CSE502: Foundations of Parallel Programming

Lecture 10: Loop-level Parallelism, False Sharing

Vivek Kumar

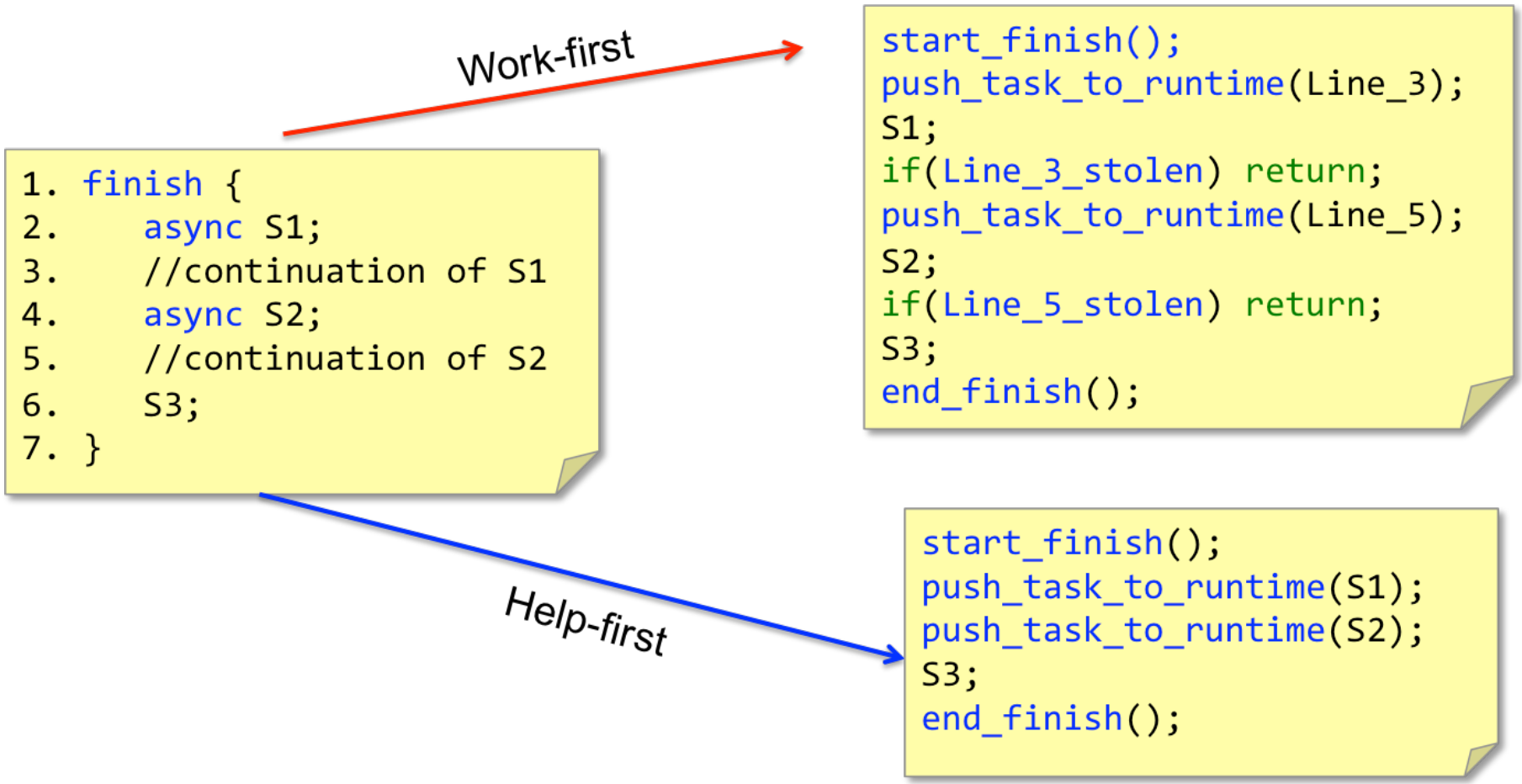
Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Class

- Types of work-stealing
- Work-stealing overheads



Today's Class

- Loop level parallelism
- False sharing

Observations on finish-for-async Version of Parallel Matrix Multiplication

- finish and async are general constructs, and are not specific to loops
- Loops in sequential version of matrix multiplication are "perfectly nested"
  - e.g., no intervening statement between "for(i = ...)" and "for(j = ...)"
- The ordering of loops nested between finish and async is arbitrary
  - They are parallel loops and their iterations can be executed in any order

Source: https://wiki.rice.edu/confluence/download/attachments/24426821/comp322-s16-lec11-slides-v1.key.pdf?version=1&modificationDate=1483206144935&api=v2

Loop Level Parallelism

```
void foo() {
  for (uint64_t i=0; i<SIZE; i++) {
    S(i); // can execute in parallel for all i
  }
}
```

- Case-1: S(i) is doing the same amount of computation in each iteration
  - Static decomposition is applied and each worker receives equal sized chunk of the for-loop
    - Pros:
      - Perfect load balancing
        - » total tasks = total workers (assuming SIZE%num_workers==0)
    - Cons
      - Programmer has to modify the sequential code for avoiding tasking overheads and for achieving perfect load balancing
        - » Hampers productivity as no serial elision

Loop Level Parallelism

```
void foo() {
  for (uint64_t i=0; i<SIZE; i++) {
    S(i); // can execute in parallel for all i
  }
}
```

- Case-2: S(i) is **NOT** doing the same amount of computation in each iteration
  - Static decomposition still possible but the programmer has to divide total iterations into “small chunks” (tiling)
    - Pros:
      - Perfect load balancing is still possible if total_chunks >> total_workers
    - Cons
      - Programmer has to modify the sequential code for avoiding tasking overheads and for achieving perfect load balancing
        - » Hampers productivity as no serial elision
      - If S(i) does not create any async then there is a single producer producing large number of tasks and there are multiple consumer
        - » How does it affects work-first and help-first scheduling ?

