Decomposing a Problem for Parallel Execution

Pablo Halpern <pablo.g.halpern@intel.com>
Parallel Programming Languages Architect, Intel Corporation

CppCon, 9 September 2014

Goal

Learn how to decompose a problem so that it can be efficiently distributed among multiple cores

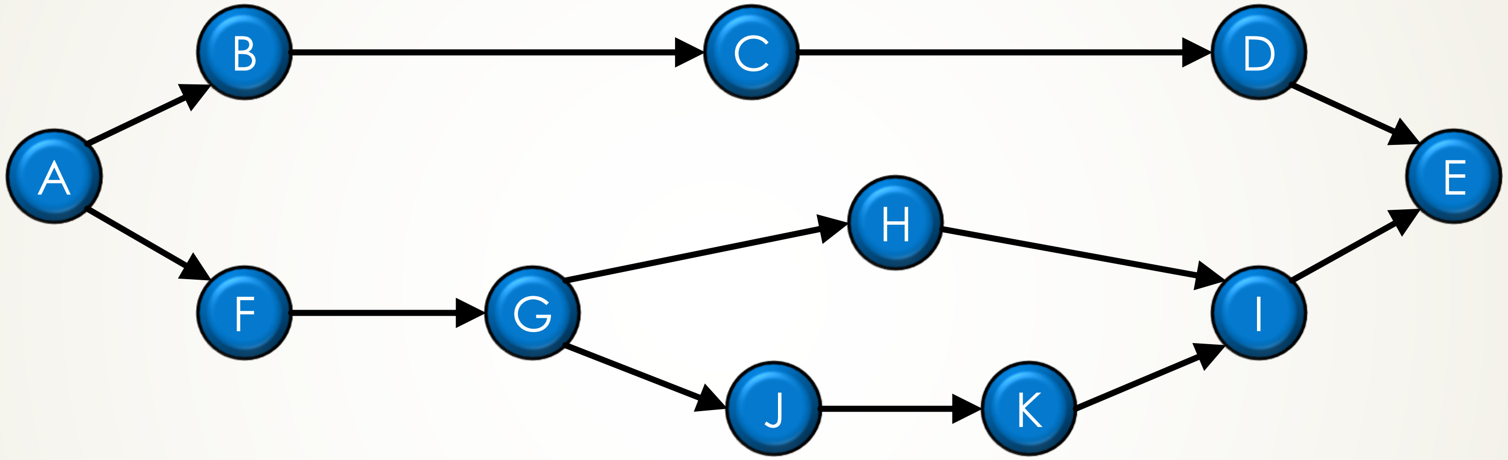
Summary

- The star-counting problem
  - A relatively easy problem
  - Exposure to a number of important issues
- The n-bodies problem
  - A more involved problem
  - With re-structuring, yields an elegant recursive solution with good cache behavior

Principles of parallelization

- The main challenge is identifying tasks within the program that minimally interact and, therefore, are logically parallel.
- Parallelization can be fun – a combination of discovery and invention, science and art.
  - One tries to discover the parallelism inherent in an algorithm or data structure.
  - One then chooses or invents new algorithms, refactors, simplifies, or approximates.

Parallelism is a graph-theoretical property of an algorithm



(Dependencies are opposite control flow, e.g. C depends on B)

- $A \prec B$ and $A \prec F$ (A *precedes* B and F)
- $B \parallel F$ (B is *in parallel with* F)
- $K \succ G$ (K *succeeds* G) and
- $K \parallel H$, $K \parallel B$ and $K \parallel C$, etc.

Cilk™ Plus as a teaching Language

```
int fib(int n)
{
    if (n < 2) return n;

    int a = cilk_spawn fib(n - 1);
    int b = fib(n - 2);
    cilk_sync;
    return a + b;
} // Implicit sync at end of function body
```

Execution *is allowed to* continue while fib(n-1) is running.

Asynchronous call must complete before using a.

```
cilk_for (auto i = vec.begin(); i != vec.end(); ++i)
{
    // Do something
} // Implicit sync at end of cilk_for
```

Iterations *are allowed to* execute concurrently.

Star-counting problem

- Introduction to the problem
- Serial implementation
- Find the parallelism
- Fix the race using atomic variable
- Improve performance using reduction

Count the stars in this image

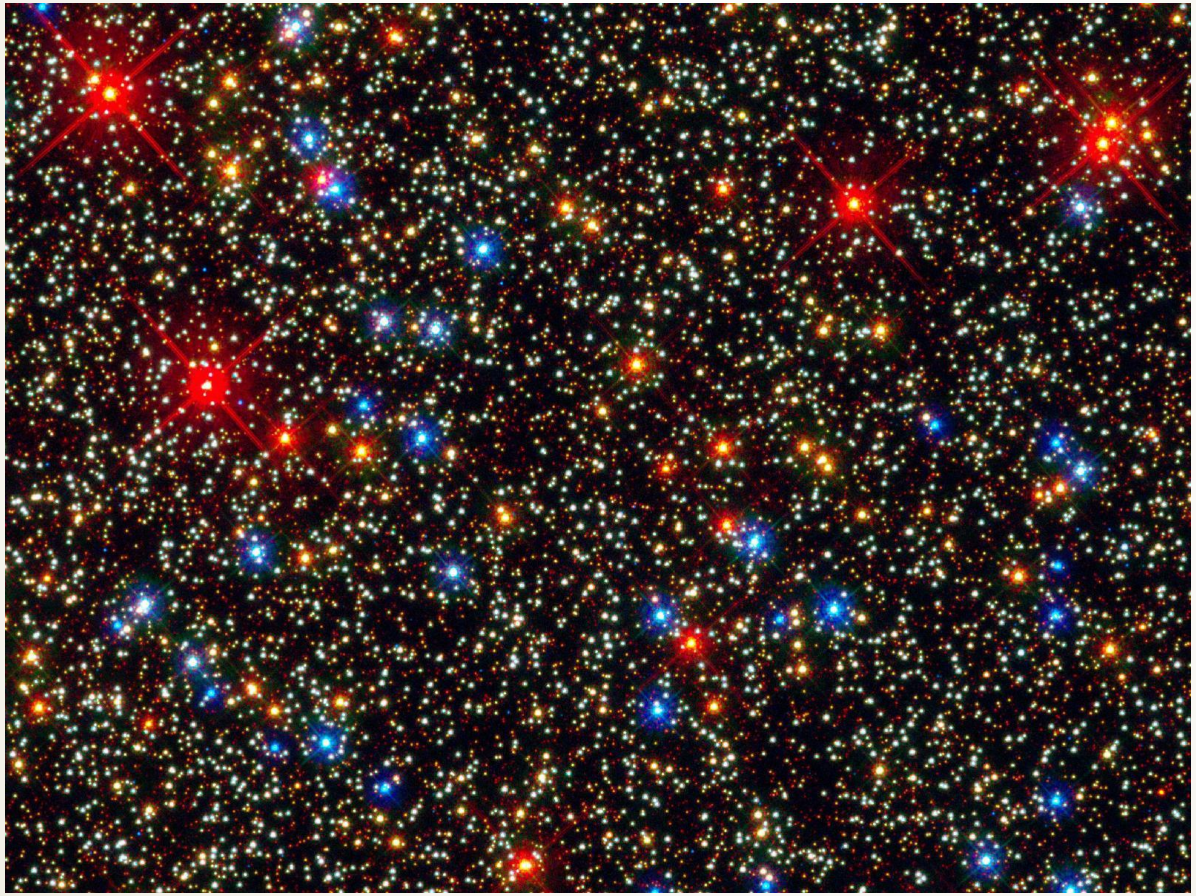


Photo courtesy NASA

Serial implementation

```
long count_stars(const Image& img)
{
    long count(0);
    // Iterate over the pixels of the image
    for (int x = 0; x < img.width(); ++x)
        for (int y = 0; y < img.height(); ++y)
            if (is_center_of_star(img, x, y))
                ++count;
    return count;
}
```

Parallelization usually starts with a working serial program

Disclaimer: This code is hypothetical. None of the variations of `count_stars` in this presentation have been implemented and tested.

