shared Memory Parallel Programming

Is it really easier?

Based on Chapter 8 of Greg Pfister's
text, "In Search of Clusters"

4-point Stencil



```
close_enough = epsilon;
repeat
   max_change = 0;
   for y = 2 to N-1
      for x = 2 to N-1
         old_value = v[x,y];
         // replace each value with avg of neighbors
         v[x,y] = (v[x-1,y]+v[x+1,y]+v[x,y-1]+v[x,y+1])/4;
         // keep max_change at largest absolute change seen
         max_change = max(max_change, abs(old_value-v[x,y]));
    end for
   end for
until max_change < close_enough
```

All we do is...

- Use maximum parallelism -- assume one node per array element
- Make the array and indexes *shared*
- Everything else stays *private*
- Change the *for* statements to *forall*

```
close_enough = epsilon;
shared v[], x, y, max_change;
private default;
repeat
   max_change = 0;
   forall y = 2 to N-1
      forall x = 2 to N-1
         old_value = v[x,y];
         // replace each value with avg of neighbors
         v[x,y] = (v[x-1,y]+v[x+1,y]+v[x,y-1]+v[x,y+1])/4;
         // keep max_change at largest absolute change seen
         max_change = max(max_change, abs(old_value-v[x,y]));
    end forall
   end forall
until max_change < close_enough
```

That was easy...

- *forall* just magically keeps everything straight
- But it doesn’t work. At least not reliably.
- Why?

That was easy...

- *forall* just magically keeps everything straight
- But it doesn't work. At least not reliably.
- Why?
- Because there is a race on access to max_change, which can cause termination before every max_change < epsilon
- Need to add a lock

```
close_enough = epsilon;
lock max_change_lock;
shared v[], x, y, max_change;
private default;
repeat
   max_change = 0;
   forall y = 2 to N-1
      row_max = 0;
      forall x = 2 to N-1
         old_value = v[x,y];
         // replace each value with avg of neighbors
         v[x,y] = (v[x-1,y]+v[x+1,y]+v[x,y-1]+v[x,y+1])/4;
         // keep max_change at largest absolute change seen
         acquire(max_change_lock);
         max_change = max(max_change, abs(old_value-v[x,y]));
         release(max_change_lock);
     end forall
   end forall
until max_change < close_enough
```

That wasn’t so hard...

- But it doesn’t actually improve performance
- Why not?

That wasn’t so hard...

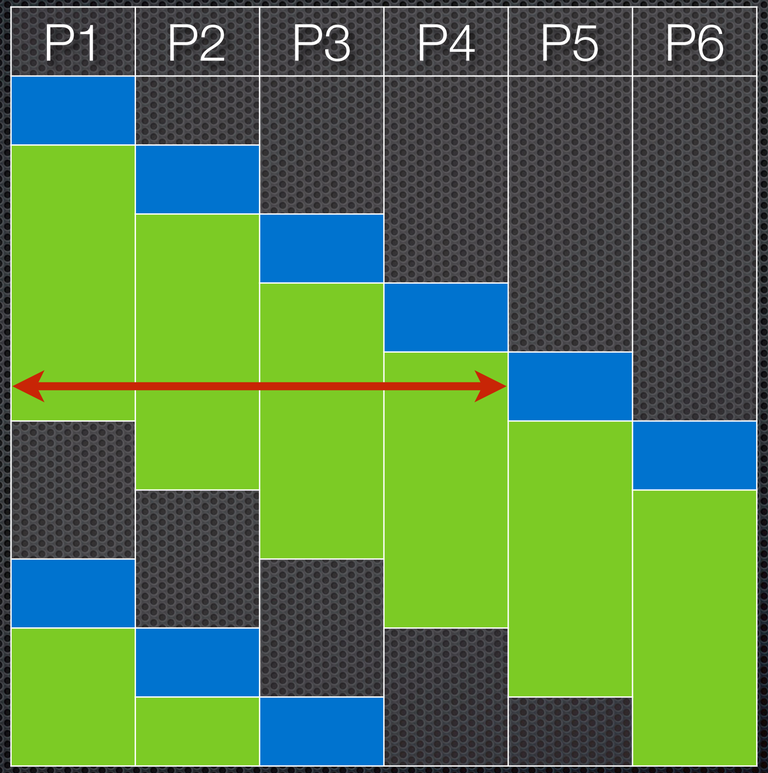
- But it doesn’t actually improve performance
- Why not?
- Because there is very little work to be done outside of the lock
- The lock itself is slow -- it has to go all the way out through the memory hierarchy as an atomic transaction
- Serialization

Serialization

Parallel Work

Serial Work

Waiting



Aggregation

- Need to do more work in parallel sections
- Create larger chunks by reducing parallelism
- Make inner loop serial
- Only check max_change for each row