“forasync” – Construct for Harnessing Loop Level Parallelism in HClib

```
void foo() {
  loop_domain_t loop = {0, SIZE, 1, tile_size};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_RECURSIVE);
  });
}
```

- loop_domain_t loop = {lowBound, highBound, loopStride, tileSize};
- forasync1D(loop_domain_t* loop, lambda_function, mode);

- loop_domain_t loop[2] = { {lowBound0, highBound0, loopStride0, tileSize0}, {…} };
- forasync2D(loop_domain_t* loop, lambda_function, mode);

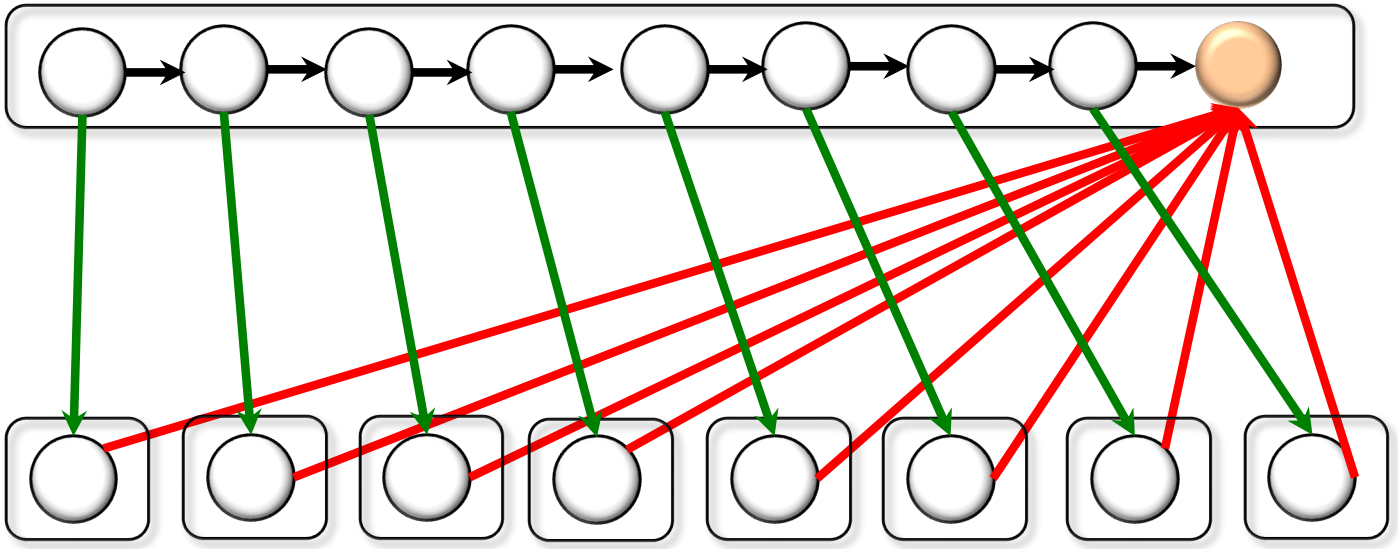
- loop_domain_t loop[3] = { {lowBound0, highBound0, loopStride0, tileSize0}, {…}, {…} };
- forasync3d(loop_domain_t* loop, lambda_function, mode);

- mode
  - FORASYNC_MODE_RECURSIVE: recursively partition total iteration space until “tileSize” is reached
  - FORASYNC_MODE_FLAT: chunk iterations into blocks of length “tileSize”

FORASYNC_MODE_FLAT

```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_FLAT);
  });
}
```

Work = O(n)
CPL = O(n)

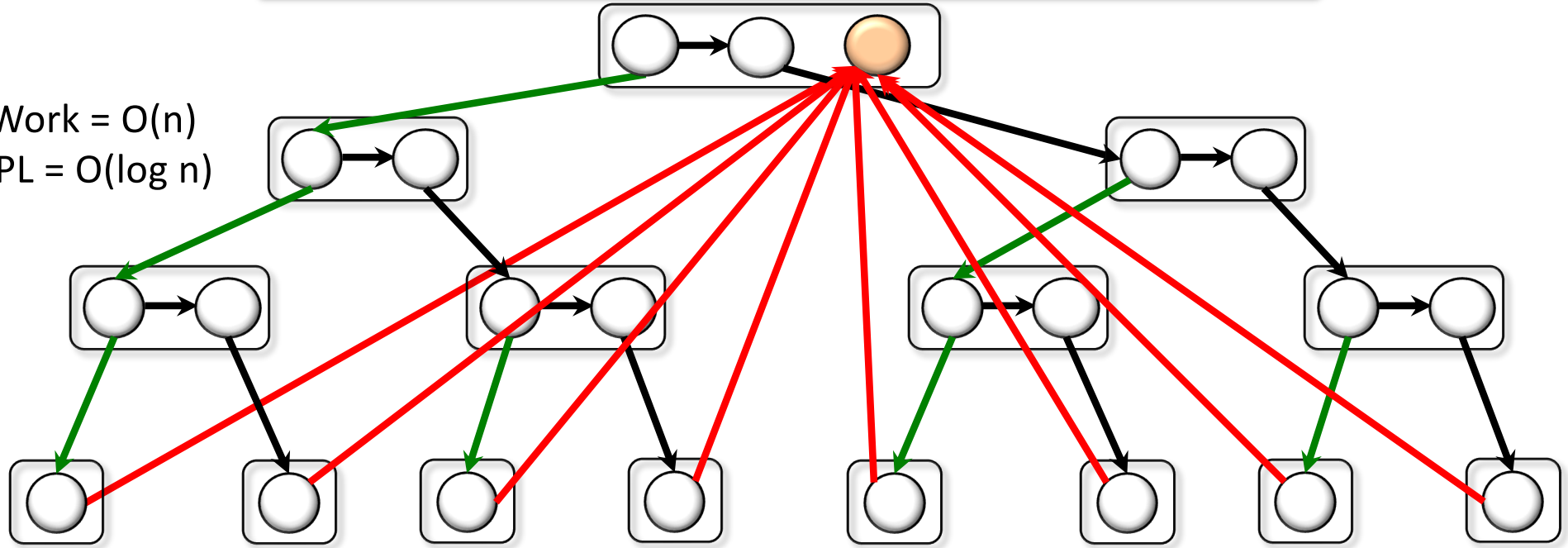


Source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

FORASYNC_MODE_RECURSIVE (Divide-and-Conquer)

