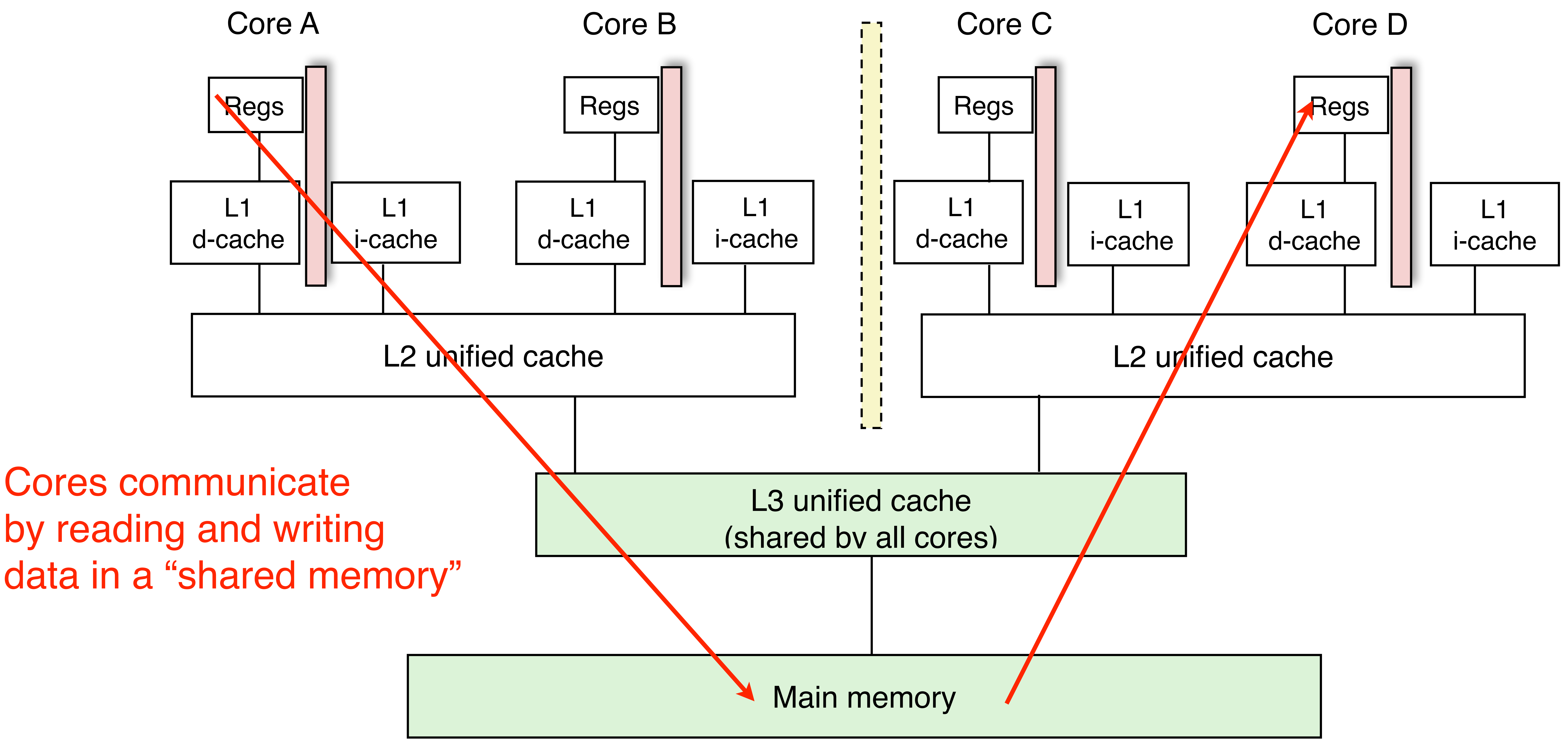
# COMP 322: Fundamentals of Parallel Programming

