Lecture #12 - OpenMP Lab

AMath 483/583

Announcements

- Homework #1 Solutions posted (with comments in your private repos)

- Homework #2 Due Tomorrow - remember to push to remote. Does your code appear in

  http://github.com/uwhpsc-2016/homework2-githubusername

- Homework #3 Released tomorrow morning.

Lab: Numerical Integration

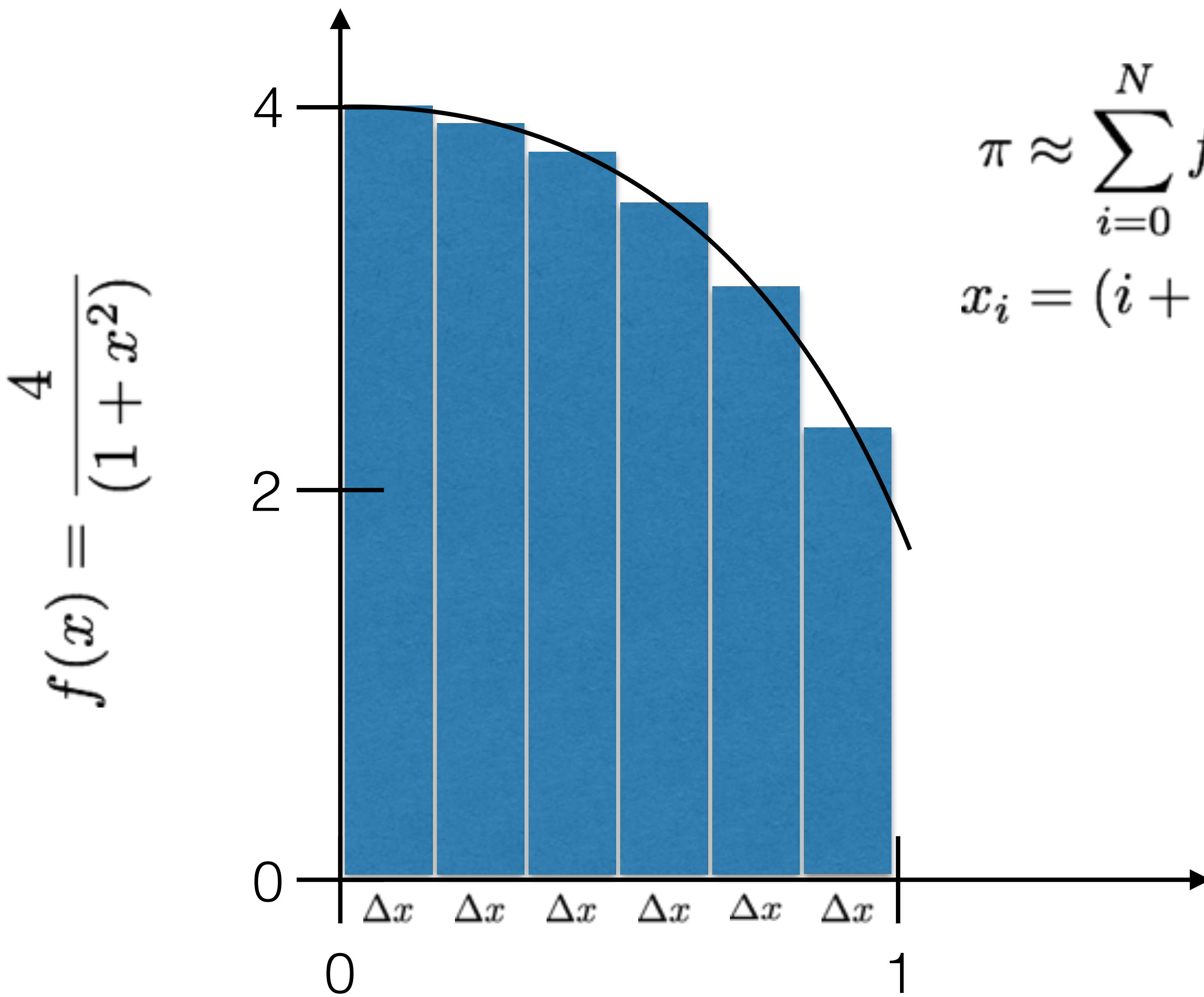
- Calculus:

$$\int_0^1 \frac{4\ \mathrm{d}x}{(1+x^2)} = \pi$$

- Want to computationally verify/approximate.