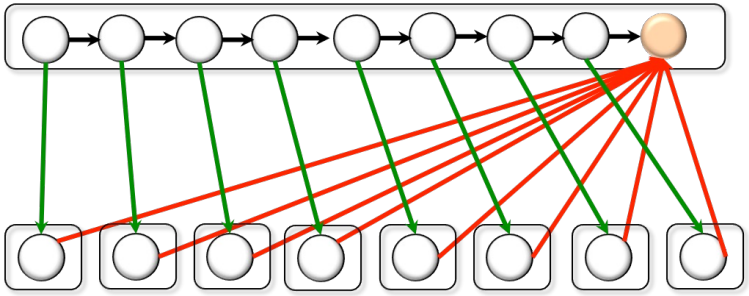
```
void foo() {
  loop_domain_t loop = {0, 8, 1, 1};
  finish([&]() {
    forasync1D (&loop, [=](int i) {
      S(i); // can execute in parallel for all i
    }, FORASYNC_MODE_RECURSIVE);
  });
}
```

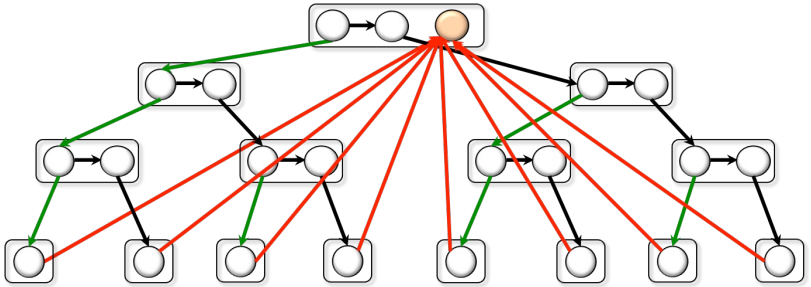
Work = O(n)
CPL = O(log n)



Modification of source: http://classes.engineering.wustl.edu/cse539/web/lectures/lec02_parallelism.pdf

Question





- In a possible scenario, HClib (help-first scheduling) is initialized with a single worker. This worker is having a deque of size **N**. You have to launch a forasync1D computation with loop_domain_t = {0, **2N**, 1, 1}. Will there will be a deque overflow?
  - Yes if FORASYNC_MODE_FLAT
  - No if FORASYNC_MODE_RECURSIVE

Why ?

Parallelizing Matrix Multiplication

```
for (uint64_t i=0; i<N; i++) {
  for (uint64_t j=0; j<N; j++) {
    C[i][j] = 0;
  }
}

for (uint64_t i=0; i<N; i++) {
  for (uint64_t j=0; j<N; j++) {
    for (uint64_t k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- Sequential matrix multiplication (NxN)
  - How to parallelize using forasync?

Parallelizing Matrix Multiplication

```
loop_domain_t loop[2] = { {0, N, 1, tile}, {0, N, 1, tile} };


  forasync2D (loop, [=] (int i, int j) {
      C[i][j] = 0;
  }, FORASYNC_MODE_RECURSIVE);


  forasync2D (loop, [=] (int i, int j) {
      for (uint64_t k=0; k<N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
  }, FORASYNC_MODE_RECURSIVE);
```

Data Race !!

- Parallel matrix multiplication (NxN)
  - forasync2D

Parallelizing Matrix Multiplication

```
loop_domain_t loop[2] = { {0, N, 1, tile}, {0, N, 1, tile} };

finish {[&]() {
  forasync2D (loop, [=] (int i, int j) {
      C[i][j] = 0;
  }, FORASYNC_MODE_RECURSIVE);
});

finish {[&]() {
  forasync2D (loop, [=] (int i, int j) {
      for (uint64_t k=0; k<N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
  }, FORASYNC_MODE_RECURSIVE);
});
```

- Parallel matrix multiplication (NxN)
  - forasync2D
    - High productivity achieved !

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec11/

Today's Class

- Loop level parallelism
- False sharing

Revisiting the ArraySum Program We Saw in Lecture 05 …

Sequential ArraySum