Finding the unexpressed parallelism

```
long count_stars(const Image& img)
{
    long count(0);
    // Iterate over the pixels of the image
    for (int x = 0; x < img.width(); ++x)
        for (int y = 0; y < img.height(); ++y)
            if (is_center_of_star(img, x, y))
                ++count;
    return count;
}
```

Completely independent

Counting is independent but incrementing count is not.

Loops are a good source of potentially-independent tasks.

Straight-forward loop parallelism

```
long count_stars(const Image& img)
{
    long count(0);
    // Iterate over the pixels of the image
    cilk_for (int x = 0; x < img.width(); ++x)
        cilk_for (int y = 0; y < img.height(); ++y)
            if (is_center_of_star(img, x, y))
                ++count;
    return count;
}
```

Parallel updates

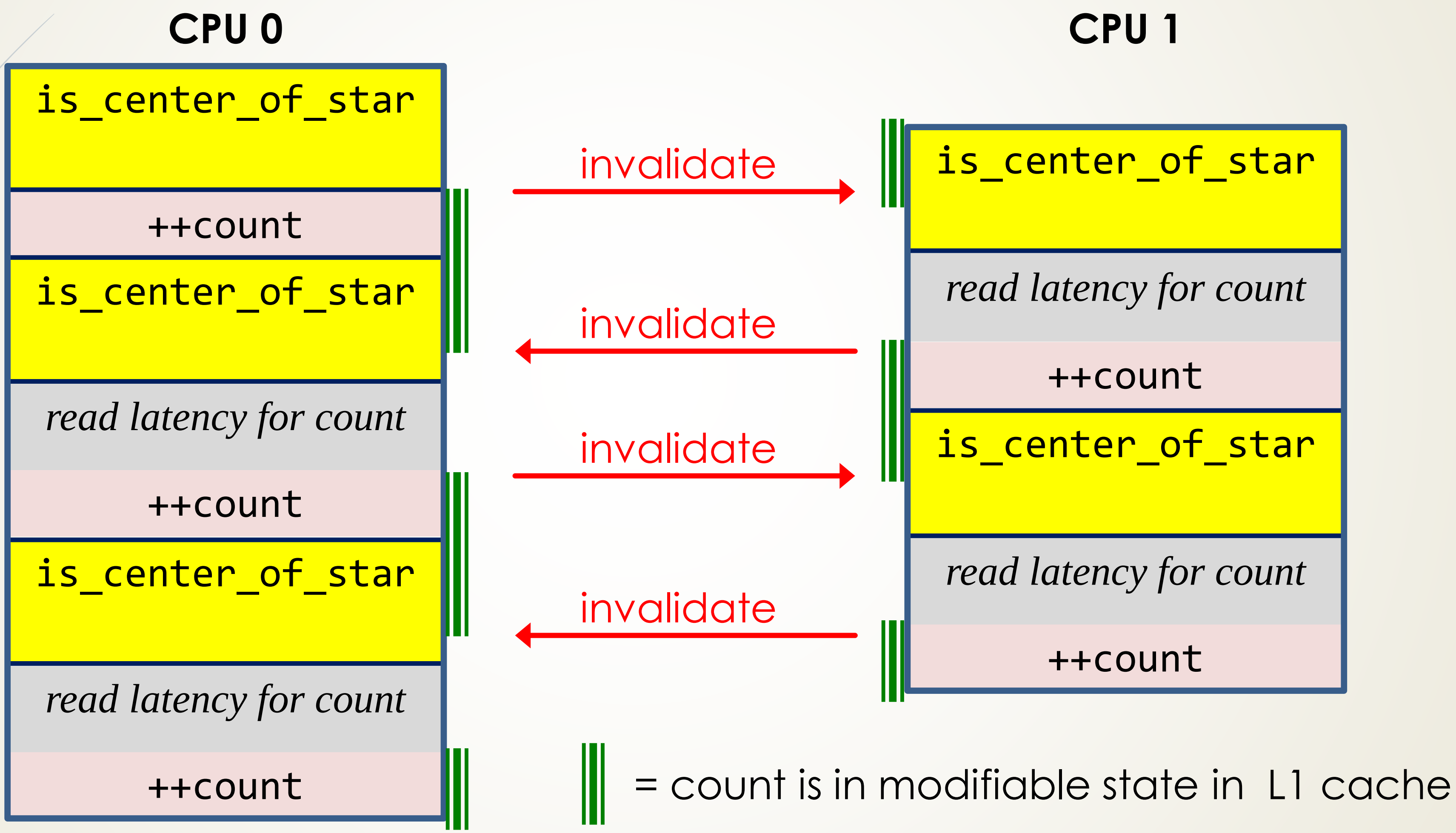
Correctness problem: data race

One solution: atomic variables

```
long count_stars(const Image& img)
{
    std::atomic<long> count(0);
    // Iterate over the pixels of the image
    cilk_for (int x = 0; x < img.width(); ++x)
        cilk_for (int y = 0; y < img.height(); ++y)
            if (is_center_of_star(img, x, y))
                ++count;
    return count;
}
```