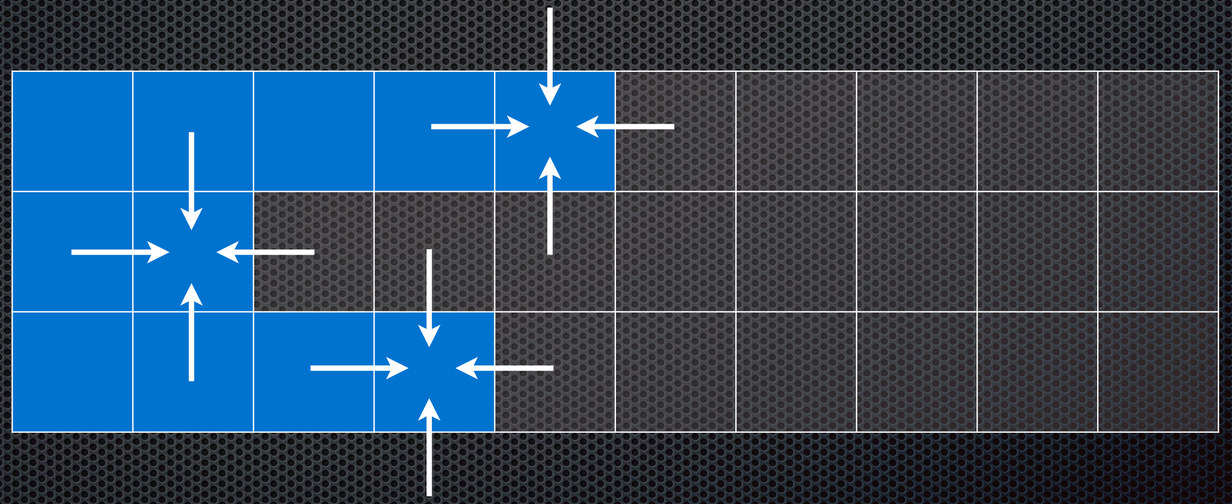
```
close_enough = epsilon;
lock max_change_lock;
shared v[], y, max_change;
private default;
repeat
   max_change = 0;
   forall y = 2 to N-1
      row_max = 0;
      for x = 2 to N-1
         old_value = v[x,y];
         // replace each value with avg of neighbors
         v[x,y] = (v[x-1,y]+v[x+1,y]+v[x,y-1]+v[x,y+1])/4;
         row_max = max(row_max, abs(old_value-v[x,y]));
      end for
      // keep max_change at largest absolute change seen
      acquire(max_change_lock);
      max_change = max(max_change, row_max));
      release(max_change_lock);
   end forall
until max_change < close_enough
```

Good performance, but...

- Results vary from one run to the next
- Why?

Good performance, but...

- Results vary from one run to the next
- Why? Because serial processor speed varies.



People don’t like it when computers give random answers, even if they are all technically correct

Obtaining consistency

- Lock all three rows
- Process entire row
- Release rows
- Processors two rows away run in parallel
- What’s the danger in this?

We can lock rows to avoid the race this way:

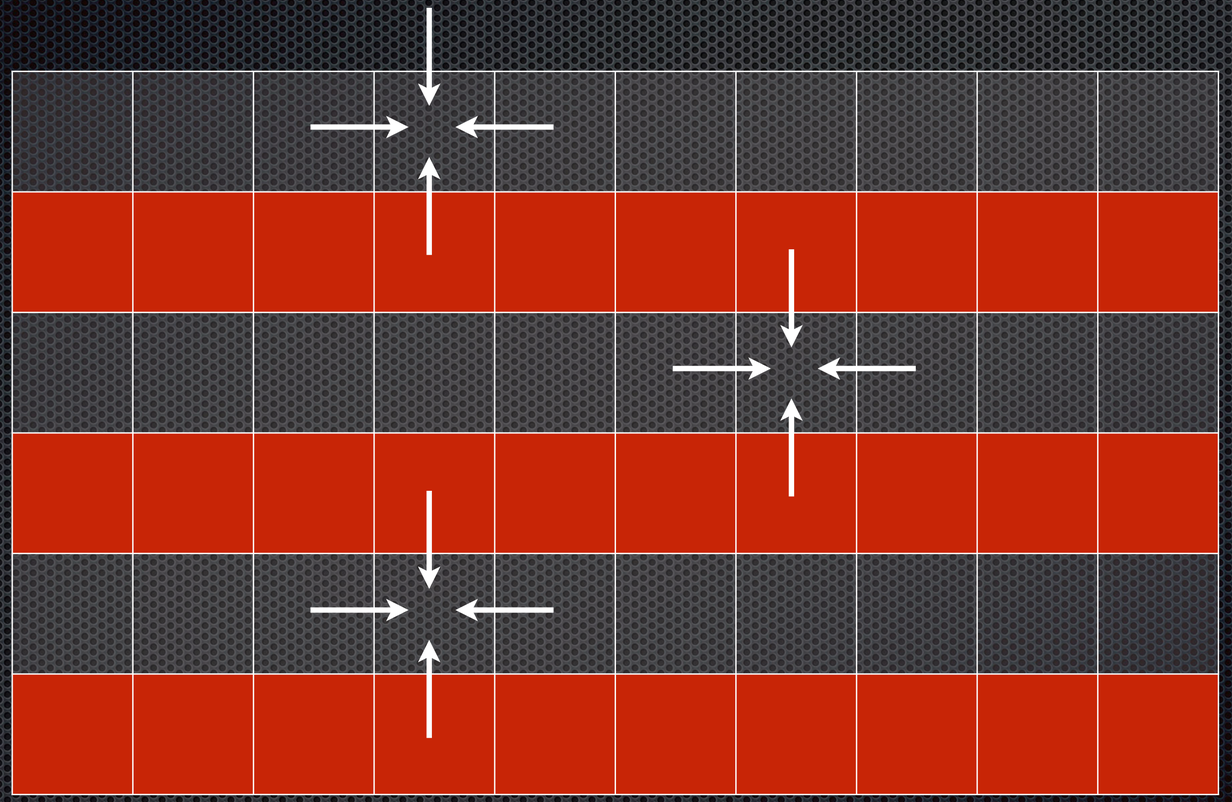
```
acquire(row_lock[y-1]);
acquire(row_lock[y]);
acquire(row_lock[y+1]);
```

But this slight variation will deadlock:

```
acquire(row_lock[y]);
acquire(row_lock[y+1]);
acquire(row_lock[y-1]);
```

Why? It can result in a cycle.
Note that if the problem wrapped at the boundaries , it could deadlock even with the version above -- cycle is created at top and bottom rows.