Riemann Sums

- Calculus: integral defined as limit of Riemann sum



$$\pi \approx \sum_{i=0}^{N} f(x_i)\Delta x$$

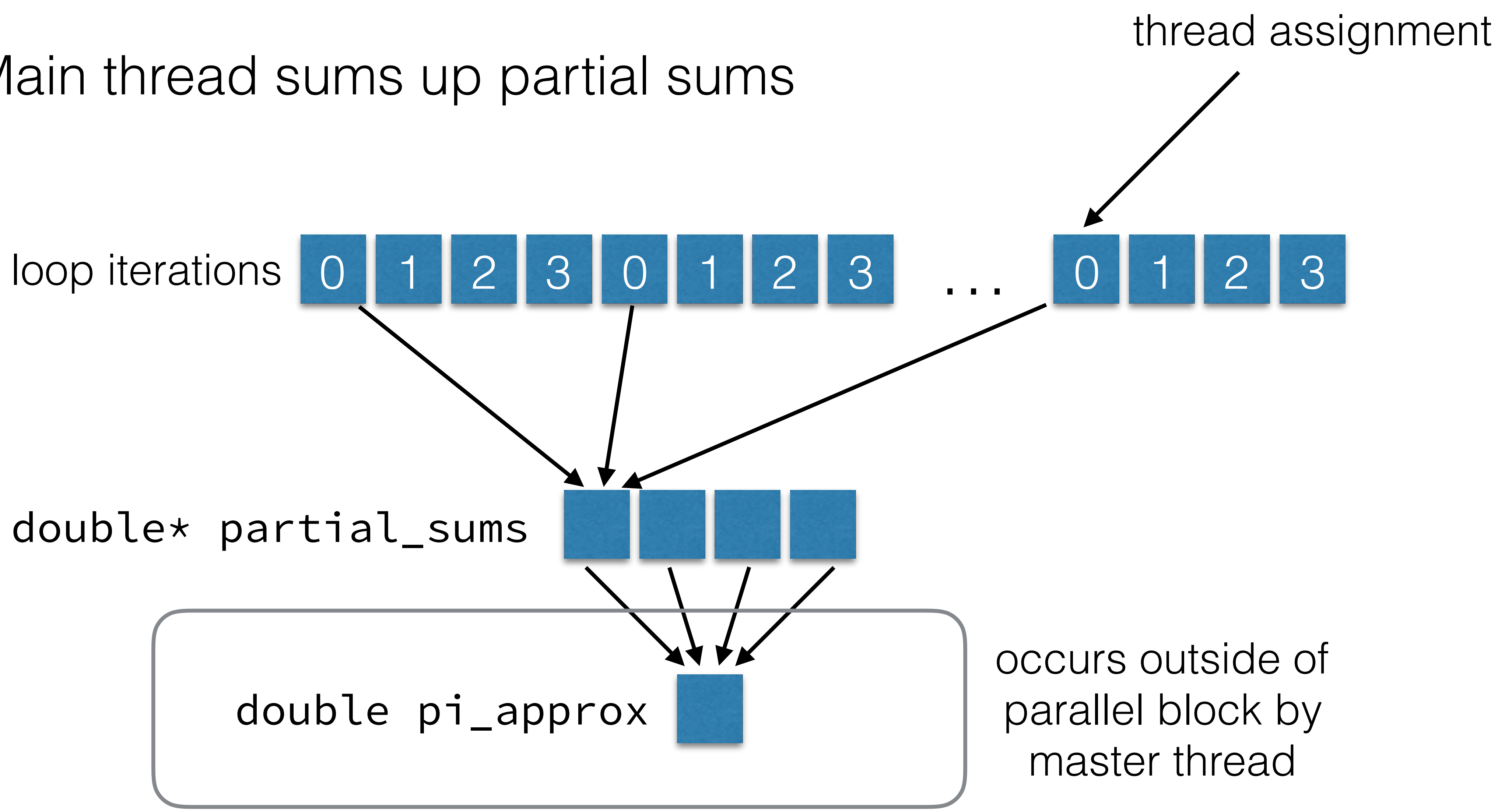
$$x_i = (i + 1/2)\Delta x$$

Demo

`pi_serial.c` — serial code for Riemann sums

Parallelizing - Attempt 1

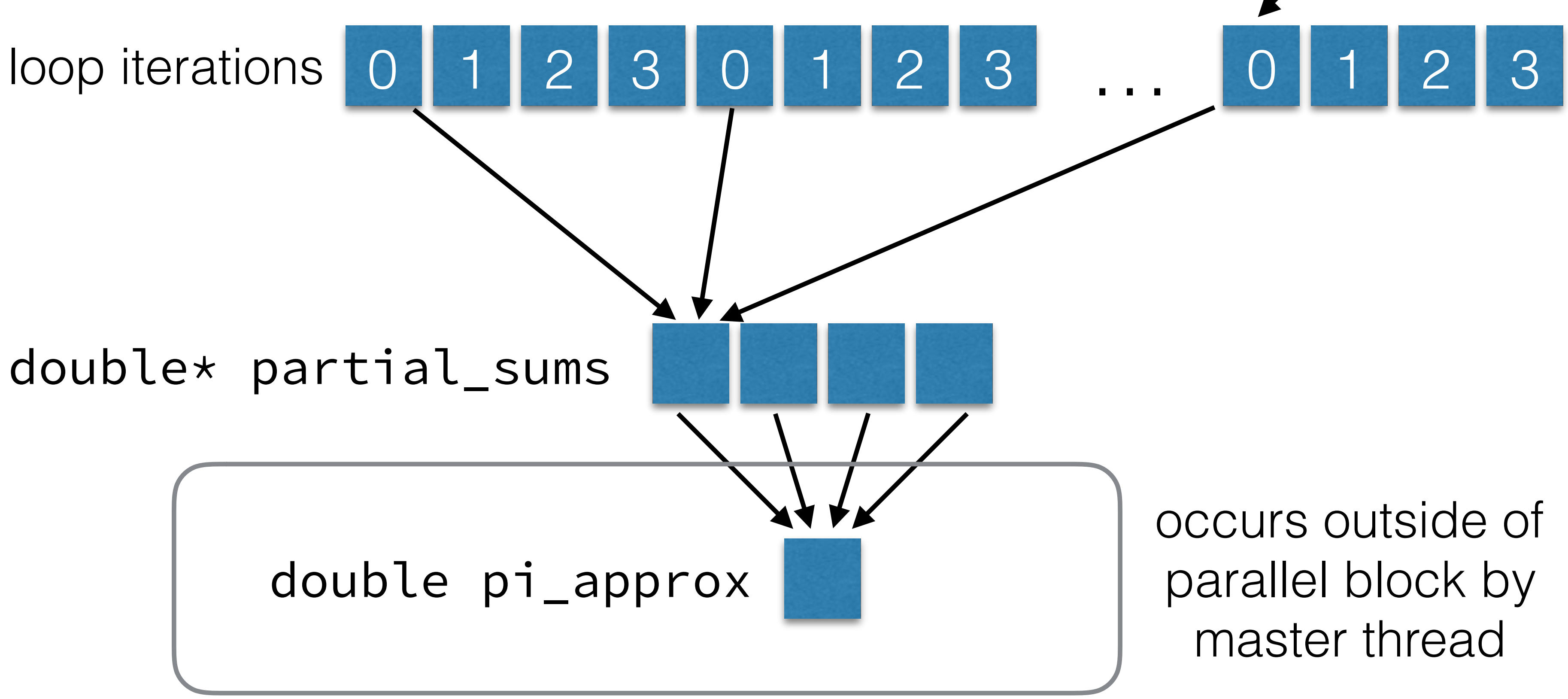
- Each thread only does a portion (manually specified) of the loop
- Store sub-sum in shared array element (sub-optimal)!
- Main thread sums up partial sums



Parallelizing -