Performance problem: Atomic variable contention

Cache Ping-Pong on atomic count



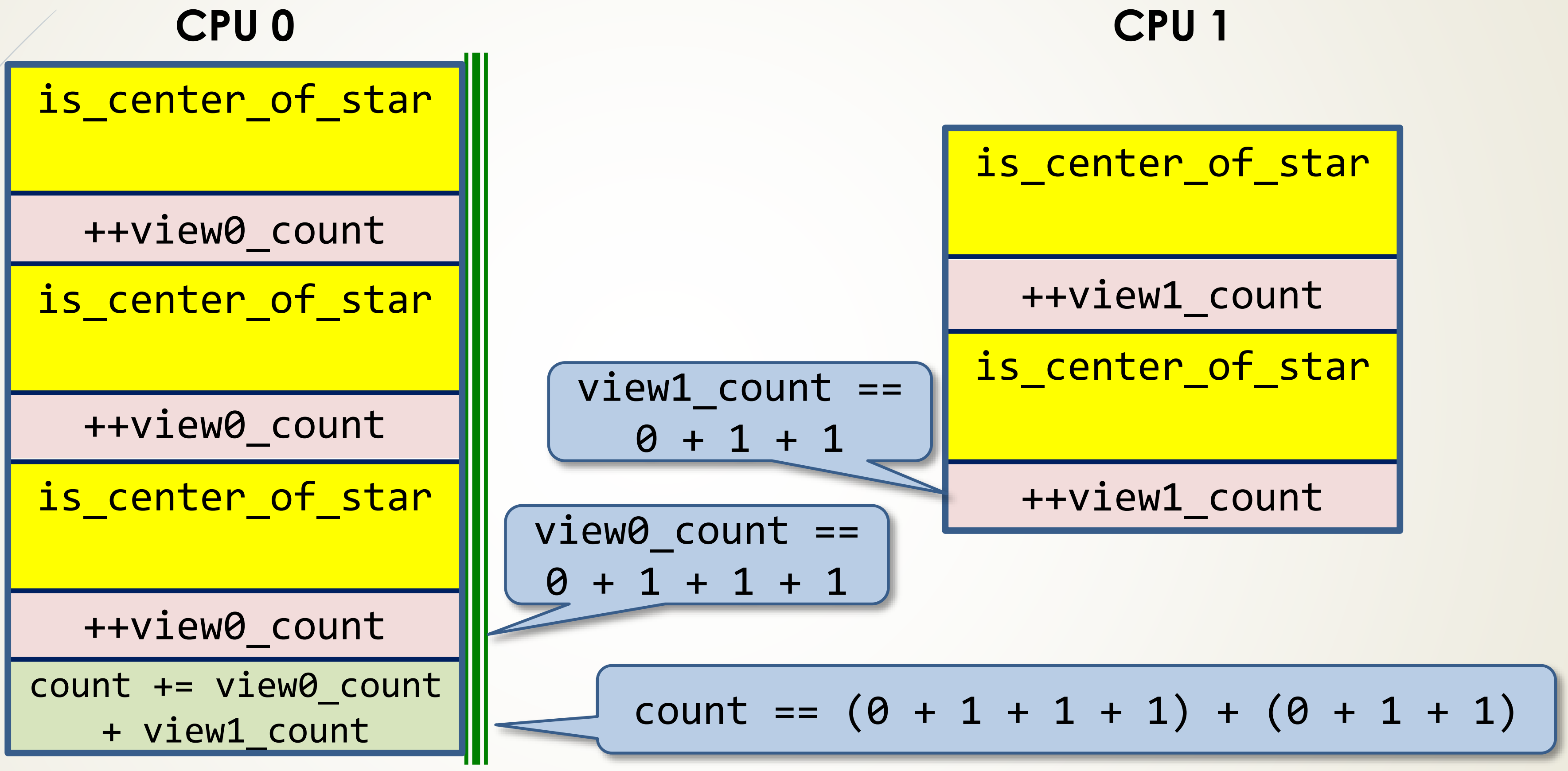
Better solution: reduction

```
long count_stars(const Image& img)
{
    cilk::reducer<cilk::op_add<long>> count_r(0);
    // Iterate over the pixels of the image
    cilk_for (int x = 0; x < img.width(); ++x)
        cilk_for (int y = 0; y < img.height(); ++y)
            if (is_center_of_star(img, x, y))
                ++*count_r;
    return count_r.get_value();
}
```

Pointer-like interface.

Each concurrent access to the reducer sees a different "view" of the variable. The parallel views are collapsed into a single value at the end of the computation.

Reducer operation (conceptual)

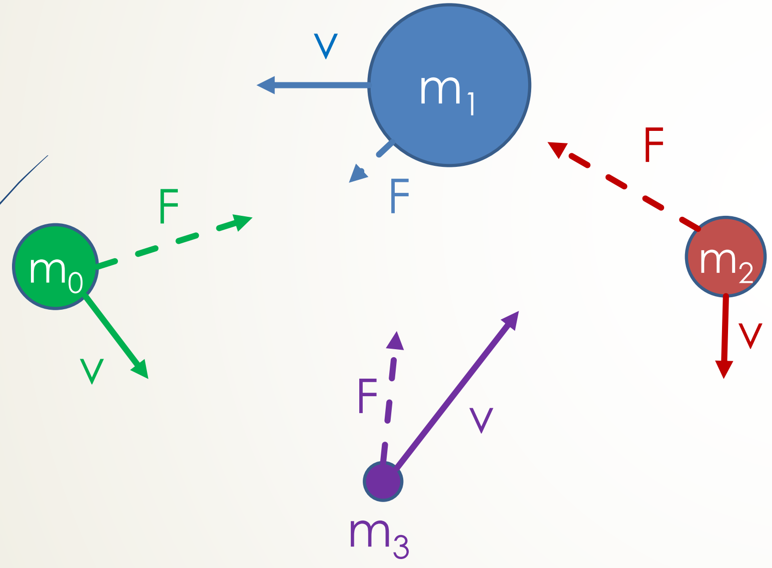


= count is in modifiable state in L1 cache

The n-bodies problem

- Introduction to the problem
- Basic implementation framework
- Parallelize the parts with parallel loops
- Try different approaches to mitigate data races
- Restructure the code into an elegant recursive algorithm with excellent cache locality

Gravity and planetary motion



To compute position, x' from position x after time increment Δt:

$$f_{ij} = \frac{G m_i m_j}{d_{ij}^{\,2}}$$

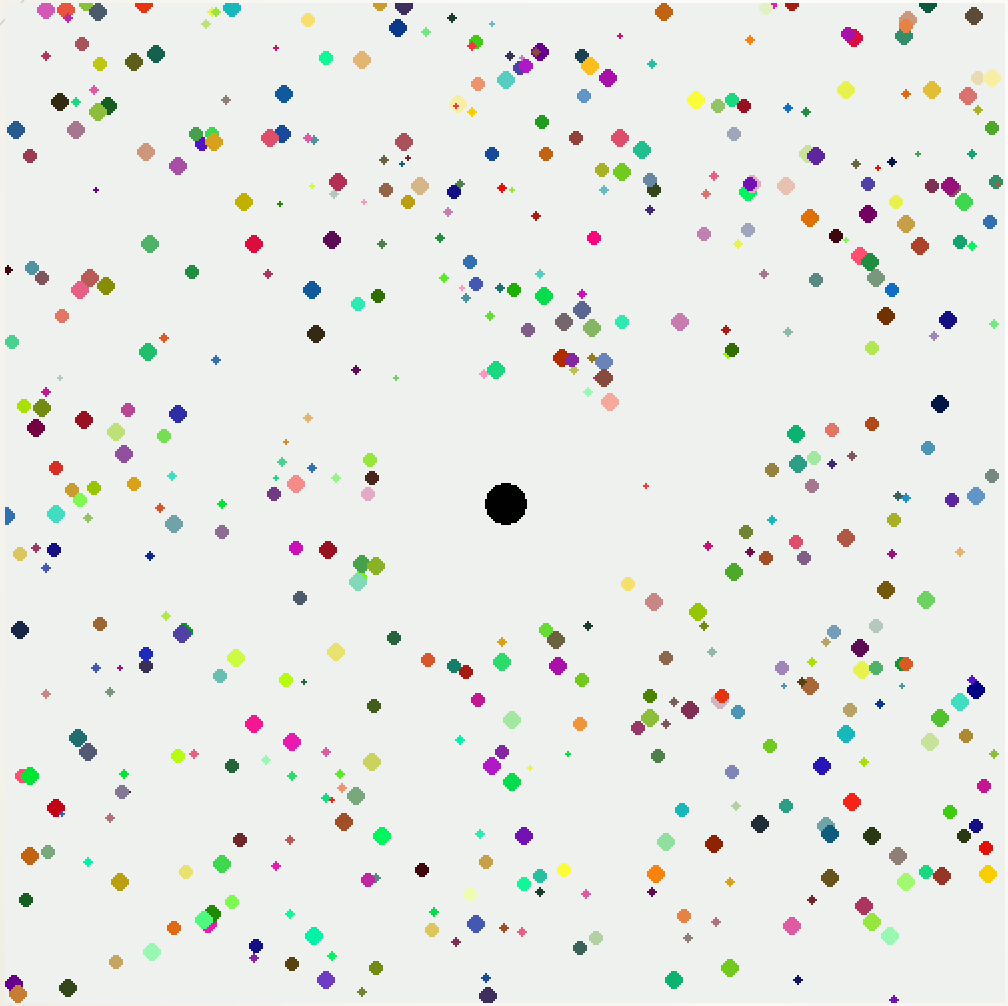
$$F_i = \sum_{j \neq i} f_{ij}$$

$$v_i' = vi + \frac{F_i \Delta t}{m_i}$$

$$x_i' = xi + avg(vi)\Delta t$$

Perform computation for each *i, j*.
Repeat for each time step.

