# CS 5220

## Shared memory

David Bindel

2024-10-03

# OpenMP

# OpenMP

- Standard API for multi-threaded code
  - Only a spec — multiple implementations
  - Lightweight syntax
  - C or Fortran (with appropriate compiler support)
- High level:
  - Preprocessor/compiler directives (80%)
  - Library calls (19%)
  - Environment variables (1%)

# OpenMP

Basic syntax: `#omp` *construct* `[` *clause* ... `]`

- Usually affects structured block (one way in/out)
- OK to have `exit()` in such a block

# Last time

- Creating parallel regions
- Sharing annotations
- Synchronization
- Parallel for

# Parallel for

```
void accumulate_a2a(std::vector<double>& f, const std::vector
{
    int n = f.size();

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        double fi = 0.0;
        double xi = x[i];
        for (int j = 0; j < n; ++j) {
            double dij = xi-x[j];
            if (dij != 0.0)
                fi += 1.0/(dij*dij);
        }
        f[i] = fi;
    }
}
```

# What about symmetry?

```
void accumulate_a2a(std::vector<double>& f, const std::vector
{
    int n = f.size();

    for (int i = 0; i < n; ++i) {
        double xi = x[i];
        for (int j = i+1; j < n; ++j) {
            double dij = xi-x[j];
            f[i] += 1.0/(dij*dij);
            f[j] += 1.0/(dij*dij);
        }
    }
}
```

Why would `omp parallel for` fail here?

# Other parallel work divisions

- `sections`: like cobegin/coend
- `single`: do only in one thread (e.g. I/O)
- `master`: do only in master thread; others skip

# Sections

```
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        do_something();

        #pragma omp section
        and_something_else();

        #pragma omp section
        and_this_too();
        // No implicit barrier here
    }
    // Implicit barrier here
}
```

# Task-based parallelism

- Work-sharing so far is rather limited
  - Work cannot be produced/consumed dynamically
  - Fine for data parallel array processing...
  - ... but what about tree walks and such?
- Alternate approach (OpenMP 3.0+): Tasks

# Tasks

Task involves:

- Task construct: task directive plus structured block
- Task: Task construct + data

Tasks are handled by run time, complete at barriers or `taskwait`.

# Example: List traversal

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (link_t* link = head; link; link = link->next)
            #pragma omp task firstprivate(link)
            process(link);
    }
    // Implicit barrier
}
```

One thread generates tasks, others execute them.

# Example: Tree traversal

```
int tree_max(node_t* n)
{
    int lmax, rmax;
    if (n->is_leaf)
        return n->value;

    #pragma omp task shared(lmax)
        lmax = tree_max(n->l);
    #pragma omp task shared(rmax)
        rmax = tree_max(n->l);
    #pragma omp taskwait

    return max(lmax, rmax);
}
```



The `taskwait` waits for all child tasks

# Example: Quicksort

```
void omp_qsort(int* a, int lo, int hi)
{
    if (lo >= hi) return;
    int p = partition(a, lo, hi);
    #pragma omp task shared(a)
    omp_qsort(a, lo, p);
    #pragma omp task shared(a)
    omp_qsort(a, p, hi);
}

void call_qsort(int* a, int lo, int hi)
{
    #pragma omp parallel
    {
        #pragma omp single
```

# Task dependencies

What happens if one task produces what another needs?