Alternation

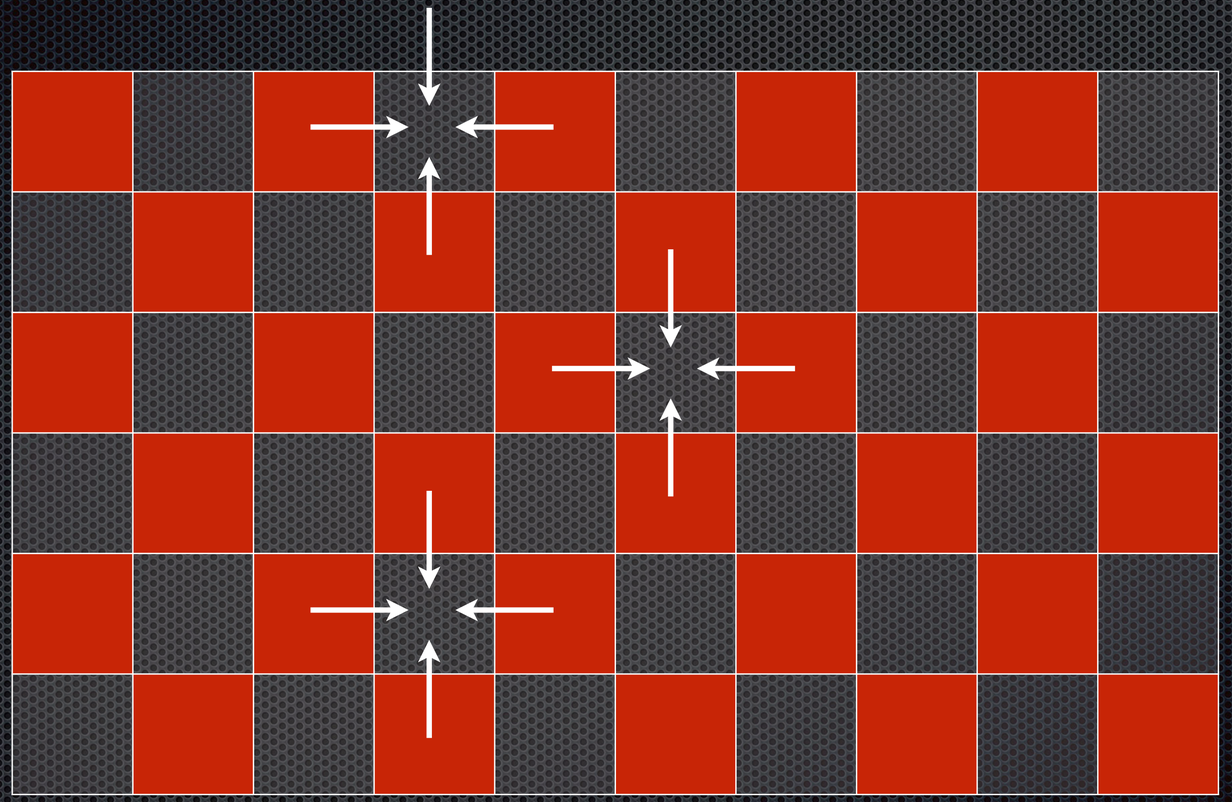


Alternate between processing red and black rows

Something old...

- Something new (something borrowed from Big Blue)
- Cells see one old, two half-old, one new value
- How about we arrange it instead like this...

Checkerboarding



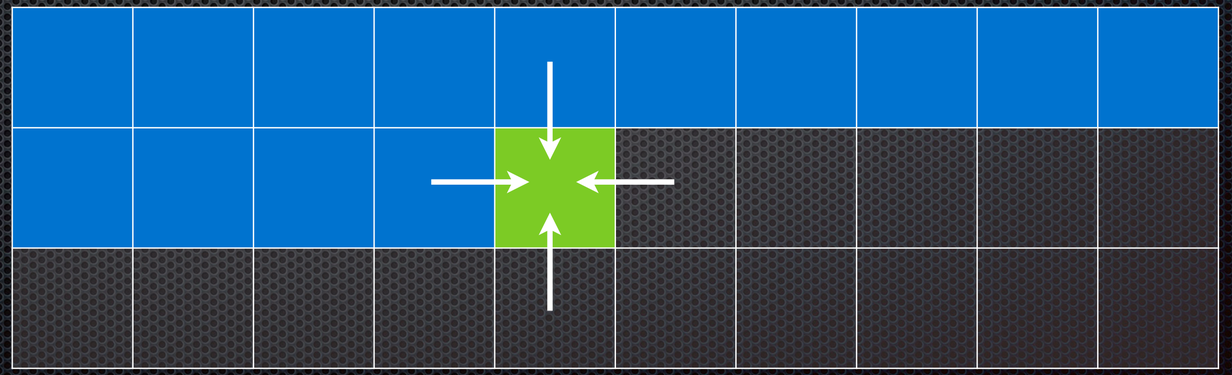
Alternate between processing red and black squares.
Everyone gets an old value from all 4 neighbors.

Consistency at last...

- But, it’s not the same answer as the serial version
- Why?

Consistency at last...