A sample run of 4000 time steps



300 Bodies

New video frame for every 40 steps (total 100 frames)

Note: this is not a real-time animation. Still frames were combined into an animated GIF using an arbitrary frame rate of 10 fps, looped.

General framework of n-bodies

Data structure and main loop

```
struct Body {
    double x;        // x position
    double y;        // y position
    double xv;       // x velocity
    double yv;       // y velocity
    double xf;       // x force
    double yf;       // y force
    double mass;     // mass
    double density;  // density
    Pixel pix;       // color
};
```

```
int main(int argc, char* argv[])
{
    int nbodies = argc > 1 ? atoi(argv[1]) : 300;
    int nframes = argc > 2 ? atoi(argv[1]) : 100;
    Body *bodies = new Body[nbodies];
    initialize_bodies(nbodies, bodies);

    draw_frame(0, nbodies, bodies);
    for (int frame_num = 1; frame_num < nframes;
         ++frame_num) {
        for (int i = 0; i < steps_per_frame; ++i) {
            calculate_forces(nbodies, bodies);
            update_positions(nbodies, bodies);
        }
        draw_frame(frame_num, nbodies, bodies);
    }

    delete[] bodies;
}
```

Central computations

```
//  Compute force, (*fx,*fy) on body bi exerted by body bj
void calculate_force(double *fx, double *fy,
                     const Body &bi, const Body &bj)
{
    double dx = bj.x - bi.x;
    double dy = bj.y - bi.y;
    double dist2 = dx * dx + dy * dy; //  distance squared
    double dist = std::sqrt( dist2 );
    double f = bi.mass * bj.mass * GRAVITY / dist2;
    *fx = f * dx / dist;
    *fy = f * dy / dist;
}
```

$$f_{ij} = \frac{Gm_i m_j}{d_{ij}^{\ 2}}$$

```
//  Add force, (fx,fy) to body b
void add_force(Body* b, double fx, double fy)
{
    b->xf += fx;
    b->yf += fy;
}
```

$$F_i = \sum_{j \neq i} f_{ij}$$

Updating positions in parallel

The easier problem

Updating positions – serial

```
void update_positions(int nbodies, Body *bodies)
{
    for (int i = 0; i < nbodies; ++i) {
        // initial velocity
        double xv0 = bodies[i].xv;
        double yv0 = bodies[i].yv;
        // update velocity based on forces
        bodies[i].xv += TIME_QUANTUM * bodies[i].xf / bodies[i].mass;
        bodies[i].yv += TIME_QUANTUM * bodies[i].yf / bodies[i].mass;
        // clear forces for next iteration
        bodies[i].xf = 0.0;
        bodies[i].yf = 0.0;
        // update position based on average velocity
        bodies[i].x += TIME_QUANTUM * (xv0 + bodies[i].xv)/2.0;
        bodies[i].y += TIME_QUANTUM * (yv0 + bodies[i].yv)/2.0;
    }
}
```

$$v_i{}' = vi + \frac{F_i \Delta t}{m_i}$$

$$x_i{}' = xi + avg(vi)\Delta t$$

Updating positions – parallel

```
void update_positions(int nbodies, Body *bodies)
{
    cilk_for (int i = 0; i < nbodies; ++i) {
        // initial velocity
        double xv0 = bodi
        double yv0 = bodi
        // update velocit
        bodies[i].xv += T
        bodies[i].yv += TIME_QUANT
        // clear forces for next i
        bodies[i].xf = 0.0;
        bodies[i].yf = 0.0;
        // update position based on average velocity
        bodies[i].x += TIME_QUANTUM * (xv0 + bodies[i].xv)/2.0;
        bodies[i].y += TIME_QUANTUM * (yv0 + bodies[i].yv)/2.0;
    }
}
```

Done!

```
tbb::parallel_for(0, nbodies, [&](int i){
    …
});
```

```
#pragma omp parallel for
for (int i = 0; i < nbodies ++i)
{
    …
}
```

Calculating forces in parallel

The harder problem

Calculating forces – naïve serial

```
void calculate_forces(int nbodies, Body *bodies) {
    for (int i = 0; i < nbodies; ++i) {
        for (int j = 0; j < nbodies; ++j) {
            // update the force vector on bodies[i] exerted
            // by bodies[j].
            if (i == j) continue;

            double fx, fy;
            calculate_force(&fx, &fy, bodies[i], bodies[j]);
            add_force(&bodies[i], fx, fy);

        }
    }
}
```

n(n – 1) applications of `calculate_force()`

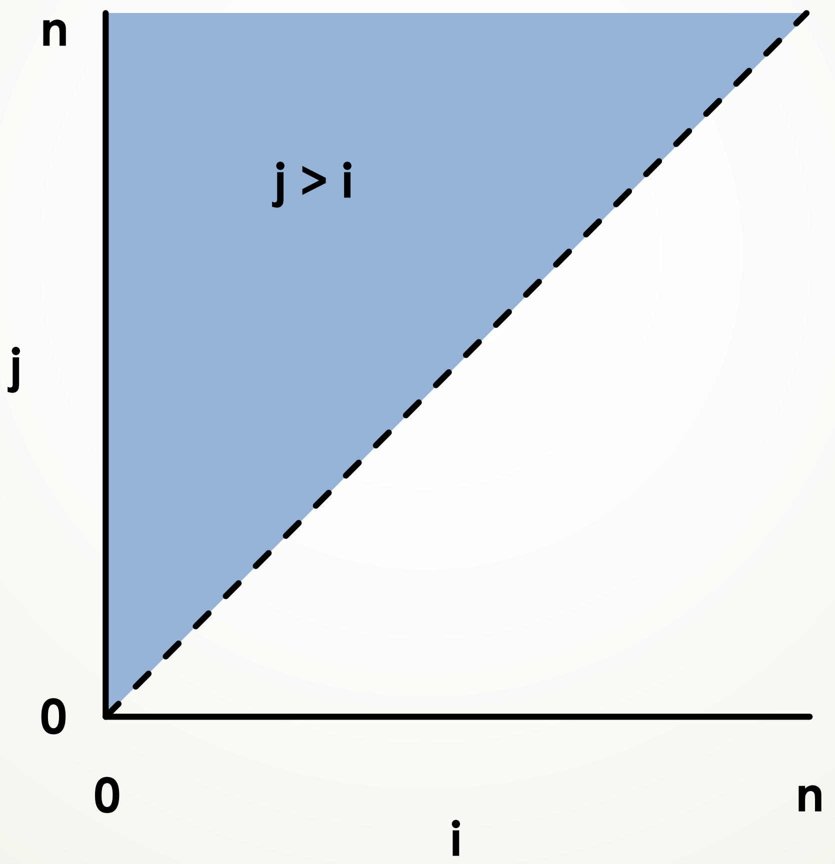
Calculating forces – half the work

```
void calculate_forces(int nbodies, Body *bodies) {
    for (int i = 0; i < nbodies; ++i) {
        for (int j = i + 1; j < nbodies; ++j) {
            // update the force vector on bodies[i] exerted
            // by bodies[j].


            double fx, fy;
            calculate_force(&fx, &fy, bodies[i], bodies[j]);
            add_force(&bodies[i], fx, fy);
            add_force(&bodies[j], -fx, -fy);
        }
    }
}
```

