Lecture 13:
Memory models and SPH

David Bindel

10 Mar 2014

Logistics

- HW 2 is due tonight
- HW 3 is posted, due April 1 (details today)
- Project proposals are also due April 1 (or sooner)

On the topic of HW 2...

How would we parallelize `wave1d` with OpenMP?

1D wave equation with OpenMP

1. Just parallelize main loop?
   - Certainly the easiest strategy – efficiency?
   - What scheduler would we use?
2. Treat almost like MPI code?
   - How is information exchanged across states?
   - What barriers are needed?
   - Could we take multiple steps between barriers?

Things to still think about with OpenMP

- Proper serial performance tuning?
- Minimizing false sharing?
- Minimizing synchronization overhead?
- Minimizing loop scheduling overhead?
- Load balancing?
- Finding enough parallelism in the first place?

Let's focus again on memory issues...

Memory model

- Single processor: return last write
  - What about DMA and memory-mapped I/O?
- Simplest generalization: *sequential consistency* – as if
  - Each process runs in program order
  - Instructions from different processes are interleaved
  - Interleaved instructions ran on one processor

Sequential consistency

A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

– Lamport, 1979

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Example: Spin lock

Initially, `flag = 0` and `sum = 0`

Processor 1:

```
sum += p1;
flag = 1;
```

Processor 2:

```
while (!flag);
sum += p2;
```

Without sequential consistency support, what if

1. Processor 2 caches `flag`?
2. Compiler optimizes away loop?
3. Compiler reorders assignments on P1?

Starts to look restrictive!

Sequential consistency: the good, the bad, the ugly

Program behavior is “intuitive”:

- Nobody sees garbage values
- Time always moves forward

One issue is *cache coherence*:

- Coherence: different copies, same value
- Requires (nontrivial) hardware support

Also an issue for optimizing compiler!

There are cheaper *relaxed* consistency models.

Snoopy bus protocol

Basic idea:

- Broadcast operations on memory bus
- Cache controllers “snoop” on all bus transactions
  - Memory writes induce serial order
  - Act to enforce coherence (invalidate, update, etc)

Problems:

- Bus bandwidth limits scaling
- Contending writes are slow

There are other protocol options (e.g. directory-based).
But usually give up on *full* sequential consistency.

Weakening sequential consistency

Try to reduce to the *true* cost of sharing

- `volatile` tells compiler when to worry about sharing
- Memory fences tell when to force consistency
- Synchronization primitives (lock/unlock) include fences

Sharing

True sharing:

- Frequent writes cause a bottleneck.
- Idea: make independent copies (if possible).
- Example problem: malloc/free data structure.

False sharing:

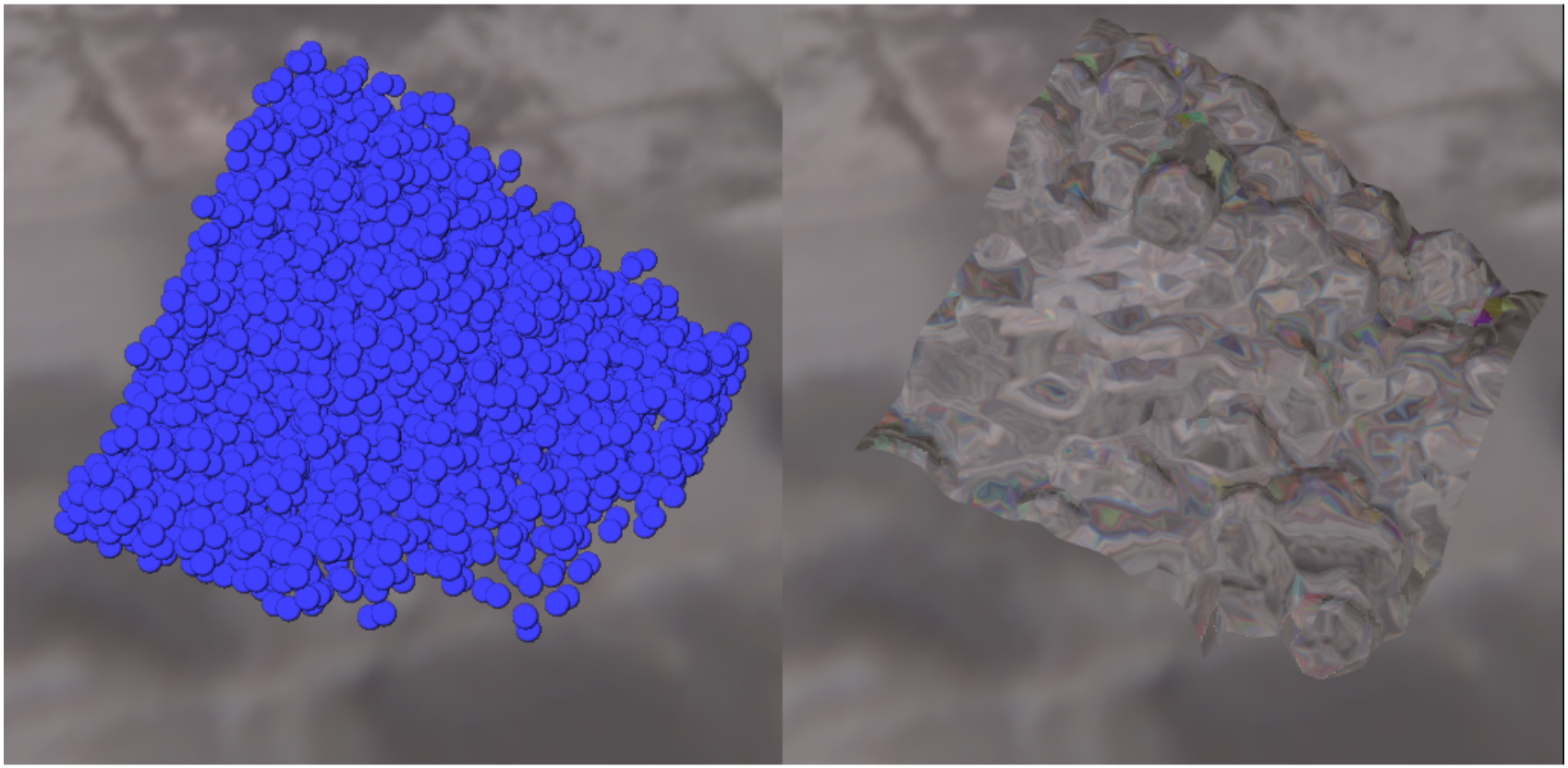
- Distinct variables on same cache block
- Idea: make processor memory contiguous (if possible)
- Example problem: array of ints, one per processor

Take-home message

- Sequentially consistent shared memory is a useful idea...
  - “Natural” analogue to serial case
  - Architects work hard to support it
- ... but implementation is costly!
  - Makes life hard for optimizing compilers
  - Coherence traffic slows things down
  - Helps to limit sharing

Have to think about these things to get good performance.

Your next mission!



Smoothed Particle Hydrodynamics (SPH)

- Particle based method for fluid simulation
  - Representative of other particle-based methods
  - More visually interesting than MD with Lennard-Jones?
- Particle i (a fluid blob) evolves according to