```
#pragma omp task depend(out:x)
x = foo();
#pragma omp task depend(in:x)
y = bar(x);
```

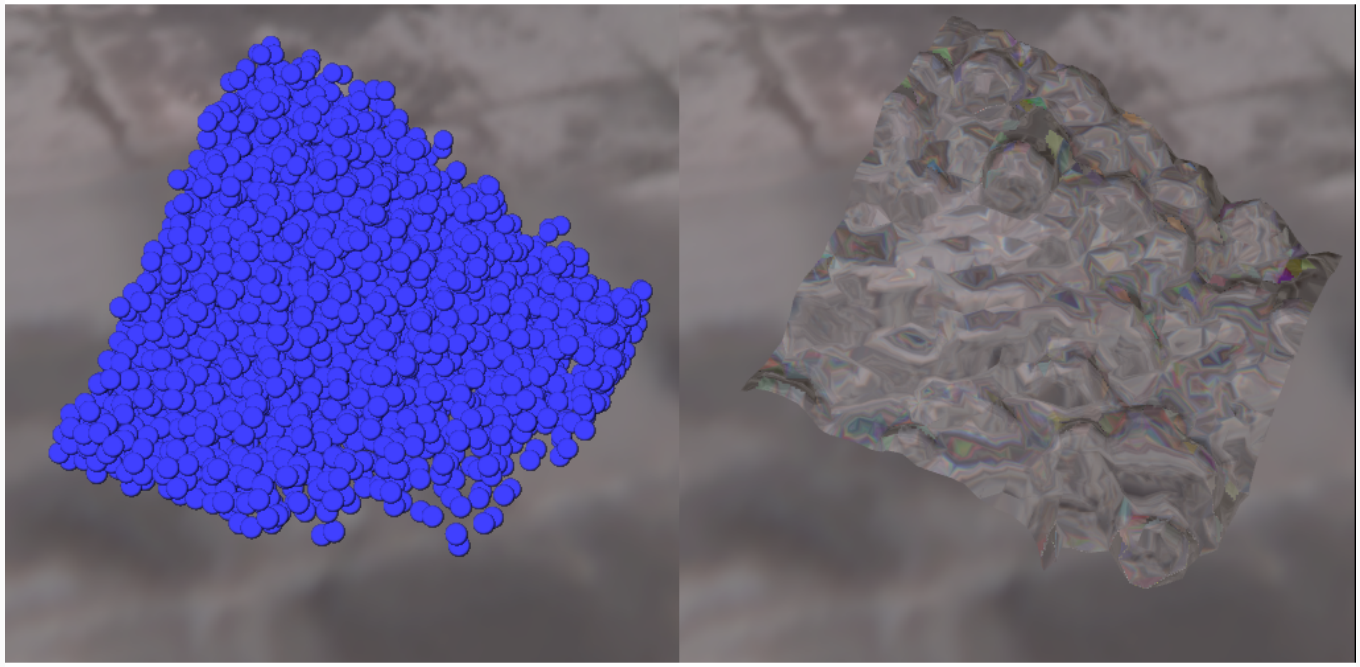
# Topics not addressed

- Low-level synchronization (locks, flush)
- OpenMP 4.x constructs for accelerator interaction
- A variety of more specialized clauses

See http://www.openmp.org/

# SPH

# Your next project!



# Smoothed Particle Hydrodynamics

- 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 as