- Each thread only does a portion (m
- Store sub-sum in shared array eleme
- Main thread sums up partial sums

We will use this pattern often when we talk about MPI.



Demo

`pi_parallel1` — first shot at parallelizing using only `omp parallel`

Observations

- **Poor time scaling**: using four threads is only about 1.2x faster! Should be closer to 4x.

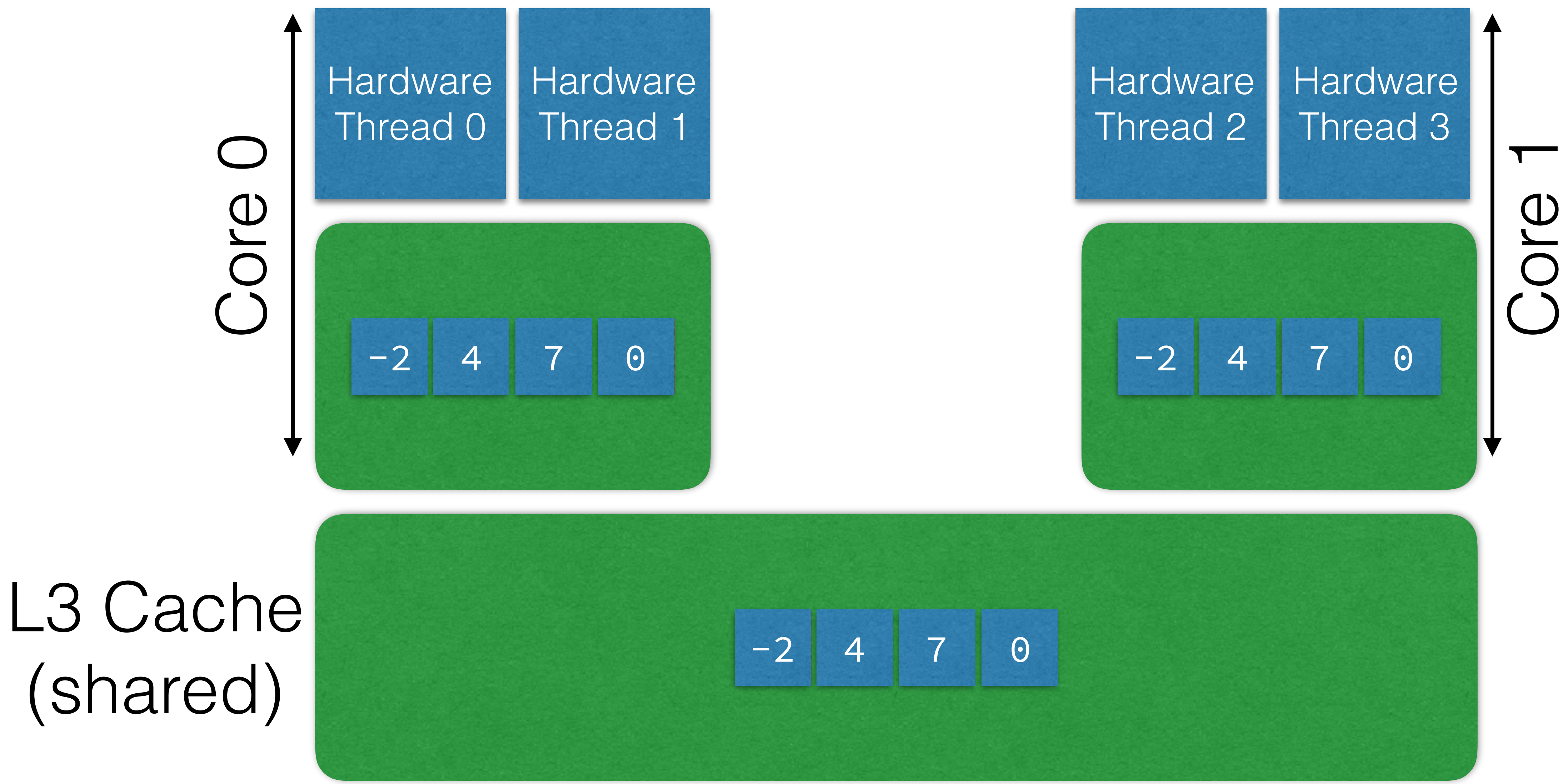
# Threads	Attempt #1 Time
1	0.54 s
2	0.56 s (!)
3	0.48 s
4	0.45 s (1.2x)