```
double array[SIZE]; // initialized with random numbers

void ArraySum() {
  double sum = 0;
  for (uint64_t i=0; i<SIZE; i++) {
    sum += array[i];
  }
}
```


Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec09/

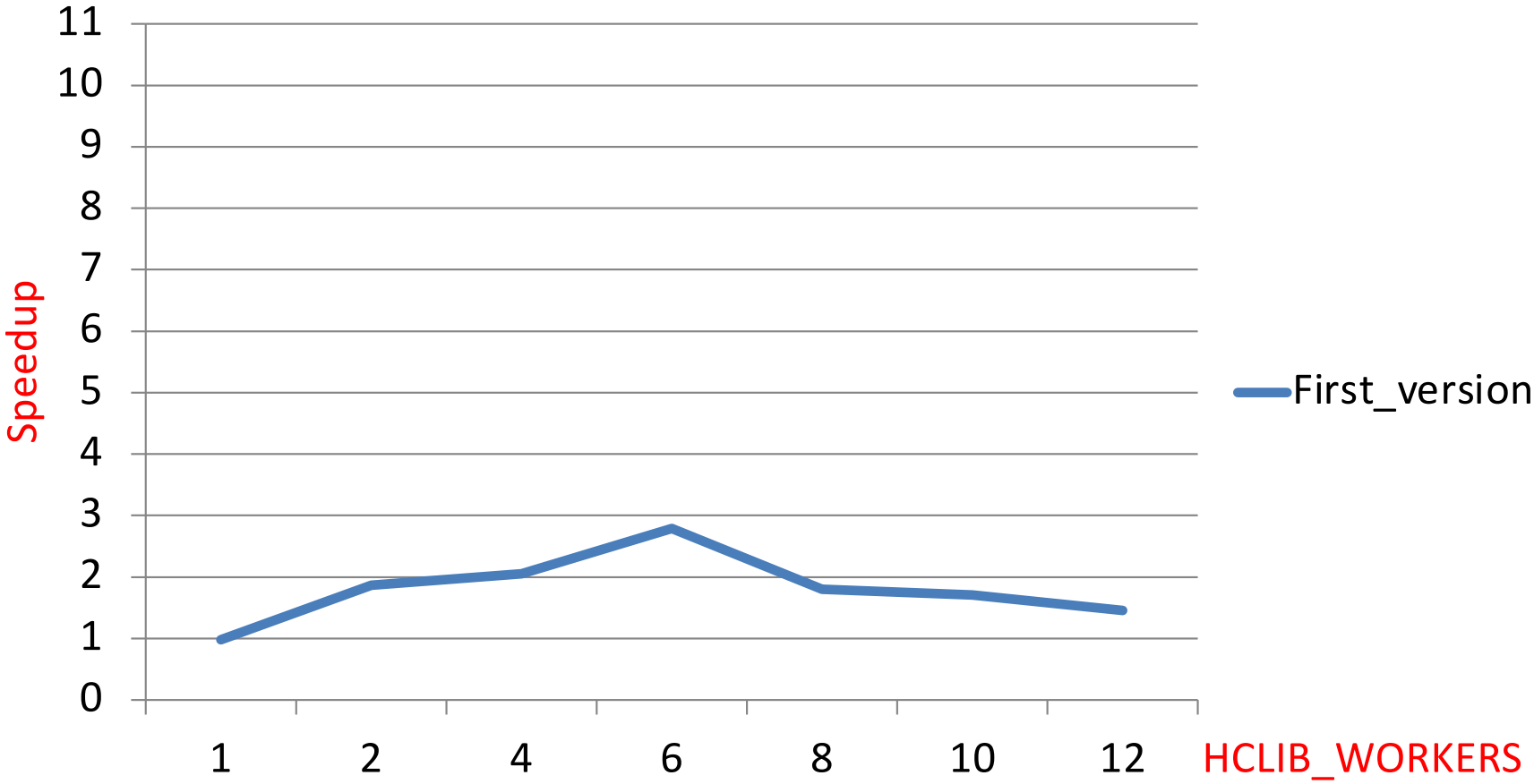
Parallel ArraySum Using async-finish

```
double array[SIZE]; // initialized with random numbers

void ArraySum() {
  double sum[num_workers()]; // zero initialized
  uint64_t chunkSize = SIZE / num_workers();
  finish {
    for (int worker=0; worker<num_workers(); worker++) {
      async {
        uint64_t start = worker * chunkSize;
        uint64_t end = start + chunkSize;
        for (uint64_t i=start; i<end; i++) {
          sum[worker] += array[i];
        }
      }
    }
  }
  double result = 0;
  for (int worker=0; worker<num_workers(); worker++)
    result += sum[worker];
}
```

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec09/

Speedup Analysis



Speedup of parallel ArraySum over its sequential implementation on a 12 core Intel E5-2667 processor

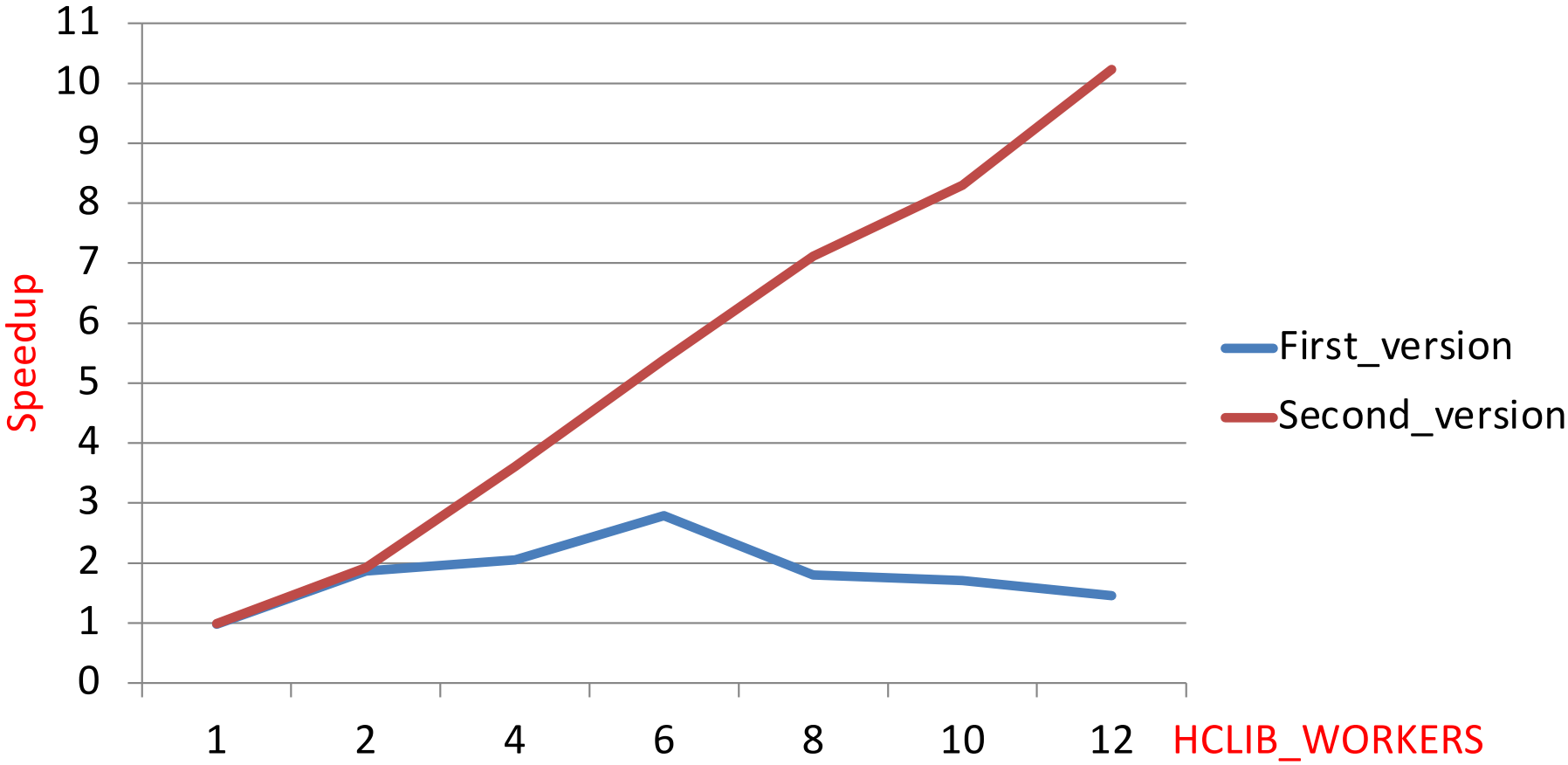
Parallel ArraySum Using async-finish (with a different approach)