n(n – 1)/2 applications of `calculate_force()`

Graphical representation of iteration space



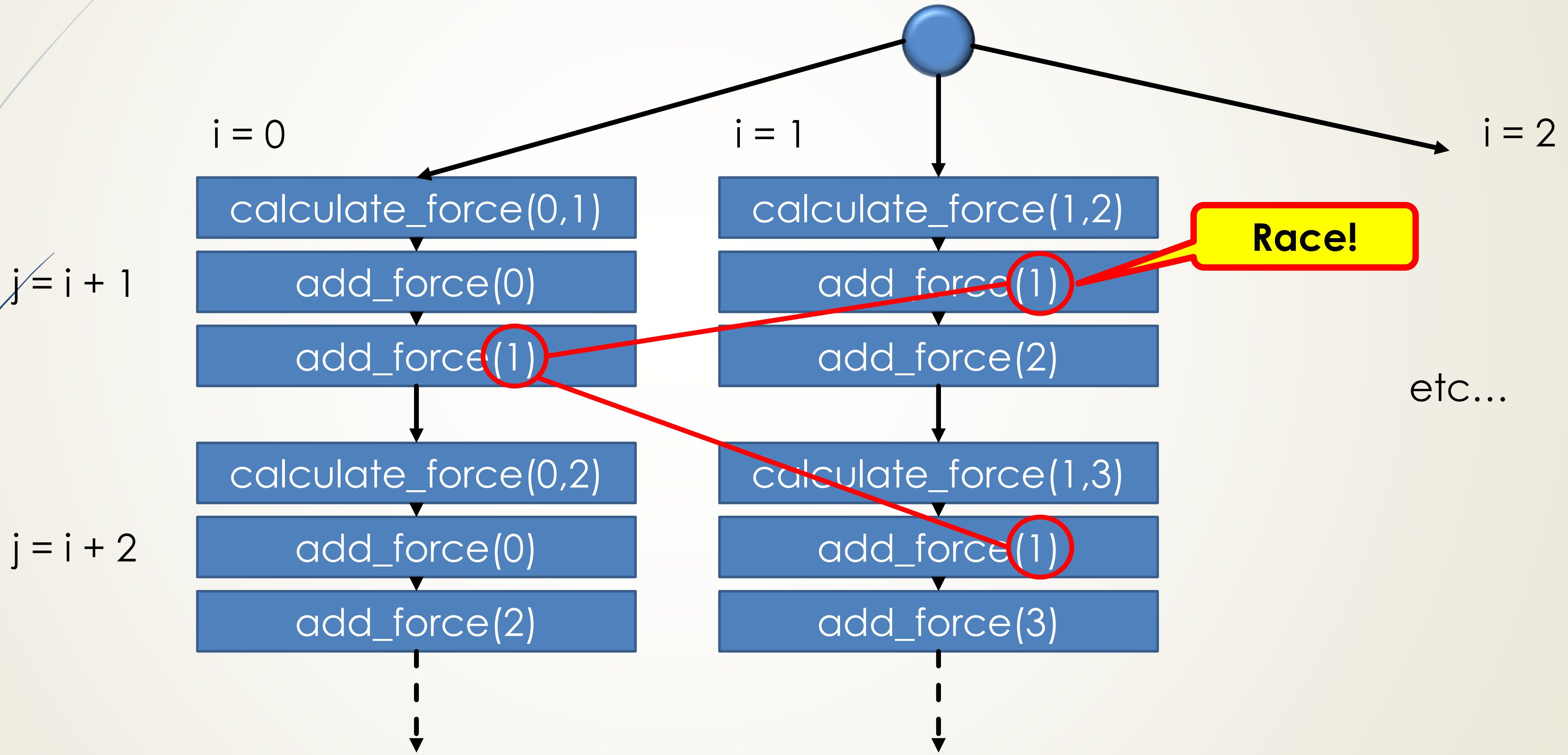
Calculating forces – naïve parallel

```
void calculate_forces(int nbodies, Body *bodies) {
    cilk_for (int i = 0; i < nbodies; ++i) {
        for (int j = i + 1; j < nbodies; ++j) {
            // update the force vector on bodies[i] exerted
            // by bodies[j].


            double fx, fy;
            calculate_force(&fx, &fy, bodies[i], bodies[j]);
            add_force(&bodies[i], fx, fy);
            add_force(&bodies[j], -fx, -fy);
        }
    }
}
```

parallel application of `calculate_force()` and `add_force()`

A look at the parallel execution



"Obvious solution": embed a mutex

```
struct Body {
    …
    SmallMutex mutex;  //  Maybe a spin lock?
};
```

```
{
    std::lock_guard<SmallMutex> g(bodies[i].mutex);
    add_force(&bodies[i], fx, fy);
}
{
    std::lock_guard<SmallMutex> g(bodies[j].mutex);
    add_force(&bodies[j], -fx, -fy);
}
```

Alternative “solution”: hashed mutexes

```
struct Body {
    …
    static std::mutex mutex_array[64];
    std::mutex& mutex() {
        size_t hash = size_t(this) / sizeof(Body);
        return mutex_array[hash % 64];
    }
};
```

```
{
    std::lock_guard<std::mutex> g(bodies[i].mutex());
    add_force(&bodies[i], fx, fy);
}
{
    std::lock_guard<std::mutex> g(bodies[j].mutex());
    add_force(&bodies[j], -fx, -fy);
}
```

What about atomics?

```
struct Body {
    …
    std::atomic<double> xf;      // x force
    std::atomic<double> yf;      // y force
    …
};
```

```
// Add force, (fx,fy) to body b
void add_force(Body* b, double fx, double fy)
{
    b->xf += fx;
    b->yf += fy;
}
```

No atomic increment for floats

```
// Add force, (fx,fy) to body b
void add_force(Body* b, double fx, double fy)
{
    double oxf = b->xf, oyf = b->yf;
    while (b->xf.compare_exchange_weak(oxf, oxf + fx)) {}
    while (b->yf.compare_exchange_weak(oyf, oyf + fy)) {}
}
```

possible contention

Counterintuitive: double the work?

```
void calculate_forces(int nbodies, Body *bodies) {
    cilk_for (int i = 0; i < nbodies; ++i) {
        for (int j = 0; j < nbodies; ++j) {
            // update the force vector on bodies[i] exerted
            // by bodies[j].
            if (i != j) {

                double fx, fy;
                calculate_force(&fx, &fy, bodies[i], bodies[j]);
                add_force(&bodies[i], fx, fy);
            }
        }
    }
}
```