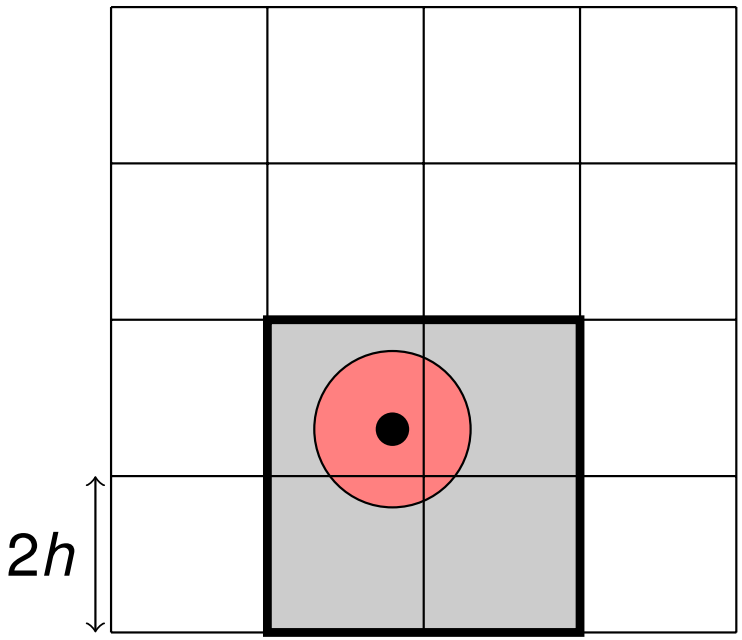
$$m\mathbf{a}_i = \sum_{|\mathbf{x}_j - \mathbf{x}_i| \leq h} \mathbf{f}_{ij}$$

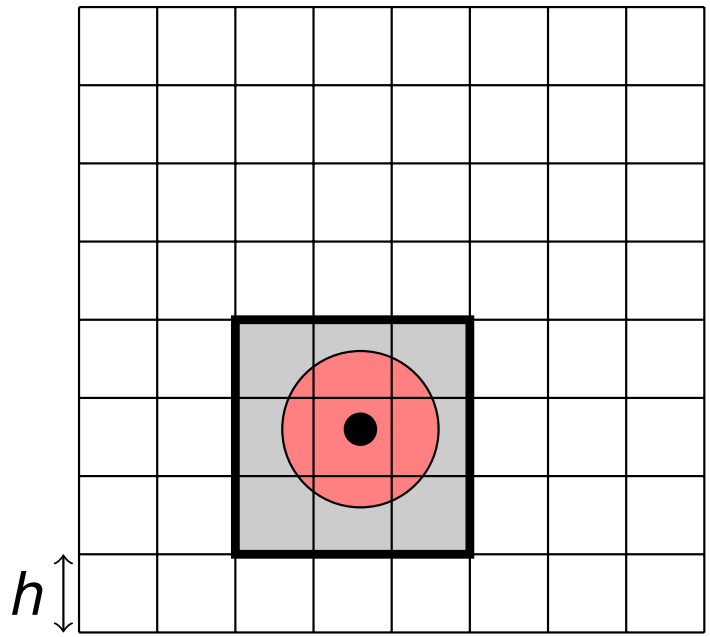
  where force law satisfies $\mathbf{f}_{ij} = -\mathbf{f}_{ji}$.
- Chief performance challenge: fast evaluation of forces!