```
double array[SIZE]; // initialized with random numbers

void ArraySum() {
  double sum[num_worker()]; // zero initialized
  uint64_t chunkSize = SIZE / num_workers();
  finish {
    for (int worker=0; worker<num_workers(); worker++) {
      async {
        uint64_t start = worker * chunkSize;
        uint64_t end = start + chunkSize;
        double my_local_sum = 0;
        for (uint64_t i=start; i<end; i++) {
          my_local_sum += array[i];
        }
        sum[worker] = my_local_sum;
      }
    }
  }
  double result = 0;
  for (int worker=0; worker<num_workers(); worker++)
    result += sum[worker];
}
```

Code available on github: https://github.com/vivkumar/cse502/blob/master/hclib/test/lec09/

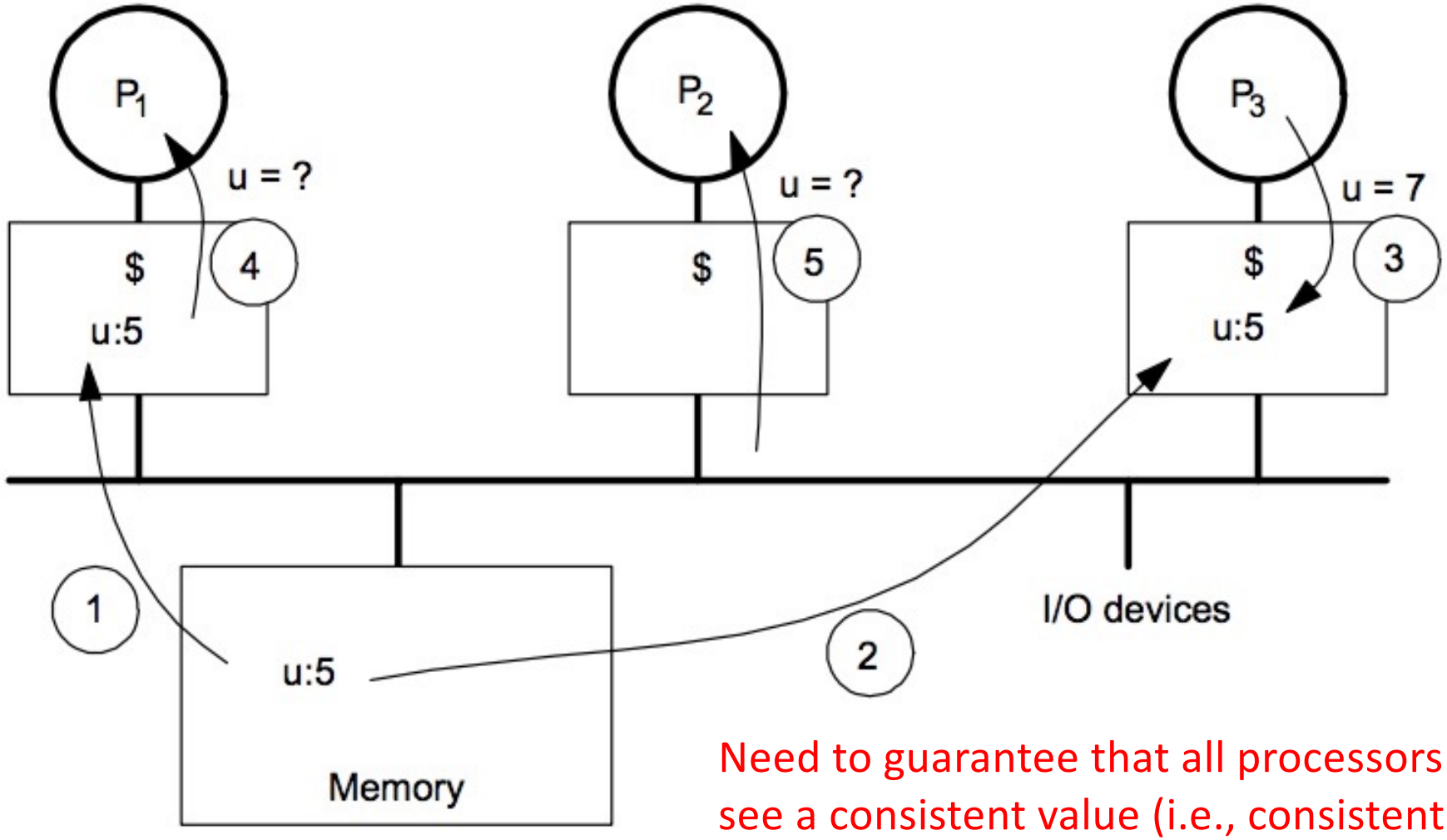
Speedup Analysis



Speedup of two different implementations of parallel ArraySum over its sequential implementation on a 12 core Intel E5-2667 processor