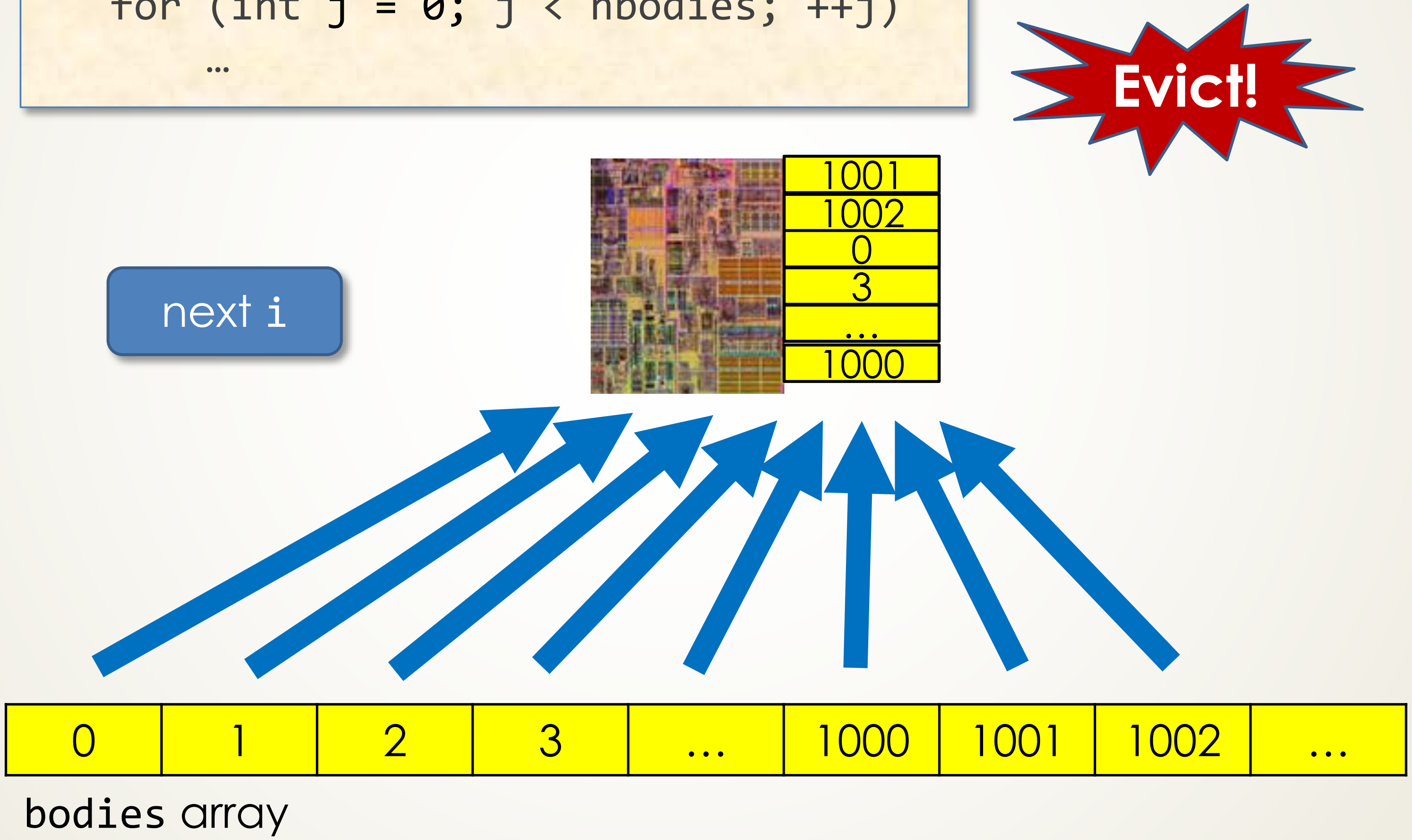
n(n – 1) applications of `calculate_force()`, again!

An elegant, cache-friendly approach

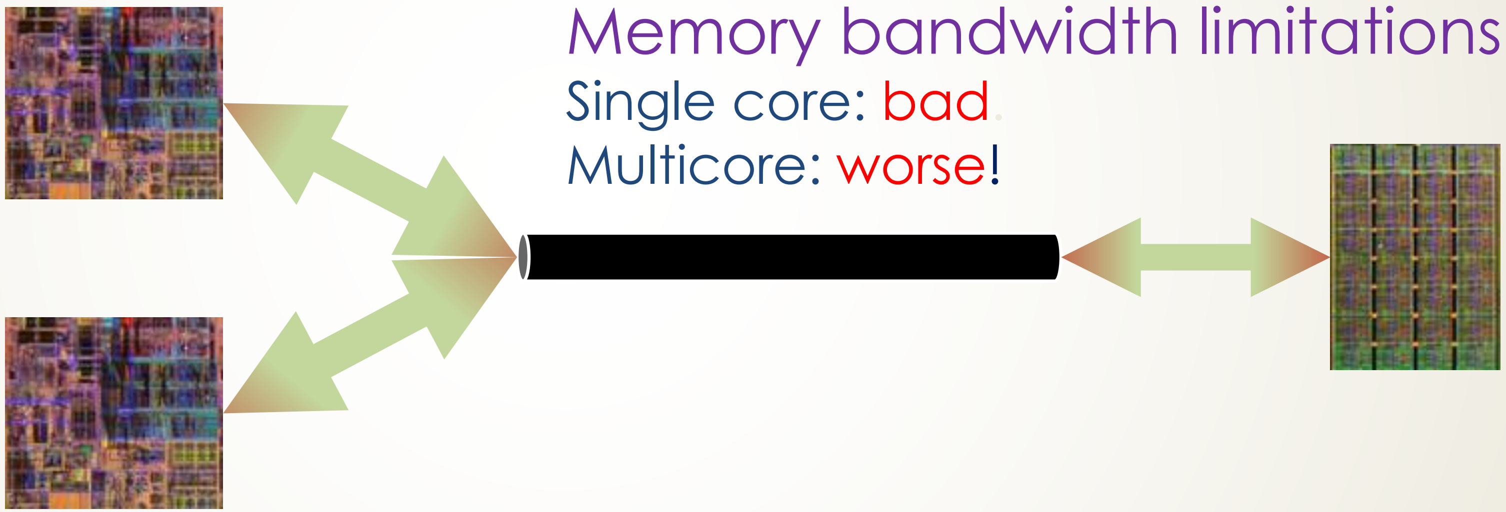
Introduction to cache-oblivious algorithms

The problem of poor cache locality

```
for (int i = 0; i < nbodies; ++i)
    for (int j = 0; j < nbodies; ++j)
        …
```