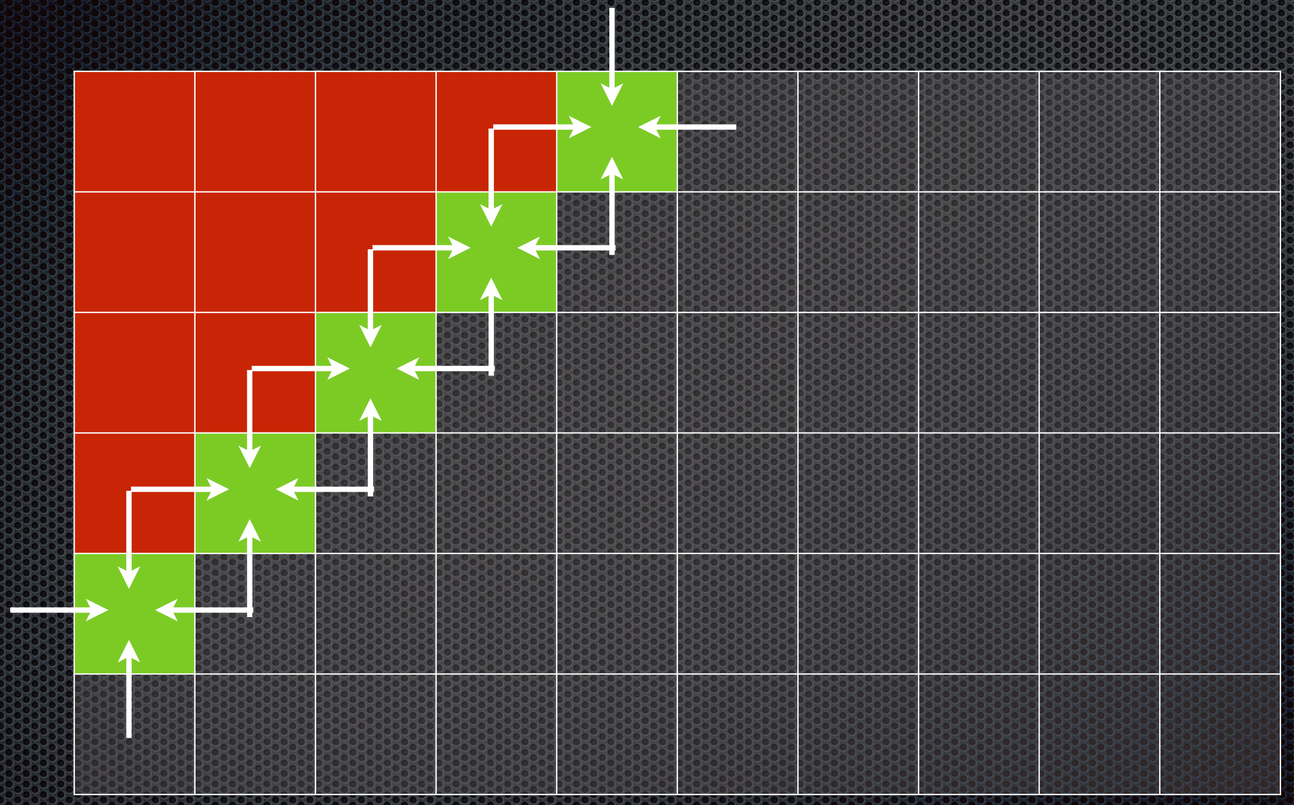
- But, it’s not the same answer as the serial version
- Why?
- Because the serial version used two new and two old values in each position



Parallel Programming Classic Error

- Solve a different problem that’s easier to parallelize
- Serial is Gauss-Seidel
- Parallel is Gauss-Jacobi
- Give different, correct results
- Sometimes they don’t converge, but for different inputs

Wavefront



Processing diagonals gives Gauss-Seidel in parallel!
But how much parallelism do we get?

It Varies…

- Very little at start and end, but a lot in the middle
- And we still have to lock max_change between small chunks of processing
- Better aggregate again -- let processors compute local maxima over a square tile, and we'll serially merge a bunch of them (say a few hundred)
- Now it runs pretty fast on a bunch of processors, compared to running on one

But...

- It’s still slower than the serial version
- Why?

But...

- It’s still slower than the serial version
- Why?
- Cache locality!
- Wavefronting is terrible for cache locality, so we lose a factor of 100
- Coherence traffic also remains high
- It’s a more complex algorithm

Blocking

- Aggregate in a cache-sensitive way
- Process a rectangular chunk of the array in each processor, with dimensions that maximize cache locality, and reduce coherence traffic
- But, this is microarchitecture-dependent
- Want to hide it in the compiler -- can we?
- Maybe, but may take more language extensions or programming effort

And the Level of Parallelism Still Varies

- Load balancing
- Set up work queue, and pass out equal segments of successive diagonals to processors
- Now we get better performance than serial, same answer, and good utilization

Except that...

- The work queue is shared (i.e., locked) data structure
- Here comes serialization again
- Need to have processors do a lot of work between queue accesses
- One processor manages the queue (bottleneck)
- Neglected to note that global queues are actually everywhere -- *forall*

With more work...

- We can solve these problems too
- But what do you think?
- Is shared memory really easier than other parallel programming models?
- Or just a case of Pavlov’s programmers?

Why do we do it?

- Because it’s easy to build a shared memory processor -- just put multiple CPUs on a bus
- And the hardware people are happy to let the software people then spend countless hours playing Whack-A-Mole to try to get performance out of it
- It’s simple enough in concept to be grasped by pointy-haired bosses

What About Message Passing?

- Ah, the joys of network-topology dependent algorithms
- Topomania

There is no Universal Parallel Model

- Every parallel architecture defines a new algorithmic model
- Consider a histogram operation on minor variants of a vector-array processor
- Portability is nonexistent
- Can’t grow code base

More Shared Memory

Beyond the bus

Parallelism Approaches

- Scale up: ILP (super pipeline, superscalar), threading, clock, short vector
- Scale out: Multiple cores, multiple multi-core nodes
- Vector/Streaming: GPU, tensor units, encryption units
- Heterogeneous: Combinations of approaches

Parallel Programming Abstractions

- Shared memory: OpenMP, pthreads, Java threads, etc. (one address space)
- Distributed memory: MPI, RPC (separate address spaces)
- Vector: Parallel operators, CUDA, map-reduce, etc. (one address space)

Shared Memory Scaling

- Shared memory is easy for a small number of CPUs
  - Place them on a bus and use MESIF or similar protocol
- Bus quickly saturates, performance decreases for more than about 8 processors
- How to extend shared memory to greater numbers?
  - Assumes this abstraction is a good idea for scaling out