Observations

- **Poor time scaling**: in part, comes from cache-coherency on `double* partial_sums`.
    - *hack-ish solution*: “pad” array so that each partial sum lives in only one cache line (hardware dependent)
    - good argument for **not** using this approach

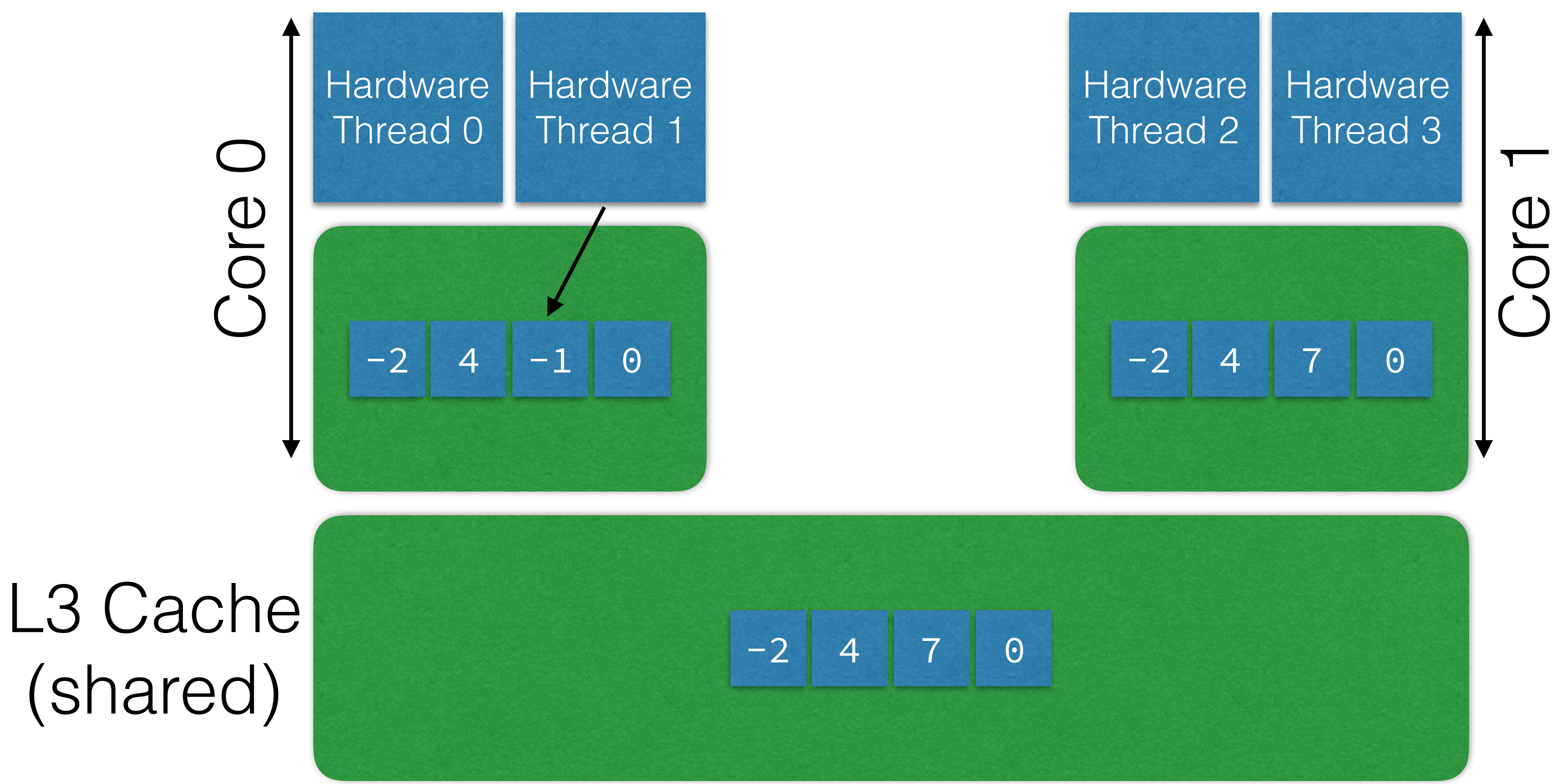