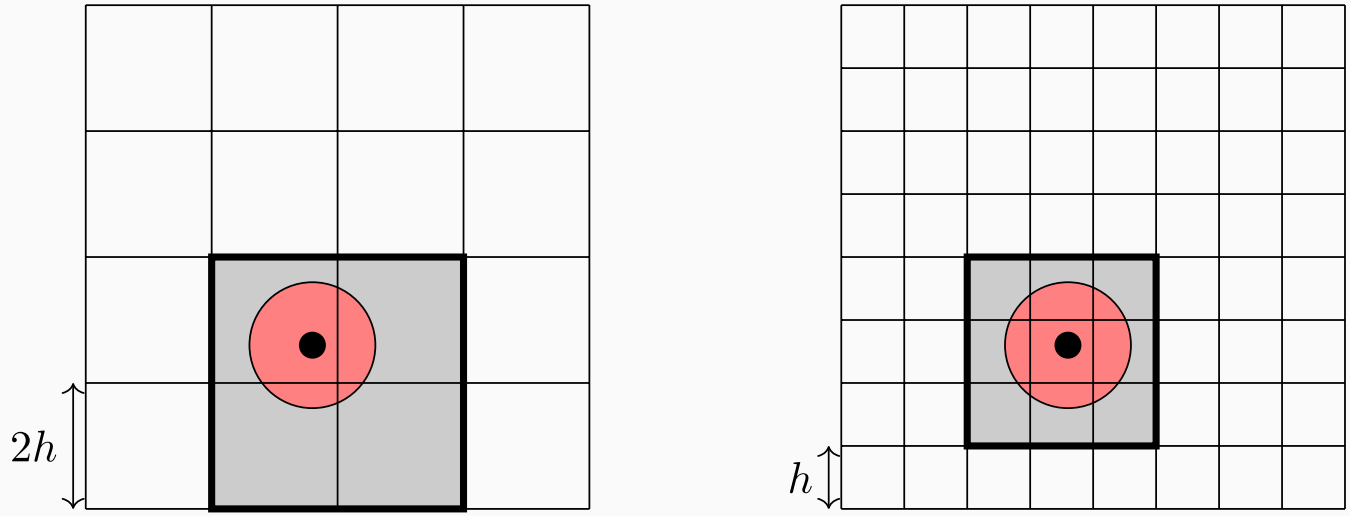
$$m\mathbf{a}_i = \sum_{|\mathbf{x}_j - \mathbf{x}_i| \leq h} \mathbf{f}_{ij}$$

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

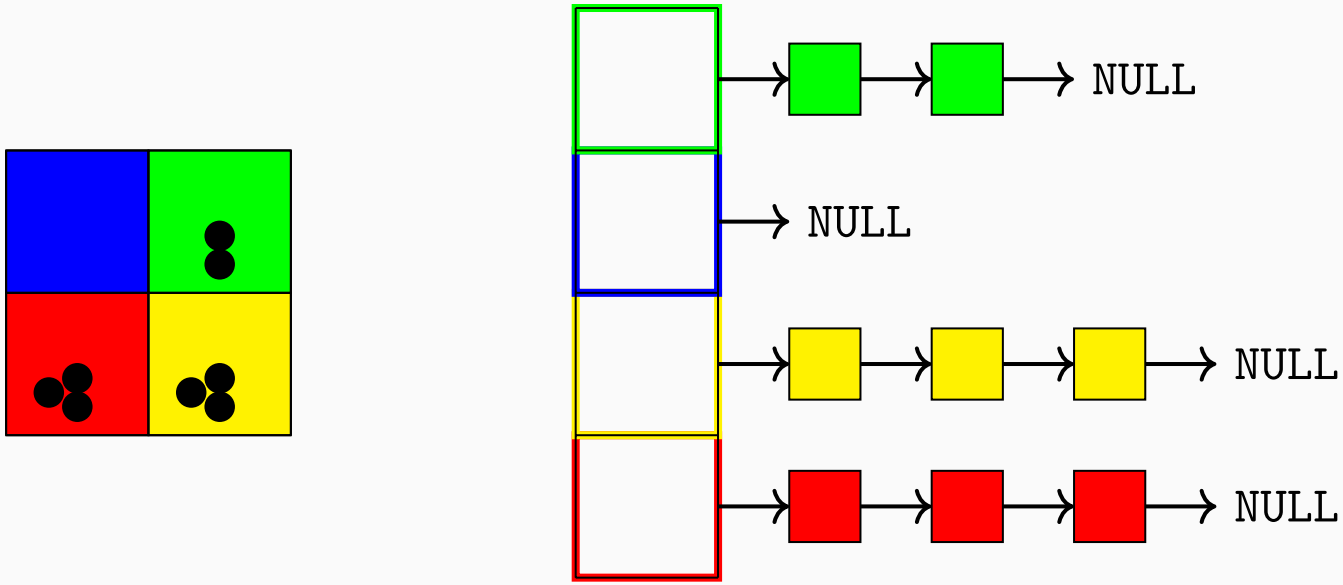
# Task 1: Binning / spatial hashing

- Partition space into bins of size $\geq h$ (interaction radius)
- Only check interactions in nearby bins
- Tradeoff 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