Source: https://youtu.be/NJ46OXN45eU

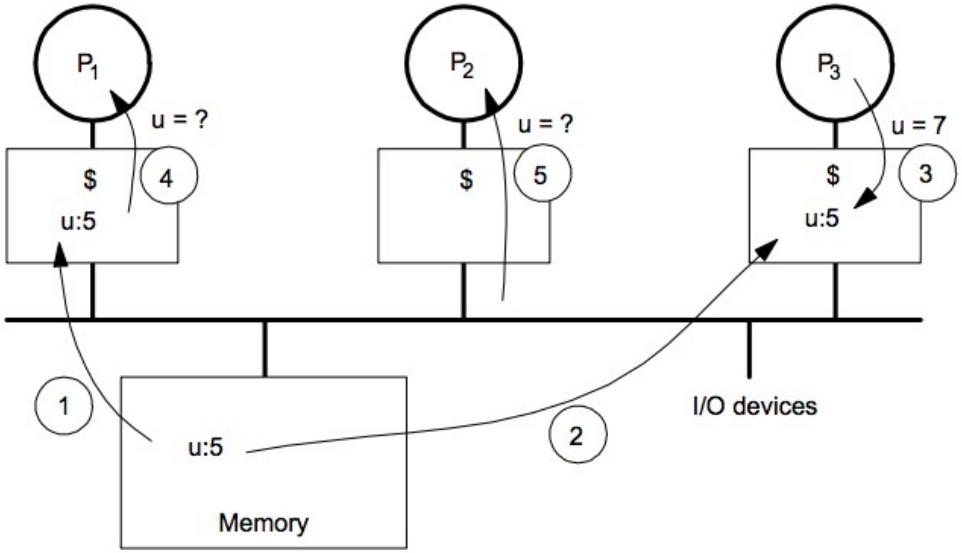
The Cache Coherence Problem



Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location

Slide Source: CMU

Definition of Coherence



Dividing a memory location's lifetime into hypothetical epochs, where

1. Every epoch has a either a single writer or multiple readers (**write serialization:** ex: if P3 observes u having value 1 and then 2, then no processor can observe u having value 2 before 1).
2. The value of the memory location propagates from the end of one epoch to the beginning of the next epoch (**value propagation:** the new value eventually gets to other cores. P3. writes 7 to u. This value is propagated to next epochs, where P1 and P2 reads u to find its updated value as 7).

A cache coherence protocol maintains these two invariants.

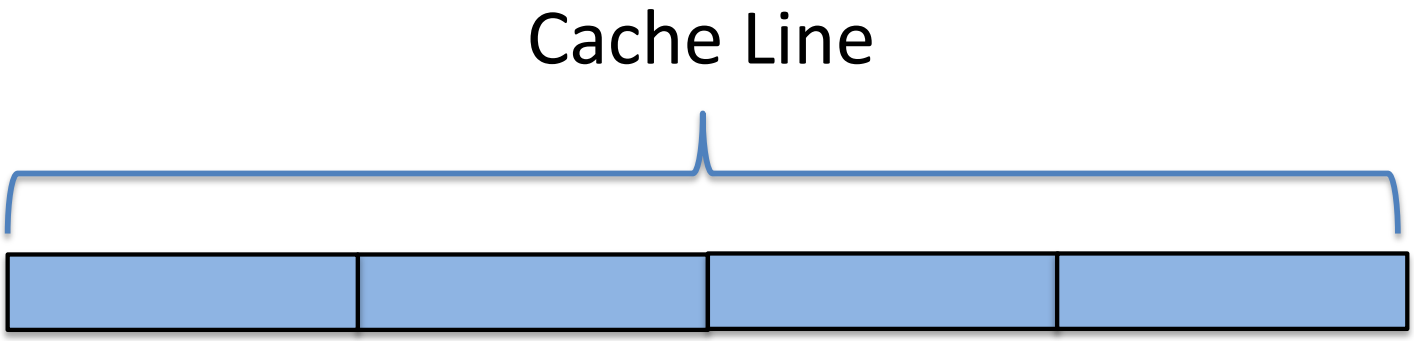
(The granularity of coherence is a cache line size.)