Brief Aside

- Cache coherency — hardware layer for keeping cache lines consistent



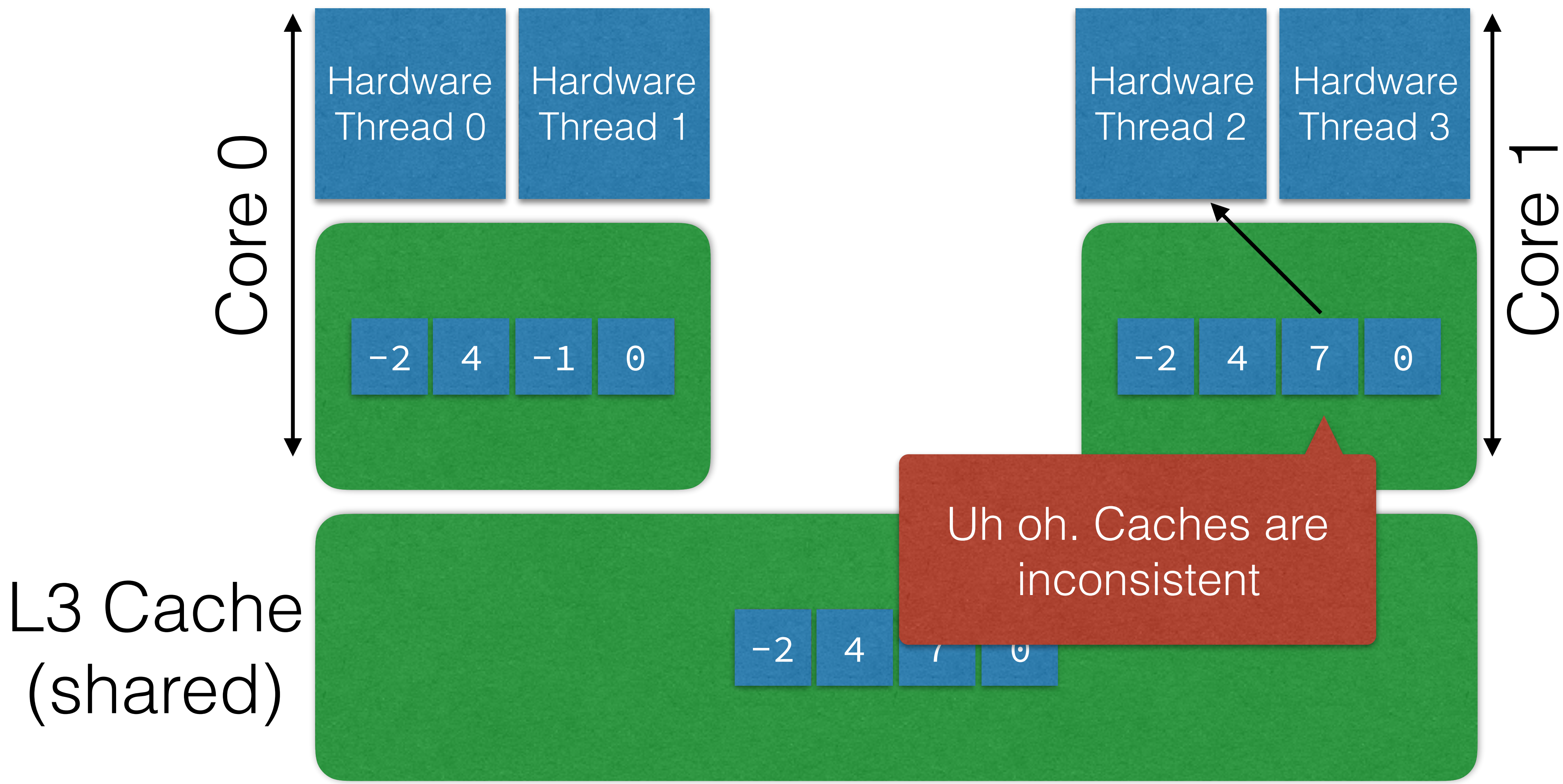
Brief Aside

- HW Thread 1 writes to cache copy



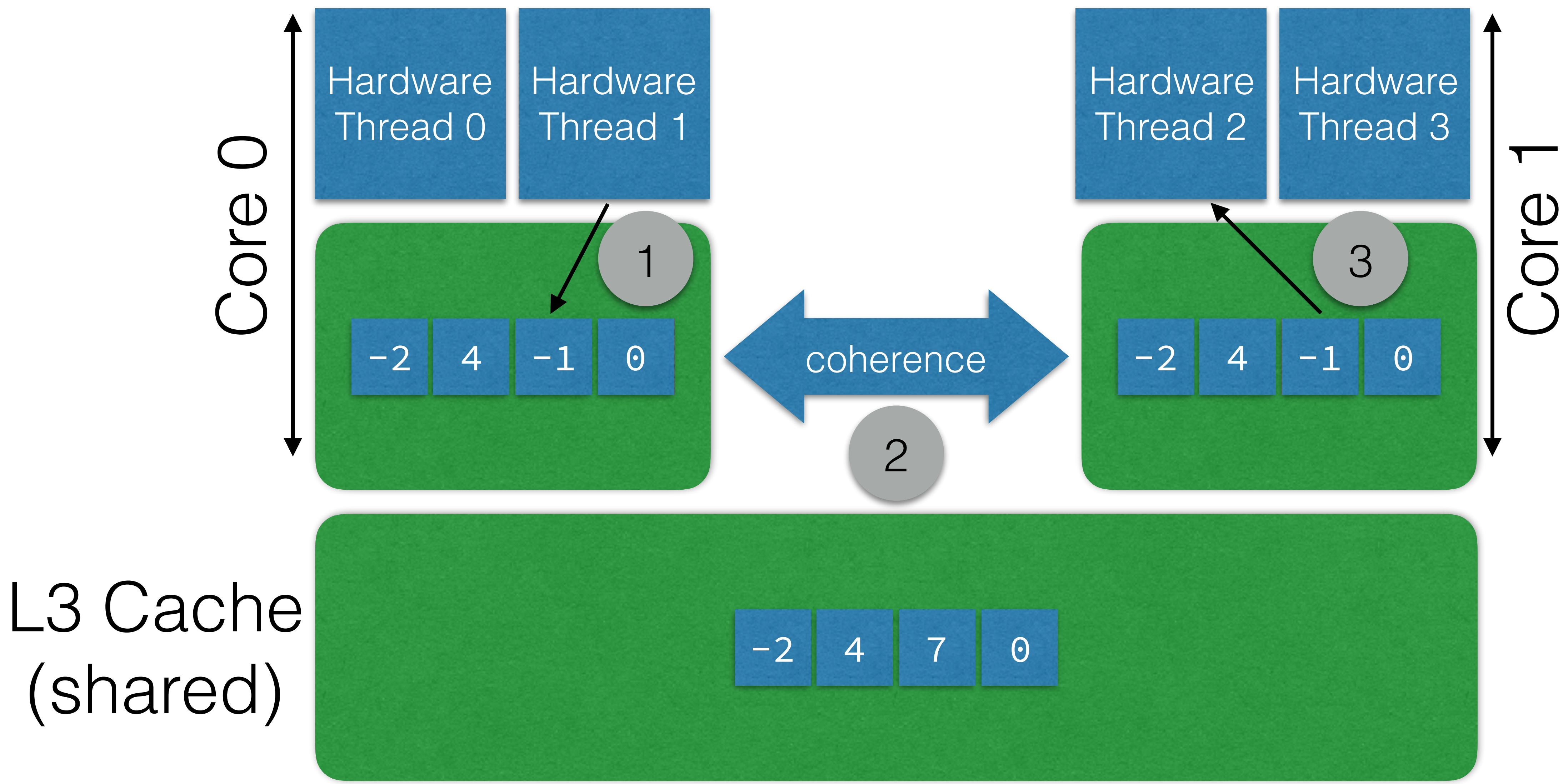
Brief Aside

- What if HW Thread 2 reads this updated value?