Directory-Based Coherence

- Full Map
  - Each block of DRAM (cache line unit) is extended
  - Extension bits correspond to processors
  - A 1 indicates the processor’s cache has a copy, a 0 means it doesn’t
  - Usually some fixed number of bits (processors) per block

Directory-Based Coherence

- Partial Map
  - Each block of DRAM (cache line unit) is extended
  - Extension is groups of bits, each representing a processor number
  - Processor numbers are entered in the list when they request a copy
  - Usually some fixed number of sharing entries (e.g., 4 or 8)
  - When sharing exceeds the available list length, default to broadcast

Directory-Based Coherence

- Chained Map
  - Each block of DRAM (cache line unit) is extended
  - Extension is a processor number for the “owner” (first request)
  - Each cache line is extended with a next processor number or null, and the owner processor number
  - Changes are sent to the owner, then forwarded down the chain (linked list)
  - Protocol for updating the chain for a new owner, or deletions from list

Breakout Discussion

What are the pros and cons of each directory paradigm?
(full map, partial map, chained)

Discuss in groups, send me an email with your names and notes

Stanford DASH
(Lenoski, ISCA 90)

- Local cluster of cache coherent, bus snooping processors
- Cluster caches are shared via cache to cache transfers
- Connect via directory and extra cache to network
- Directory keeps track of sharing, manages messages for coherence

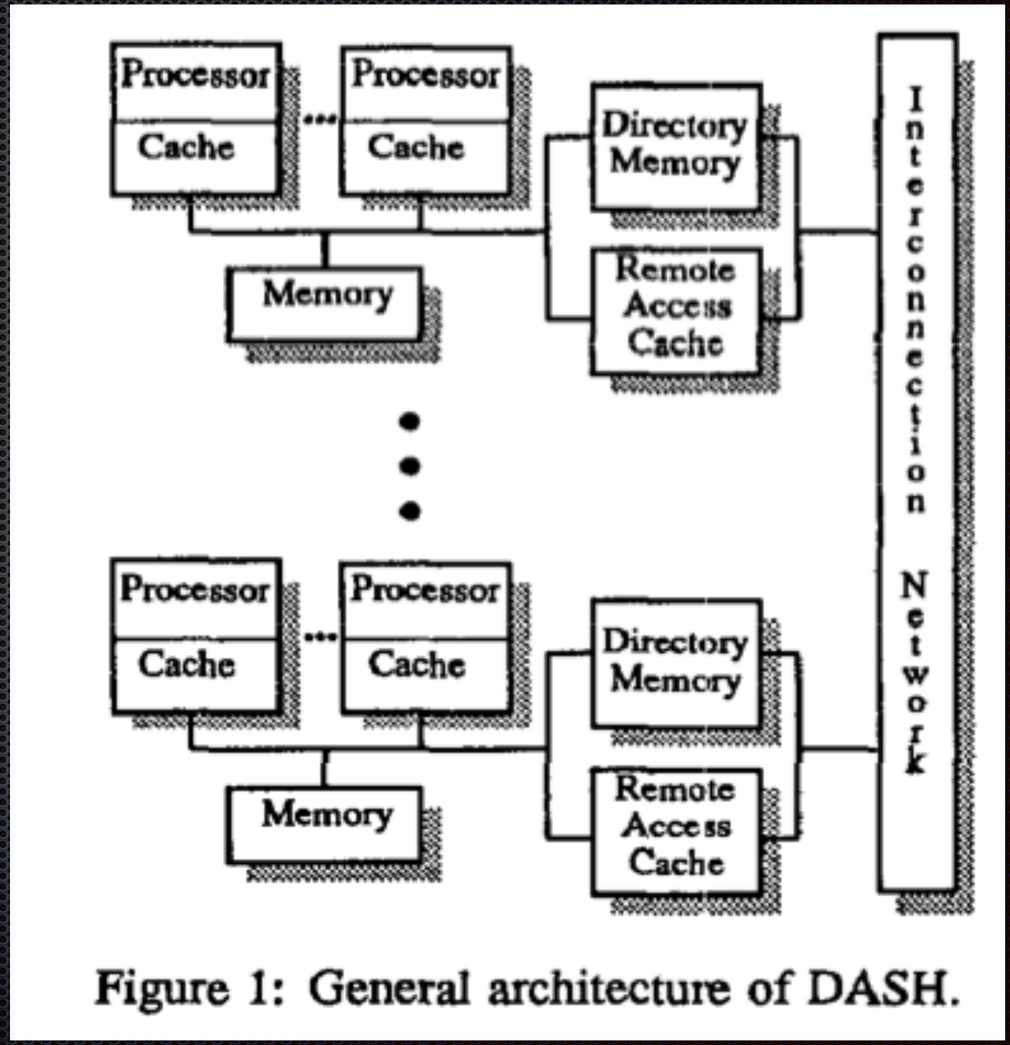


Figure 1: General architecture of DASH.

Node Architecture

- Local main memory and disk, level 1 and 2 caches
- Separate mesh-connected networks for memory request and replies (topology not important)