Task 1: Binning / spatial hashing

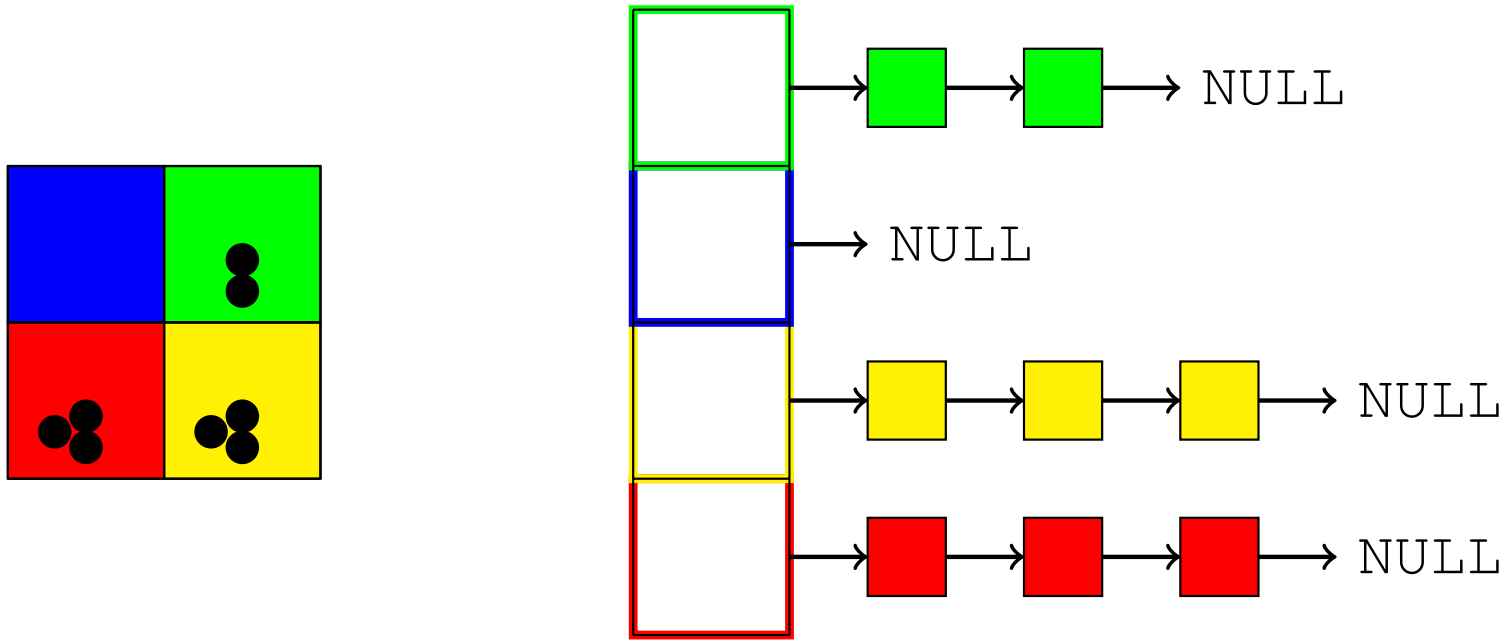
- Partition space into bins of size $\geq h$ (interaction radius)
- Only check for interactions in nearby bins
- Trade off between bin size, number of interaction checks

Task 1: Binning / spatial hashing





Task 1: Binning / spatial hashing



- Keep particles in an array as usual
- Also keep array of head pointers for bins
- Thread linked list structures for bin contents