```
struct particle_t {
    float rho;          // Density
    float x[3];         // Position
    float v[3];         // Velocity (full step)
    float vh[3];        // Velocity (half step)
    float a[3];         // Accelerations
    particle_t* next; // Link for hashing
};

struct sim_state_t {
    float mass;                         // Particle mass
    std::vector<particle_t> part;    // Particles
    std::vector<particle_t*> hash;  // Hash table
};
```

# Task 1: Binning / spatial hashing

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

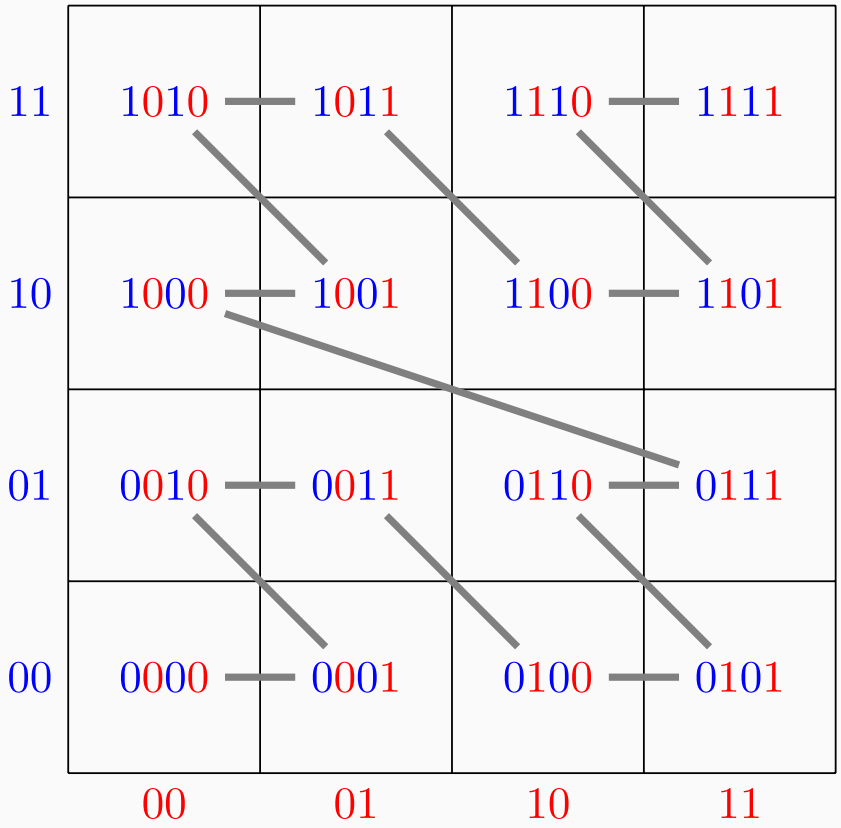
# Ordering

- Bins naturally identified with two or three indices
- Want to map to a single index
    - To serve as a hash key
    - For ordering computations

# Ordering

- Row/column major: mediocre locality
- Better: Z-Morton ordering
  - Interleave bits of $(x, y, z)$ indices
  - Efficient construction a little fidly
  - 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

How much does each cost? Scaling?

# Profiling

- Want to act based on timing data
  - Manual instrumentation?
  - Profilers?
- Fix what takes most time first
- Consider both algorithm improvements and tuning

# Task 3: Parallelization

How do we decompose the problem?

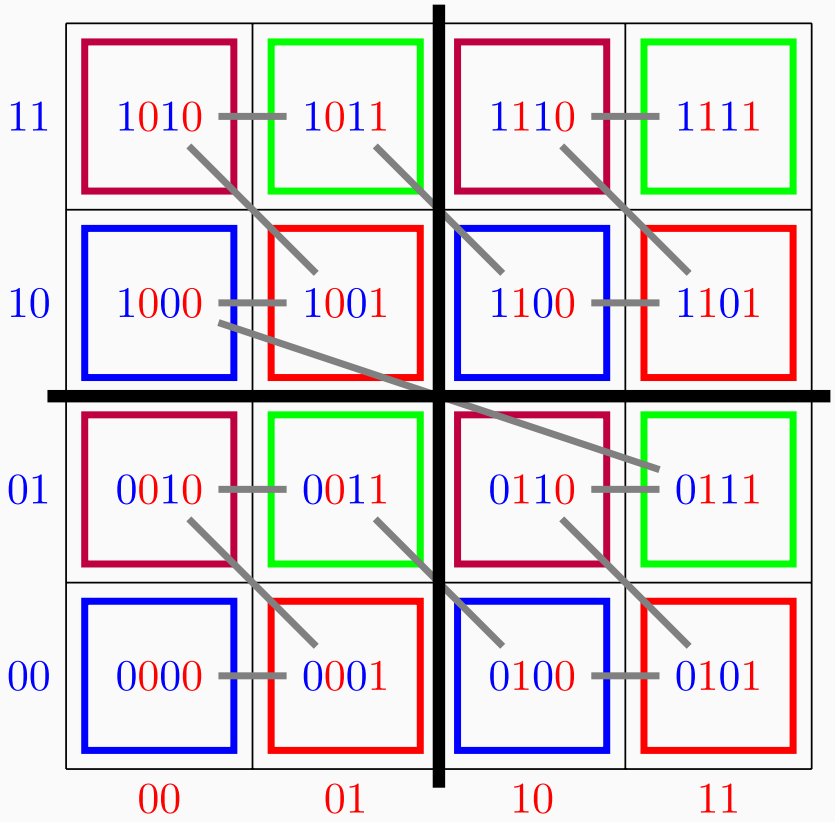
- Processors own regions of space?
- Processors own fixed set of particles?
- Processors own sets of force interactions?
- Hybrid recommended!

# Task 3: Parallelization

How do we synchronize force computation?

- Compute $\mathbf{f}_{ij}$ and $\mathbf{f}_{ji}$ simultaneously in reference code
- Could keep multiple updates per processor and reduce?
- Use multi-color techniques?
- Interacts with problem decomposition!

# Example: 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 the parallel code for $p = 1$?
- Are there load balance issues?
- Do we synchronize often?
- Do we get good strong scaling? weak scaling?

# Task 4: Play

- How fast can we make the serial code?
- How should we improve initialization?
- Could we time step smarter?
- What about surface tension?
- What about better BCs?
- Swishing, pouring, etc?
- Improvements for incompressible flow?