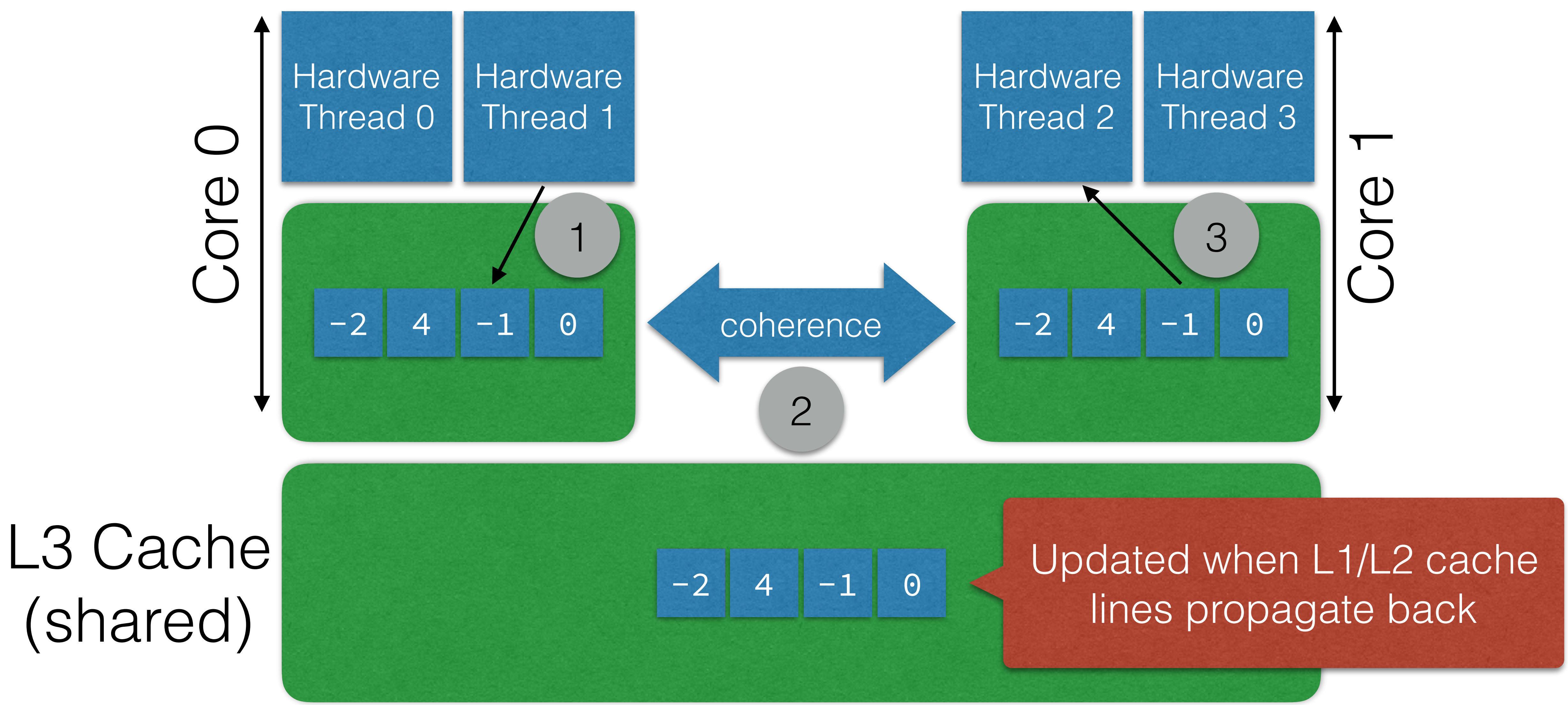
Brief Aside

- There exists a hardware layer to keep caches consistent. (“snooping”)



Brief Aside

- There exists a hardware layer to keep caches consistent. ("snooping")