Source: http://classes.engineering.wustl.edu/cse539/web/lectures/mem_coherence_snoop.pdf

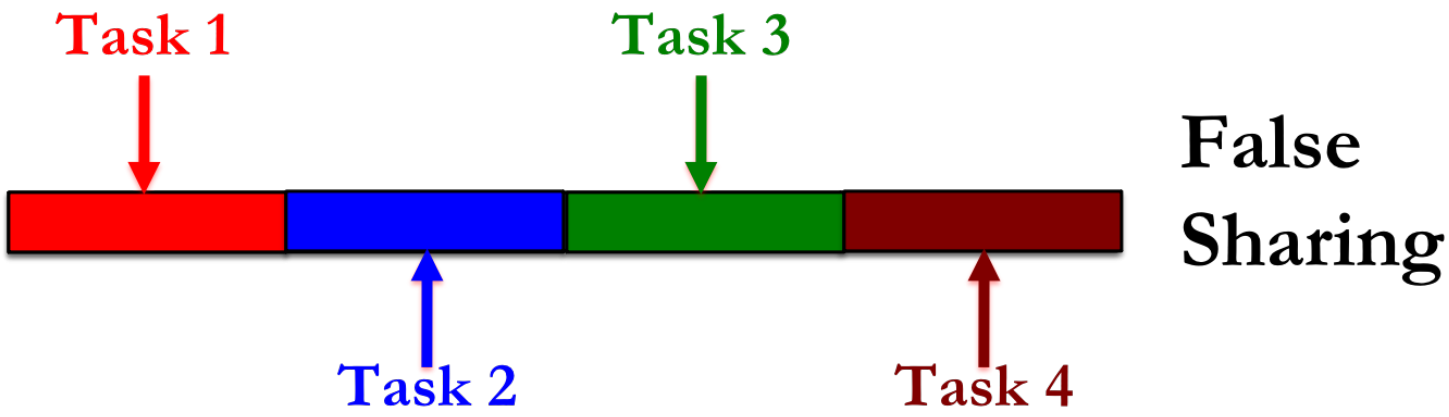
Hardware Cache Coherence

- Basic idea
  - Processor/cache broadcasts its write/update to a memory location to all other processors
  - Other caches that that memory address either update or invalidates its local copy

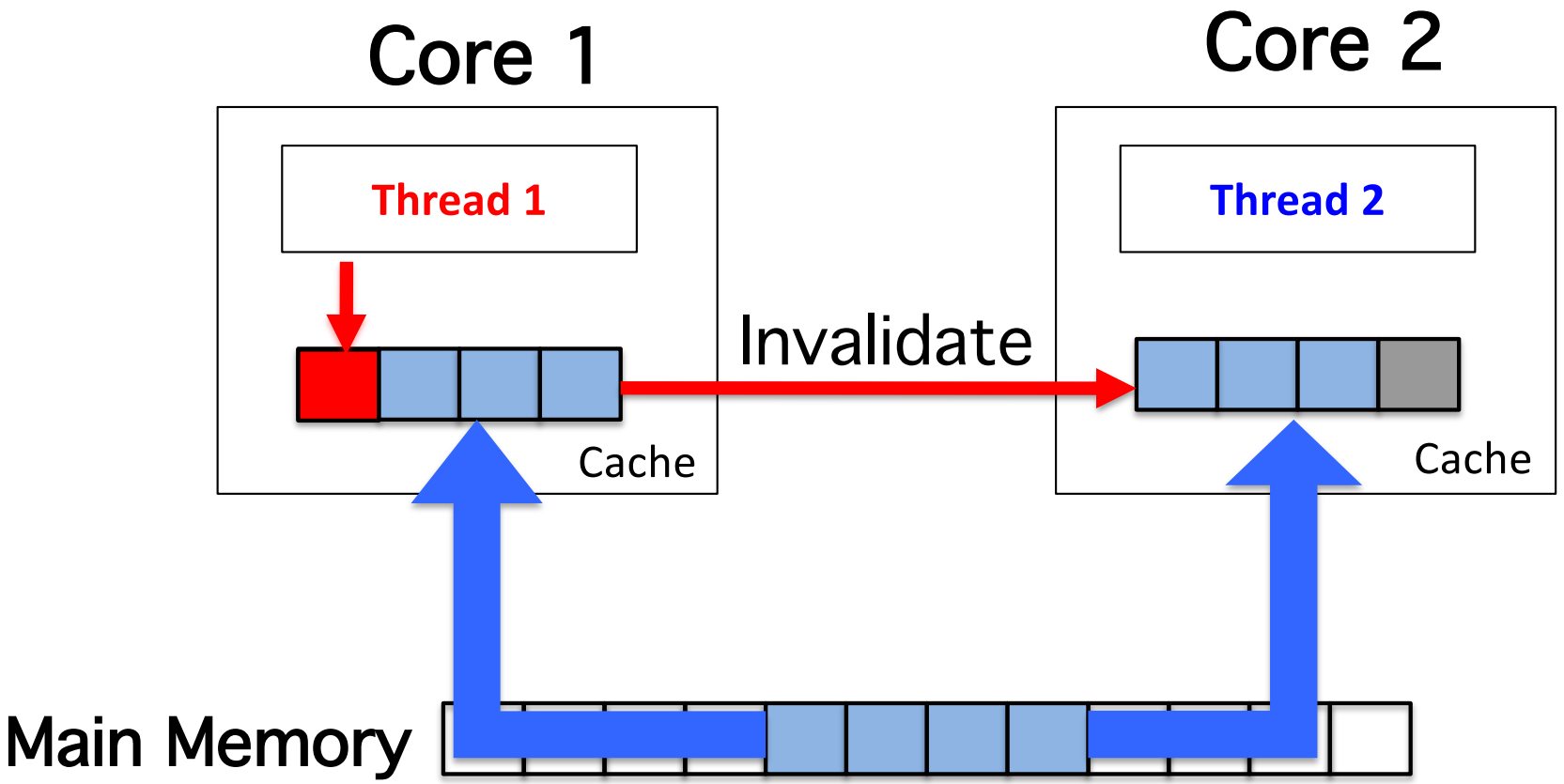
False Sharing



False Sharing

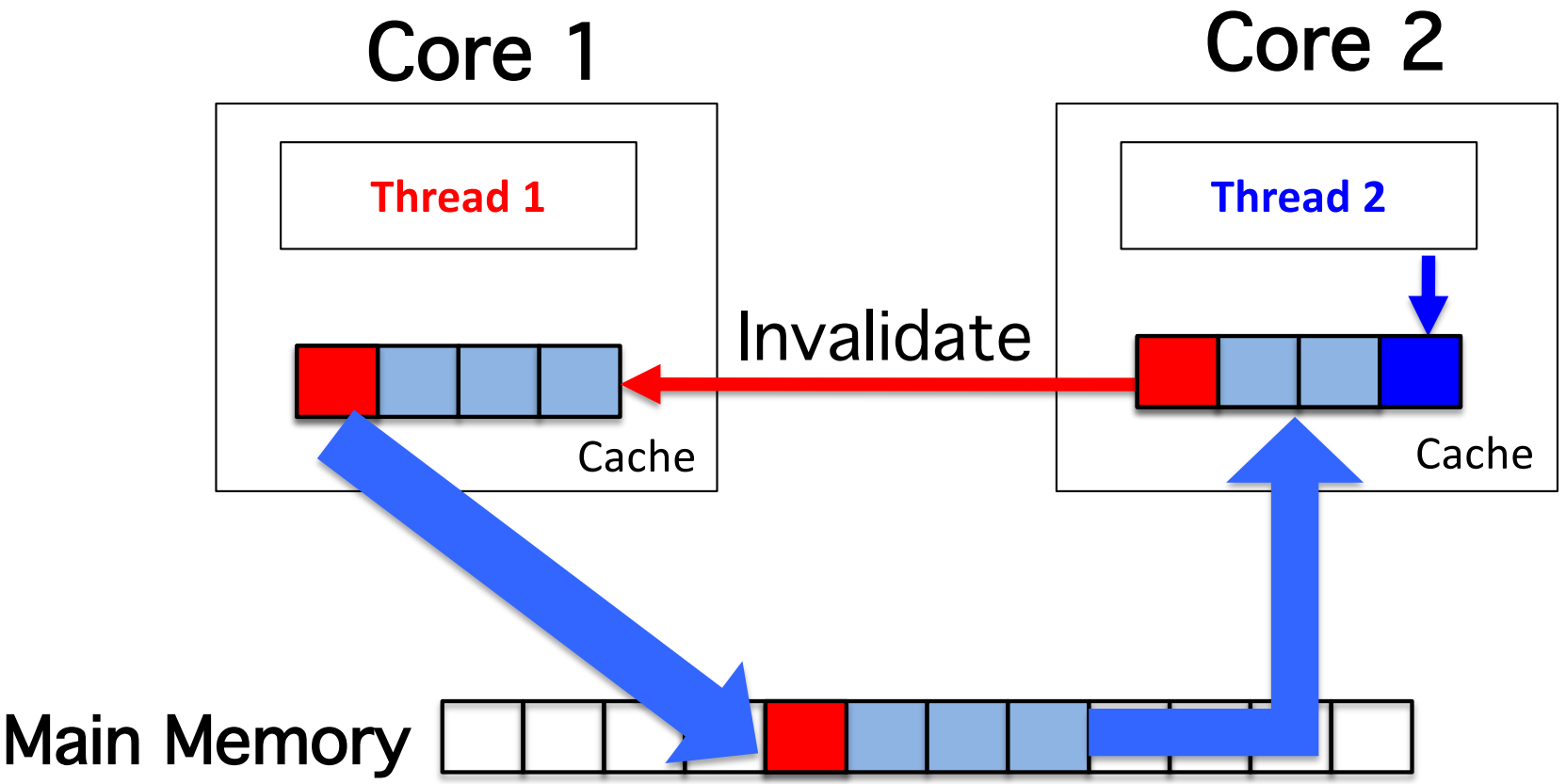


False Sharing



Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

False Sharing



False sharing occurs when threads on different processors modify variables that reside on the same cache line. This stall the CPU, invalidates the cache line, and forces a memory update to maintain cache coherency

Sources: https://people.umass.edu/tongping/pubs/dthreads-final.pptx, and https://people.umass.edu/tongping/pubs/sheriff-final.pptx

Avoiding False Sharing