Cache locality is important for parallelism



2-D Tiling to improve cache locality
A brief reprise of `count_stars`

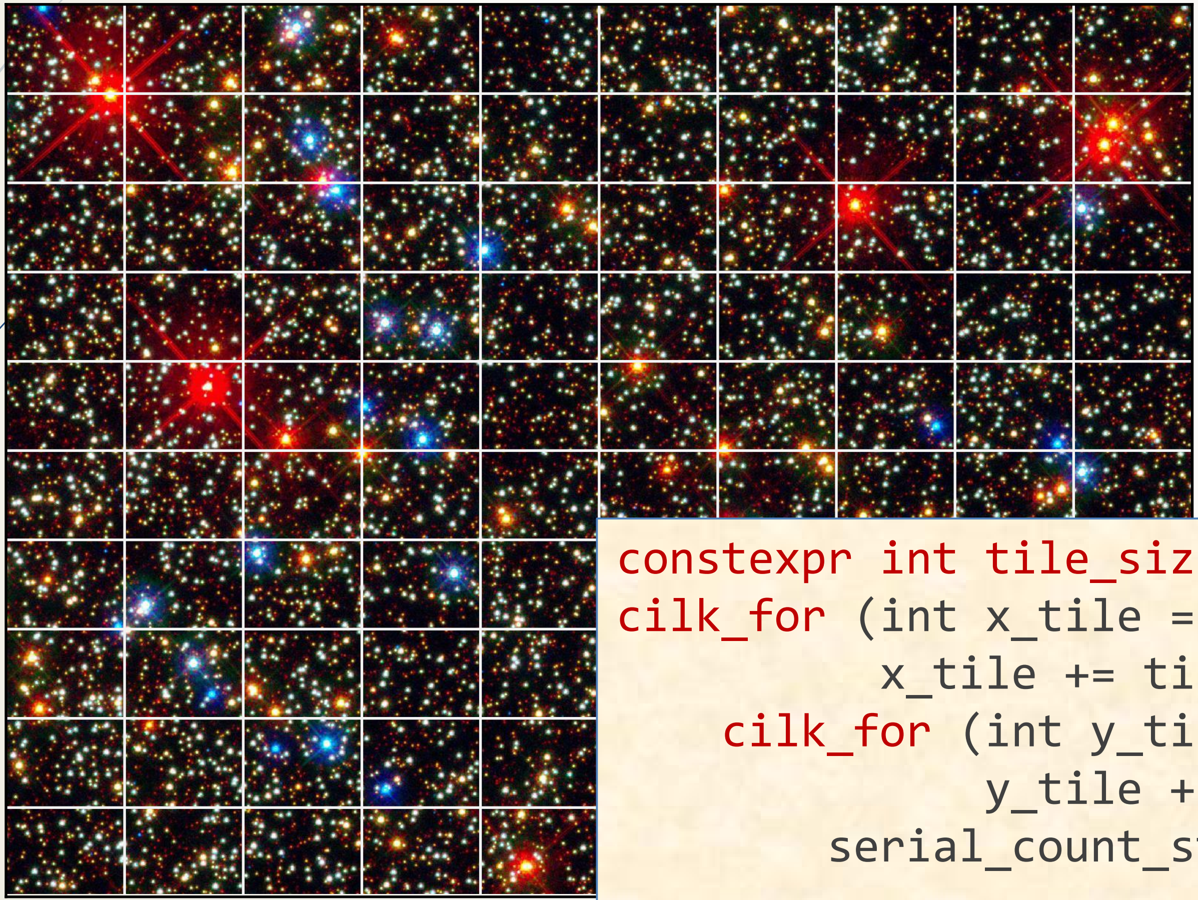


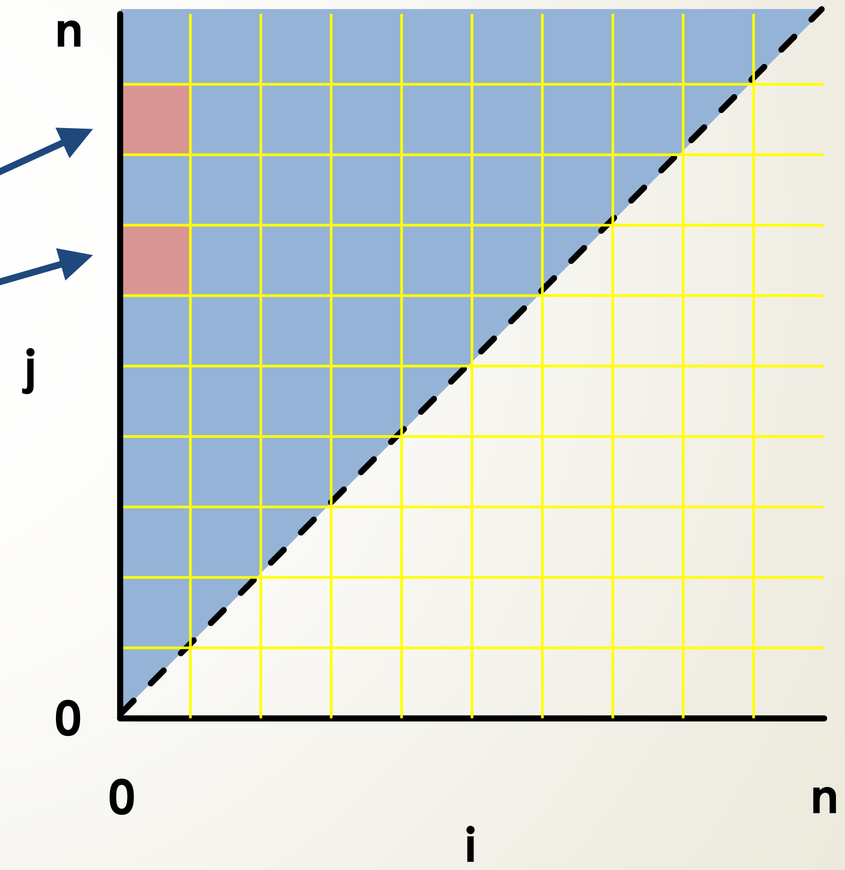
Photo courtesy NASA

```
constexpr int tile_size = 16;
cilk_for (int x_tile = 0; x_tile < img.width();
          x_tile += tile_size)
    cilk_for (int y_tile = 0; y < img.height();
              y_tile += tile_size)
        serial_count_stars(img, x_tile, tile_size,
                           y_tile, tile_size);
```

Can we tile the n-bodies problem, and return to the triangular computation?

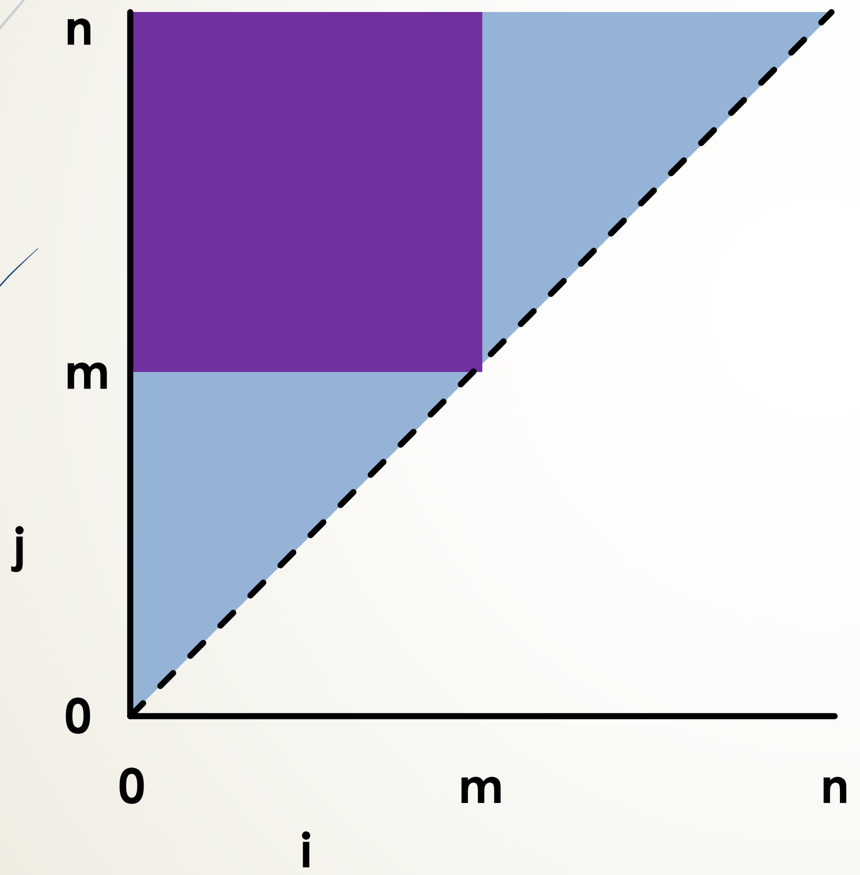
Yes, but not the same way as for count_stars

Same range of i values; not independent



Tiles cannot all be processed in parallel.