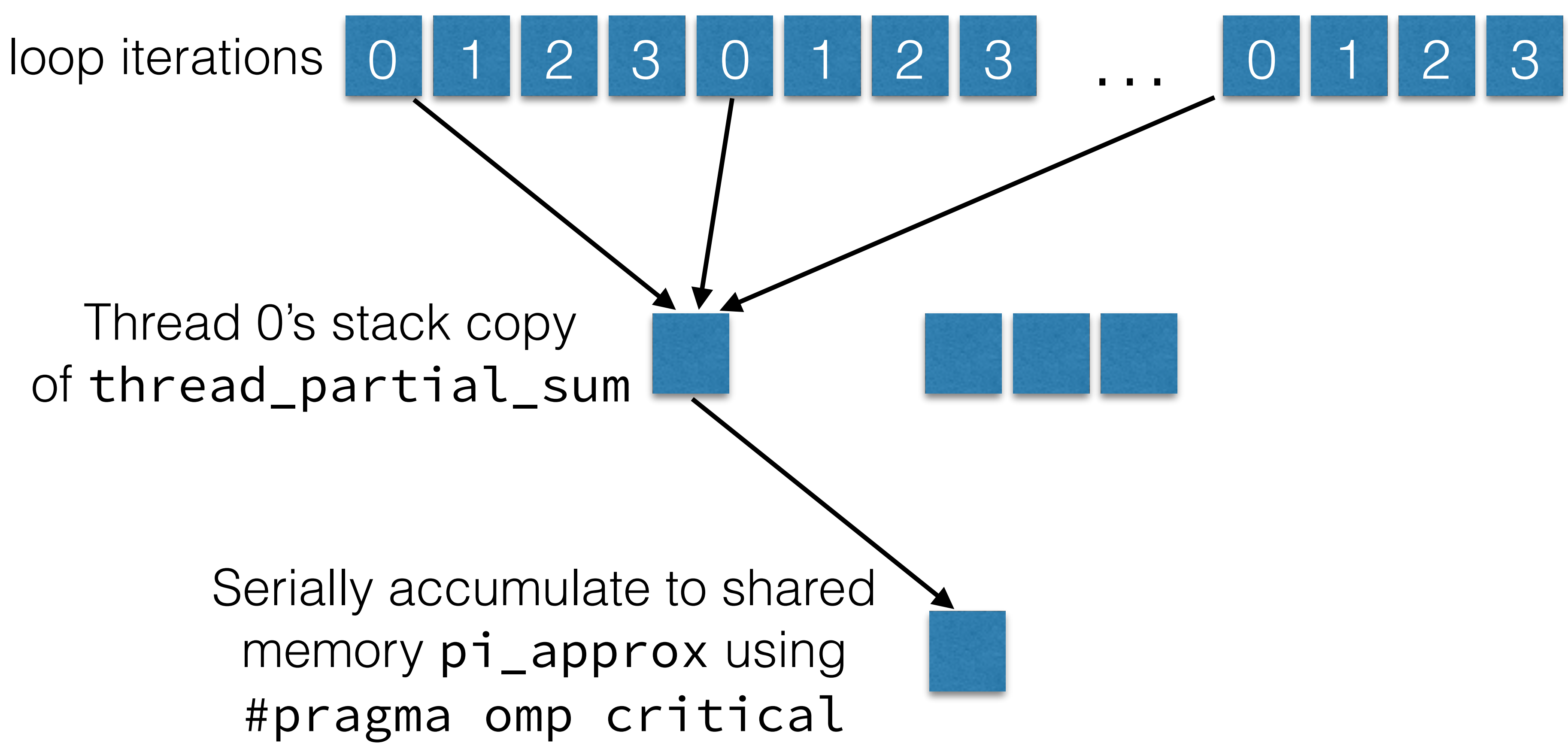


Observations

- **Poor time scaling**: in part, comes from cache-coherency on `double* partial_sums`.
  - *hack-ish solution*: “pad” array so that each partial sum lives in only one cache line (hardware dependent)
  - good argument for **not** using this approach

Parallelizing - Part 2

- Let's use `omp critical` to update `pi_approx`
- Store thread's partial sum in private var (on different cache line)



Demo

`pi_parallel2` — attempt 2 using `omp critical`/`omp atomic` to update shared memory location

Observations

- **Better Overall**: thread cache management / usage
- **Thread Scaling:** How to improve w.r.t. threads?

# Threads	Attempt #1 Time	Attempt #2 Time
1	0.54 s	0.38 s (!)
2	0.56 s	0.20 s
3	0.48 s	0.19 s
4	0.45 s (1.3x)	0.18 s (3.0x)

Parallelizing - Part 3

- Manual loop sharing is error-prone: use `omp for`
- Any naive speedups?
- Any speedups with chunk size and scheduling?

Demo

`pi_parallel3` — replace manual loop sharing with `omp for`

Observations

- **OpenMP Optimizations**: let it deal with worksharing
- **Chunk Size Optimization**: maybe…on diff. problem

# Threads	Attempt #1 Time	Attempt #2 Time	Attempt #3 Time
1	0.54 s	0.38 s	0.32 s
2	0.56 s	0.20 s	0.16 s
3	0.48 s	0.19 s	0.16 s
4	0.45 s (1.3x)	0.18 s (3.0x)	0.15 s (3.6x)

Parallelizing - Part 4

- Numerical integration is a type of reduction operation

```
for (…)
  pi_approx += expression
```

- `#pragma omp for reduction(op:list)`

Demo

`pi_parallel4` — use `reduction(op:list)` clause

Observations

- **Reduce:** no easily measurable speedup, but much cleaner than previous attempts

# Threads	Attempt #1 Time	Attempt #2 Time	Attempt #3 Time	Attempt #4 Time
1	0.54 s	0.38 s	0.32 s	0.32 s
2	0.56 s	0.20 s	0.16 s	0.16 s
3	0.48 s	0.19 s	0.16 s	0.15 s
4	0.45 s (1.3x)	0.18 s (3.0x)	0.15 s (3.6x)	0.15 s (3.6x)