```
double array[SIZE]; // initialized with random numbers

void ArraySum() {
  double sum[num_worker()]; // zero initialized
  uint64_t chunkSize = SIZE / num_workers();
  finish {
    for (int worker=0; worker<num_workers(); worker++) {
      async {
        uint64_t start = worker * chunkSize;
        uint64_t end = start + chunkSize;
        double my_local_sum = 0;
        for (uint64_t i=start; i<end; i++) {
          my_local_sum += array[i];
        }
        sum[worker] = my_local_sum;
      }
    }
  }
  double result = 0;
  for (int worker=0; worker<num_workers(); worker++)
    result += sum[worker];
}
```

Using local variables in parallel region

```
double array[SIZE]; // initialized with random numbers

void ArraySum() {
  double sum[num_workers() * CACHE_LINE_SIZE]; //zero initialized
  uint64_t chunkSize = SIZE / num_workers();
  finish {
    for (int worker=0; worker<num_workers(); worker++) {
      async {
        uint64_t start = worker * chunkSize;
        uint64_t end = start + chunkSize;
        for (uint64_t i=start; i<end; i++) {
          sum[worker * CACHE_LINE_SIZE] += array[i];
        }
      }
    }
  }
  double result = 0;
  for (int worker=0; worker<num_workers(); worker++)
    result += sum[worker * CACHE_LINE_SIZE];
}
```

Padding the array

Next Class

- Task affinity with HPTs

Acknowledgements

- Several of the slides used in this course are borrowed from the following online course materials:
  - Course COMP322, Prof. Vivek Sarkar, Rice University
  - Course COMP 422, Prof. John Mellor-Crummey, Rice University
  - Course CSE539S, Prof. I-Ting Angelina Lee, Washington University in St. Louis
- Contents are also borrowed from following sources:
  - “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
  - https://computing.llnl.gov/tutorials/parallel_comp/
  - https://images.google.com/