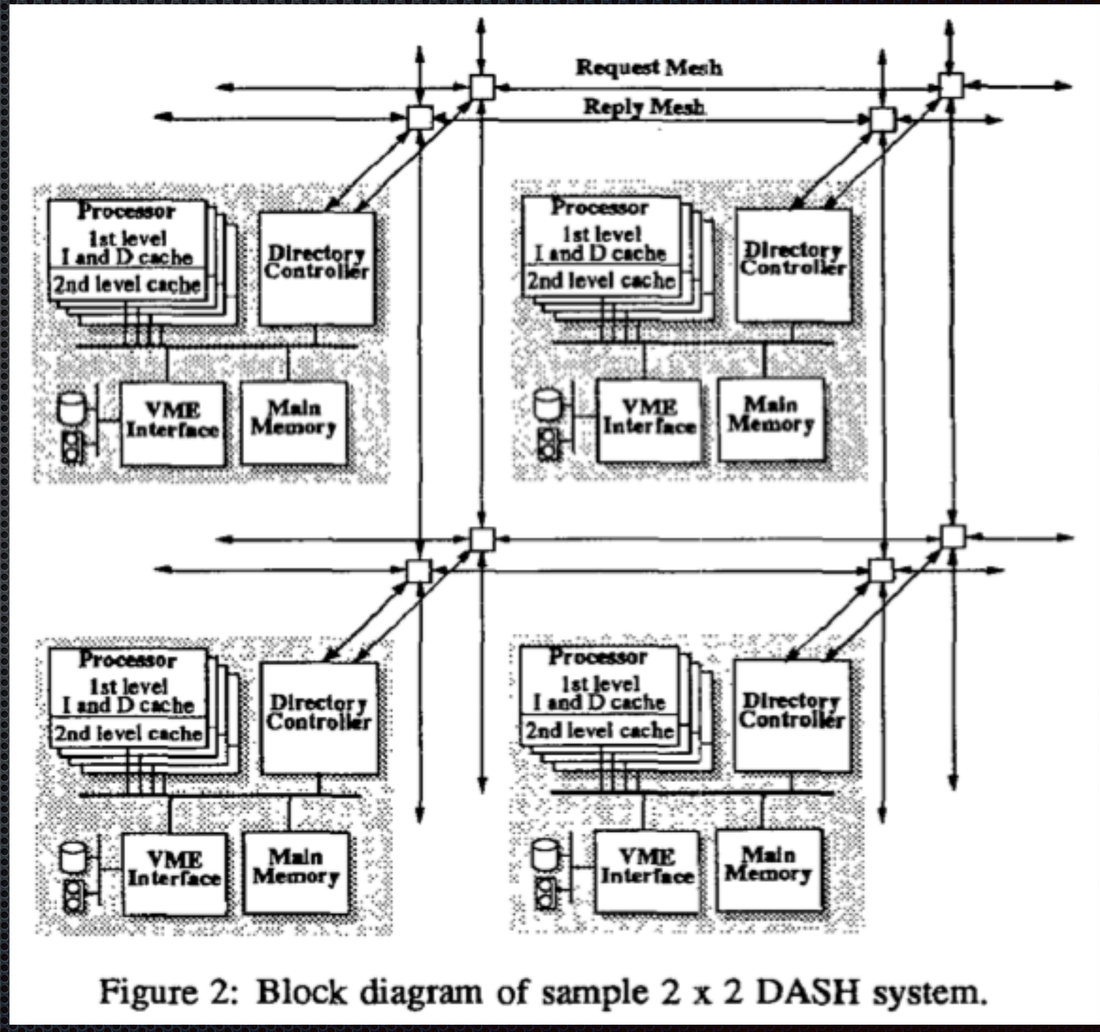


Figure 2: Block diagram of sample 2 x 2 DASH system.

Directory Board

- Directory controller -- sends requests out
- Pseudo-CPU -- handles outside requests
- Reply controller -- receives reply to request and passes to local node