Observations

- Simple operation —> difficult to speed up further
- More complicated —> many avenues to take for "performance tuning"
    - *loop chunking* takes advantage of *cache locality*
    - *private* vs. *shared* variables
    - *memory accesses* still important when working with data arrays (hint: Homework #3)

Programmer Time

- **Programmer time is valuable**: given original code what is quickest way to parallelize?

```
long N = 100000000;
double pi_approx = 0;
double xi;
double dx = 1.0 / (double) N;



for (int i=0; i<N; ++i)
  {
    xi = (i + 0.5)*dx;
    pi_approx += 4.0/(1.0 + xi*xi) * dx;
  }
```

Programmer Time

- Programmer time is valuable: given original code what is quickest way to parallelize?

```
long N = 100000000;
double pi_approx = 0;
double xi;
double dx = 1.0 / (double) N;

#pragma omp parallel for private(xi)      \
  reduction(+:pi_approx) num_threads(4)
for (int i=0; i<N; ++i)
  {
    xi = (i + 0.5)*dx;
    pi_approx += 4.0/(1.0 + xi*xi) * dx;
  }
```

Programmer Time

- Programmer time
  what is quickest

```
long N = 100
double pi_ap
double xi;
double dx =

#pragma omp parallel for private(xi)     \
  reduction(+:pi_approx) num_threads(4)
for (int i=0; i<N; ++i)
  {
    xi = (i + 0.5)*dx;
    pi_approx += 4.0/(1.0 + xi*xi) * dx;
  }
```

Beware race conditions!

If `xi` were shared then multiple threads will be setting the value of the same `xi`.

Solution: give each thread a private copy.

Programmer Time

- Programmer tim
what is quickest

Final comment: why not `private(xi,dx)`?

`private()` creates a new local copy.
This copy is ***uninitialized***!

```
long N = 100
double pi_ap
double xi;
double dx =

#pragma omp parallel for private(xi)      \
  reduction(+:pi_approx) num_threads(4)
for (int i=0; i<N; ++i)
  {
    xi = (i + 0.5)*dx;
    pi_approx += 4.0/(1.0 + xi*xi) * dx;
  }
```