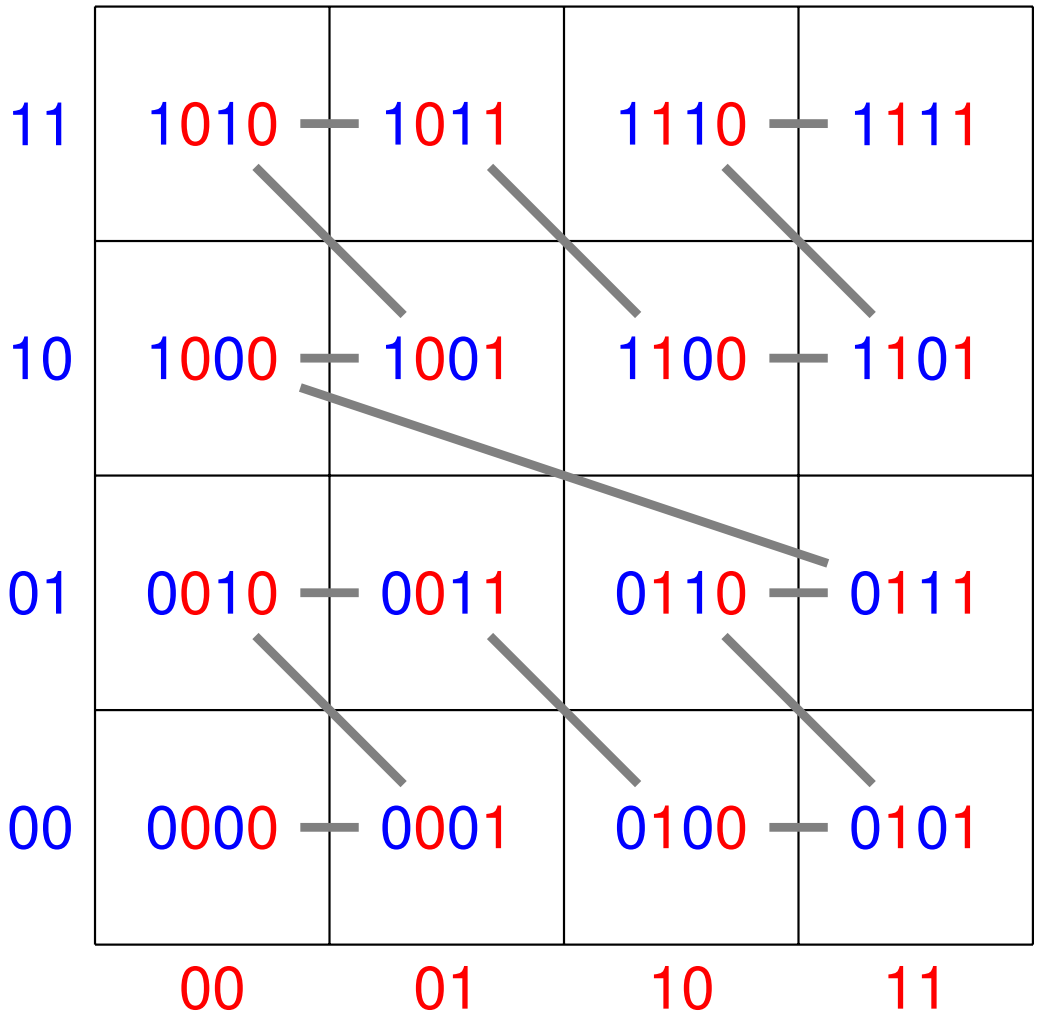
Task 1: Binning / spatial hashing

- Typical situation: lots of empty bins
- Empty boxes take up space!
- Alternative: spatial hashing
  - Map multiple bins to one storage location
  - Avoid *collisions* (several bins map to same place)
  - Maybe preserve locality?
- Idea: Figure out potential neighbors good for a few steps?

Ordering

- Bins naturally identified with two or three indices
- Want to map to a single index
  - To serve as a hash key
  - For ordering computations
- Row/column major: mediocre locality
- Better idea: Z-Morton ordering
  - Interleave bits of (x, y, z) indices
  - Efficient construction a little fiddly
  - But basic picture is simple!

Z Morton ordering



Equivalent to height-balanced quadtree / octtree.