Cache-oblivious recursive tiling

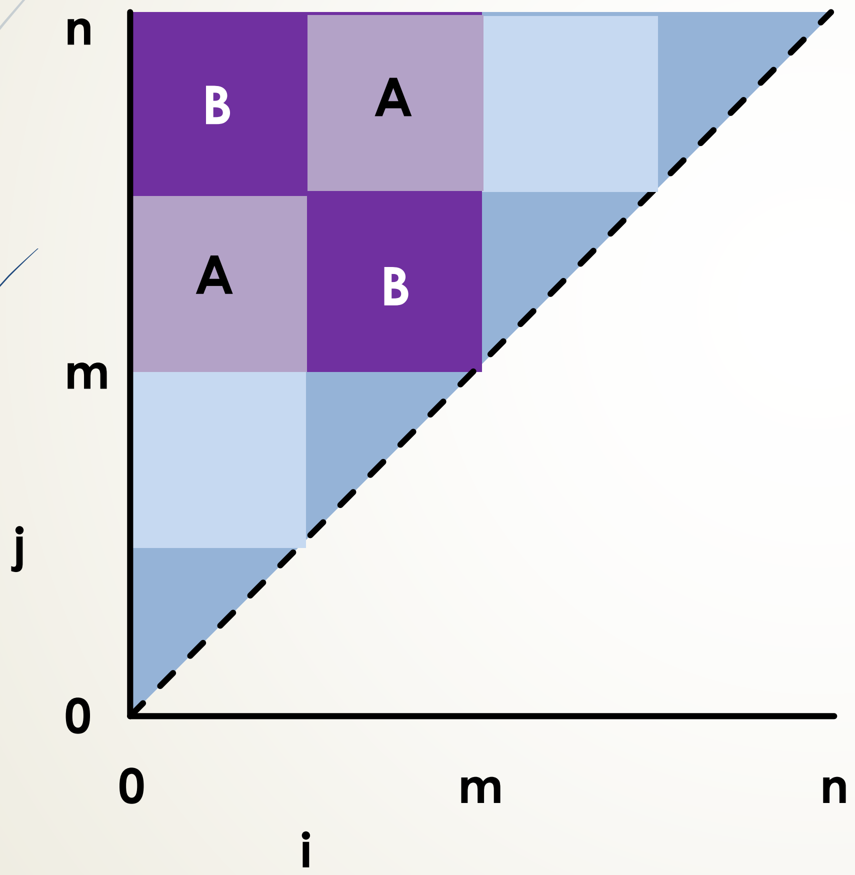


For two tiles to be computed in parallel, the i range of one must not overlap either the i or j range of the other.

The triangles are in parallel with each other.

Neither triangle is in parallel with the rectangle.

The next level of recursion



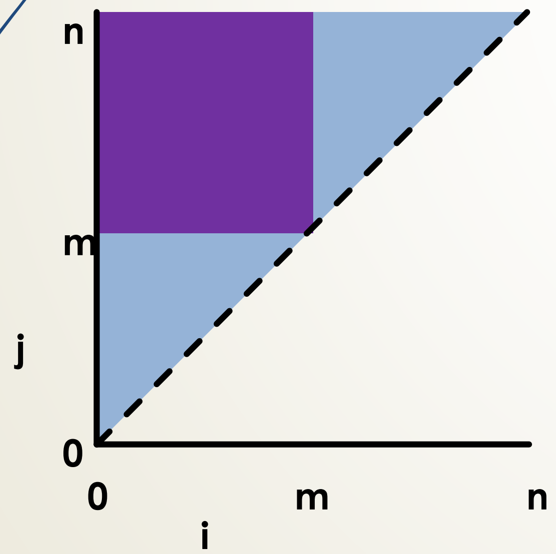
Each triangle can be recursively subdivided the same way, yielding the same parallelism at the next level.

Each rectangle can also be subdivided into four rectangles.

The rectangles marked A are in parallel with each other. The rectangles marked B are in parallel with each other (but not with the A rectangles).

Cache-oblivious n-bodies algorithm

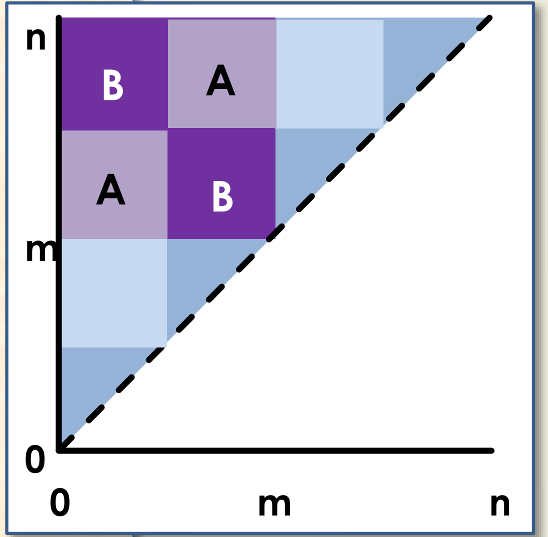
```
void calculate_forces(int nbodies, Body *bodies)
{
    triangle(0, nbodies, bodies);
}
```




```
// traverse the triangle n0 <= i <= j < n1
void triangle(int n0, int n1, Body *bodies)
{
    int dn = n1 - n0;
    if (dn > 1) {
        int nm = n0 + dn / 2;
        cilk_spawn triangle(n0, nm, bodies);
        triangle(nm, n1, bodies);
        cilk_sync;
        rect(n0, nm, nm, n1, bodies);
    }
}
```

Cache-oblivious n-bodies algorithm (continued)

```
// traverse the rectangle i0 <= i < i1,  j0 <= j < j1
void rect(int i0, int i1, int j0, int j1, Body *bodies) {
    int di = i1 - i0, dj = j1 - j0;
    if (di > 1 && dj > 1) {
        int im = i0 + di / 2, jm = j0 + dj / 2;
        cilk_spawn rect(i0, im, j0, jm, bodies); // A
        rect(im, i1, jm, j1, bodies);            // A
        cilk_sync;
        cilk_spawn rect(i0, im, jm, j1, bodies); // B
        rect(im, i1, j0, jm, bodies);            // B
        cilk_sync;
    } else if (di > 0 && dj > 0) {
        double fx, fy;
        calculate_force(&fx, &fy, bodies[i0], bodies[j0]);
        add_force(&bodies[i], fx, fy);
        add_force(&bodies[j], -fx, -fy);
    }
}
```



Coarsening to reduce overhead

```
// traverse the rectangle i0 <= i < i1,  j0 <= j < j1
void rect(int i0, int i1, int j0, int j1, Body *bodies) {
    int di = i1 - i0, dj = j1 - j0;
    constexpr int threshold = 16;
    if (di > threshold && dj > threshold) {
        int im = i0 + di / 2, jm = j0 + dj / 2;
        cilk_spawn rect(i0, im, j0, jm, bodies); // A
        rect(im, i1, jm, j1, bodies);            // A
        cilk_sync;
        cilk_spawn rect(i0, im, jm, j1, bodies); // B
        rect(im, i1, j0, jm, bodies);            // B
        cilk_sync;
    } else
      for (int i = i0; i < i1; ++i)
        for (int j = j0; j < j1; ++j) {
            double fx, fy;
            calculate_force(&fx, &fy, bodies[i], bodies[j]);
            add_force(&bodies[i], fx, fy);
            add_force(&bodies[j], -fx, -fy);
        }
}
```

recursive spawn is cheap, but not free

Serial loop is faster than recursion a the leaves.

Could we do more?

- The last version is fast and parallel, but you can almost *always* do more!
- The data structure for array-of-bodies is ill-suited for vectorization. This can and should be the topic of a whole talk at a future CppCon.
- Measure, measure, measure!
  - Using performance-analysis tools, we might find other bottlenecks.
  - Some of our logically-reasoned speed-ups might not work in practice on real hardware.

Summary

- Parallelism requires decomposing a problem into independent parts.
- Some creativity is required for all but the simplest algorithms.
- Even a correct parallel program can suffer from negative cache effects and contention.
- Measure and iterate!

More Information

- *A cute technique for avoiding certain race conditions*, Matteo Frigo, 2009, https://software.intel.com/en-us/articles/a-cute-technique-for-avoiding-certain-race-conditions

Thank You!