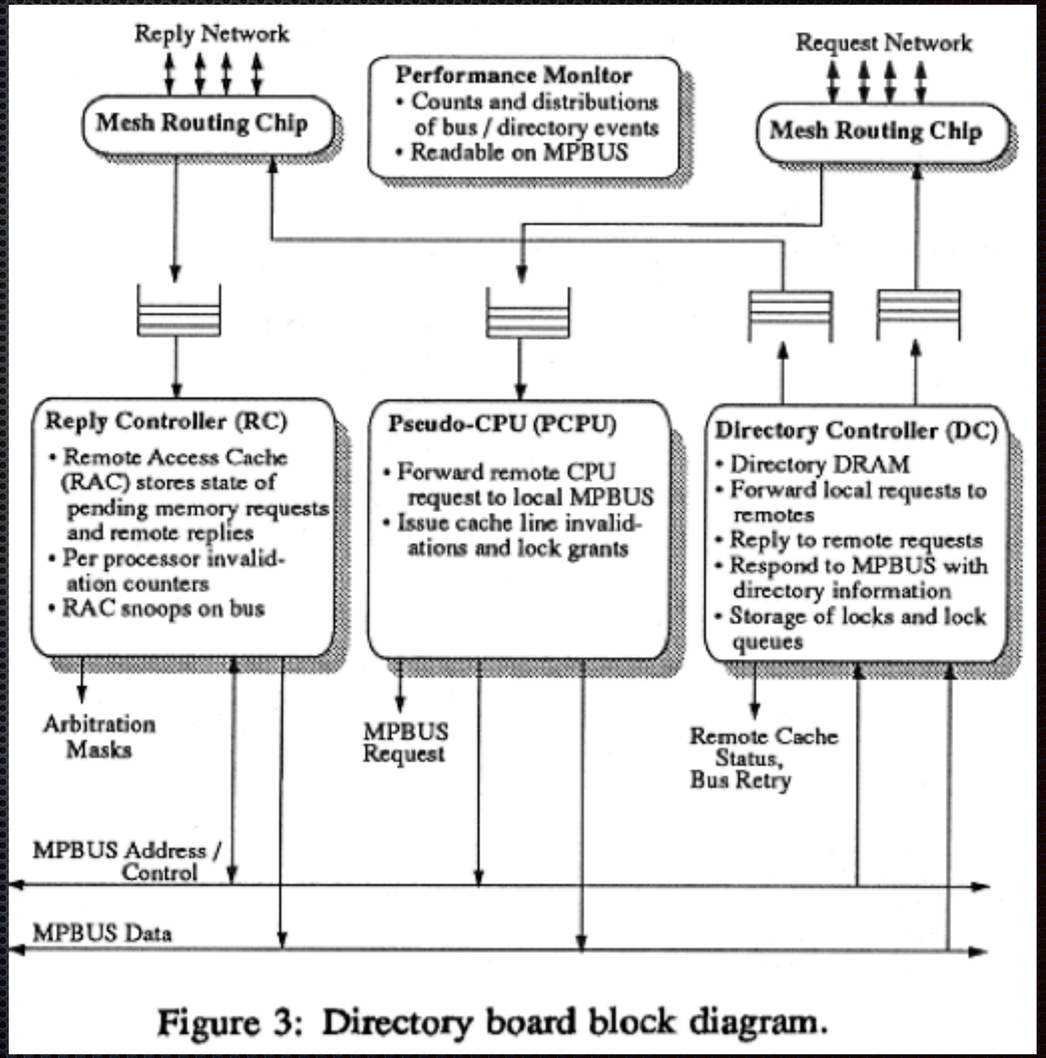


Figure 3: Directory board block diagram.

Release Consistency

- Weaker than sequential consistency
- Acquire and release operations
- An acquire must complete before subsequent reads and writes (everybody sees the acquire consistently)
- Non-owners can observe changes to protected variables, assuming they are aware of the critical section
- Before release all write (and read) operations must complete -- many remote operations will already be done

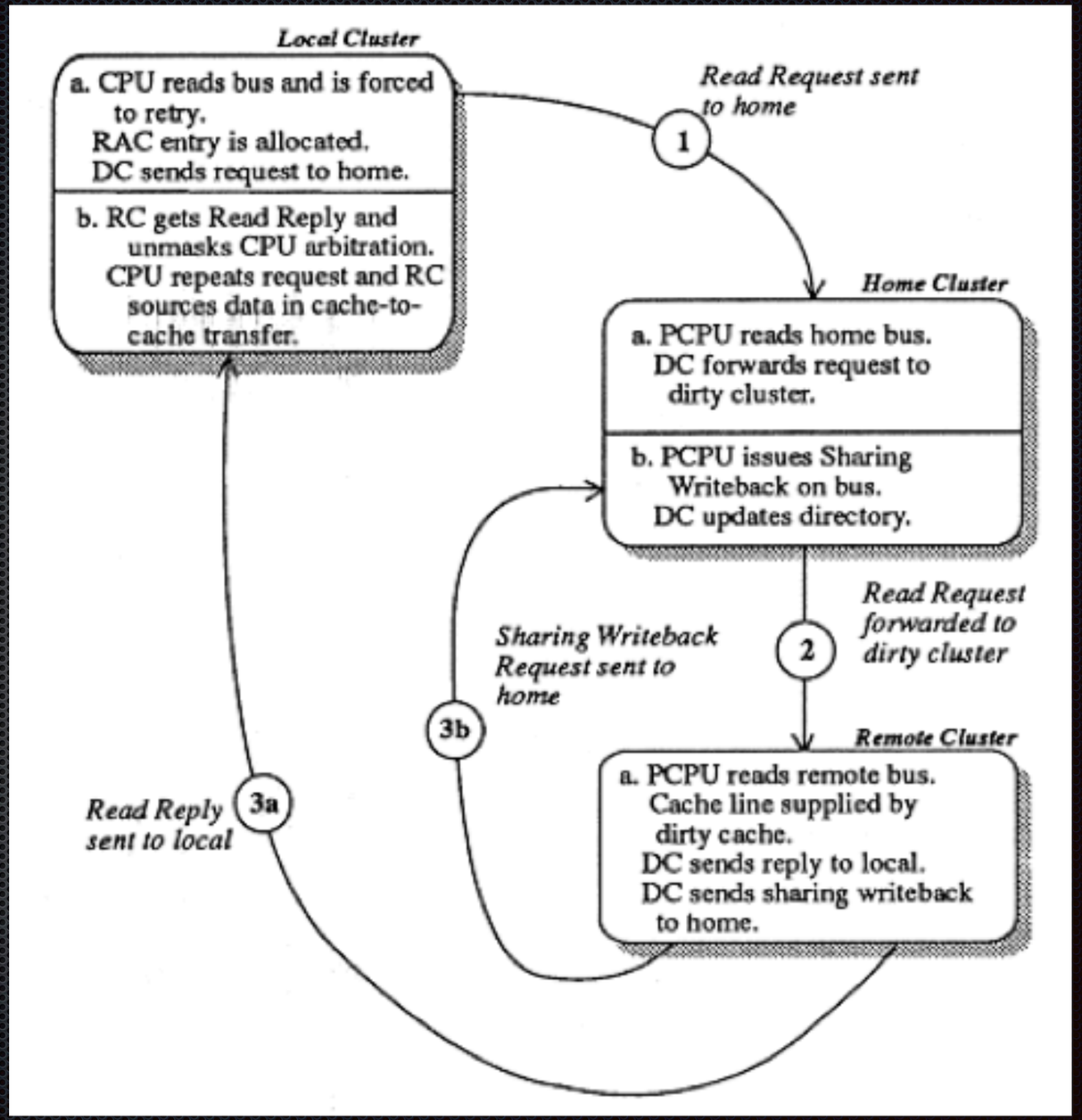
Types of Directory Clusters

- Local: Cluster from which a request is issued
- Home: Cluster holding the main memory location in the global address space
- Remote: Any other cluster