Task 2: Profiling

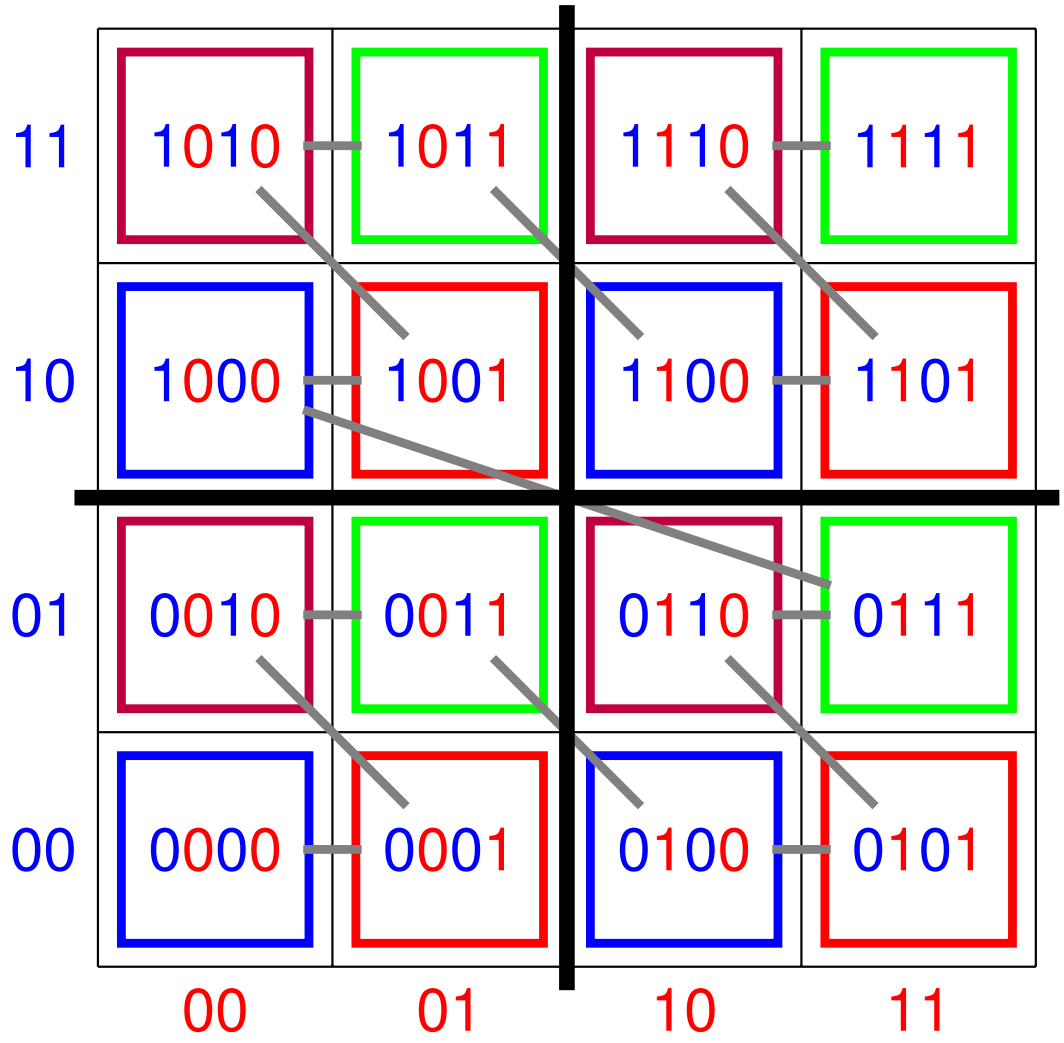
- Computation involves several different steps
  - Finding nearest neighbors
  - Computing local densities
  - Computing local pressure / viscous forces
  - Time stepping
  - I/O
- Want to act based on timing data
  - Manual instrumentation?
  - Profilers?
- Which takes the most time? Fix that first.
- Consider both algorithm improvements and tuning.
- Lecture Thursday will be (at least partly) profiling.

Task 3: Parallelization

Two major issues:

- How do we decompose the problem?
  - Processors own regions in space?
  - Processors own fixed sets of particles?
  - Processors own sets of possible force interactions?
  - Hybrids recommended!
- How do we synchronize force computations?
  - Note: Compute $\mathbf{f}_{ij}$ and $\mathbf{f}_{ji}$ simultaneously now
  - Could keep multiple updates per processor and reduce?
  - Could use multi-color techniques
    (no two processors handle neighbors concurrently)?
  - Interacts with the problem decomposition!

Example approach: multi-color ordering



No blue cell neighbors another blue.

Things to think about

- How do we make sure we don't break the code?
- How fast is parallel code for $p = 1$?
- Are there load balance issues?
- Do we synchronize frequently (many times per step)?
- Do we get good strong scaling?
- Do we get good weak scaling?
  - Note: Smaller particles $\implies$ smaller time step needed!
  - Have to be careful to compare apples to apples

Task 4: Play

- How fast can we make the serial code?
- How should we improve the initialization?
- Could we time step more intelligently?
- What about surface tension (see Müller et al)?
- What about better boundary conditions?
- What about swishing, pouring, etc?
- What about recent improvements for incompressible flow?