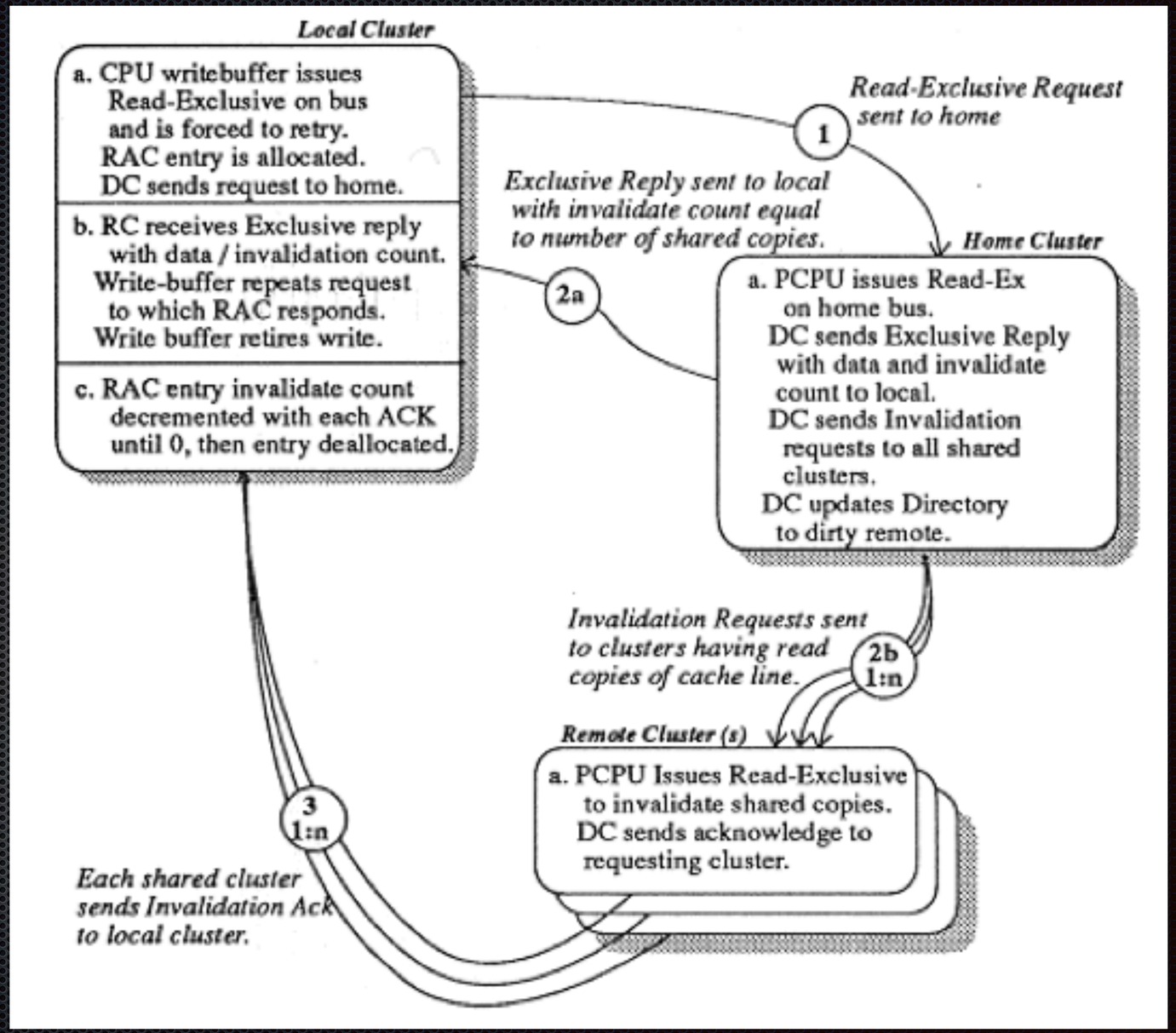
Remote Memory States

- Uncached remote -- not remotely cached
  - Home is owner
- Shared remote -- clean, cached remote
  - Home is owner
- Dirty remote -- one remote copy is dirty
  - Remote is owner

Read Request



Read Exclusive Request

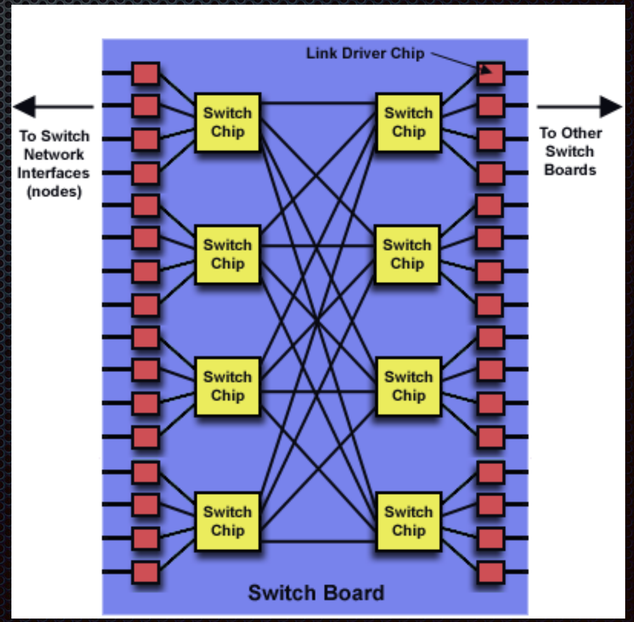


Scaling

- Full map directory limited to size of bit vector representing clusters
  - Each memory block extended with map vector
- Partial map directory defaults to broadcast when more clusters are present
  - Memory block has short list of sharing clusters
- Chained map directory uses linked structures, has variable access times, complex ownership protocol

Distributed Memory

- Sharing in local cluster but not beyond
- High performance network for memory to memory data movement
  - Infiniband, Myrinet, custom
- Special network stack for low latency
- Programmer manages data placement and messaging for sharing copies
- Often requires new algorithms



Images: Lawrence Livermore National Laboratory

Hybrid

- Modern nodes have up to 24 cores per socket, often dual sockets, with shared memory
  - Local computations (e.g., 96 threads) communicate through shared memory
- Nonlocal communication uses message passing (e.g., MPI)
- Partitioned global address space (PGAS languages) try to elevate the abstraction by adding information about locality, partitioning, communication