## Lecture 33: Task Affinity with Places

Mack Joyner
mjoyner@rice.edu

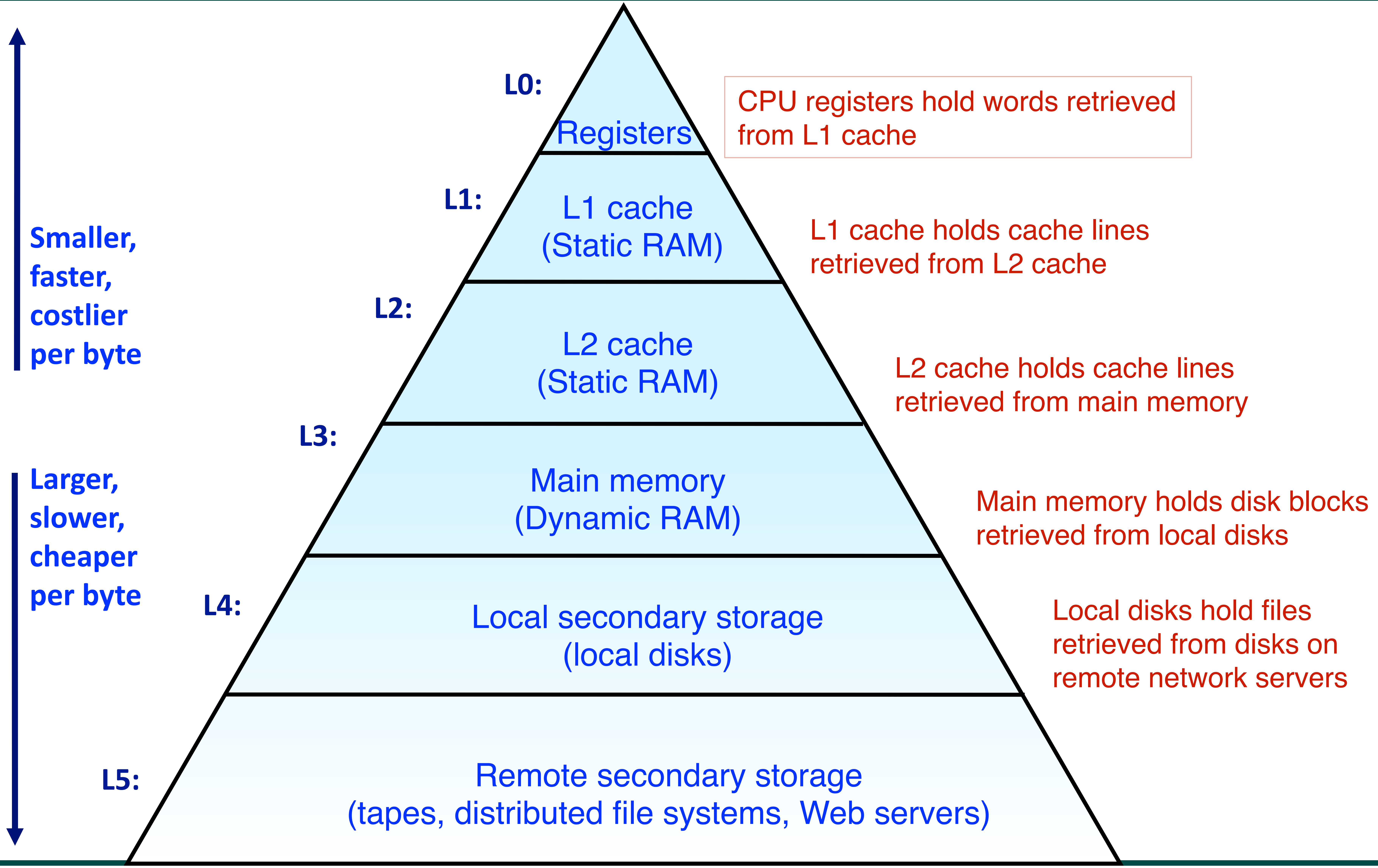
http://comp322.rice.edu

# Organization of a Shared-Memory Multicore Symmetric Multiprocessor (SMP)




- Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip
  — A NOTS node contains TWO 8-core or 12-core E5-2650 v2 Ivy Bridge chips, for a total of 16 or 24 cores

# What is the cost of a Memory Access?
# An example Memory Hierarchy

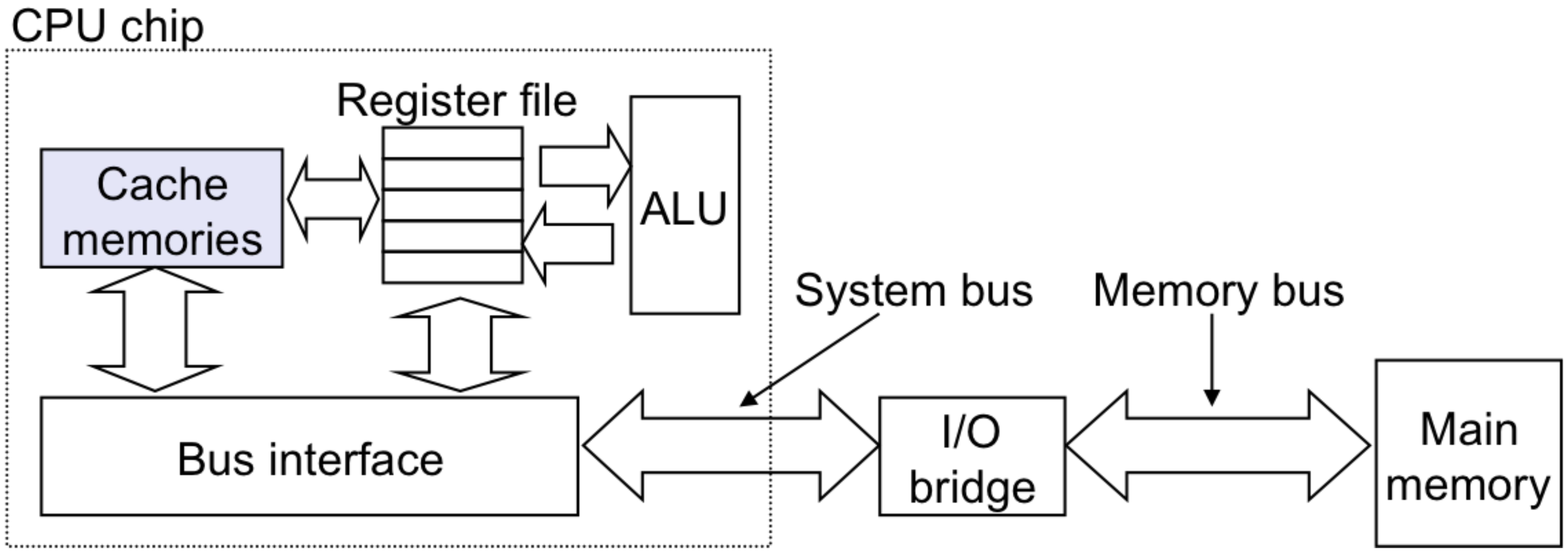





Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

# Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



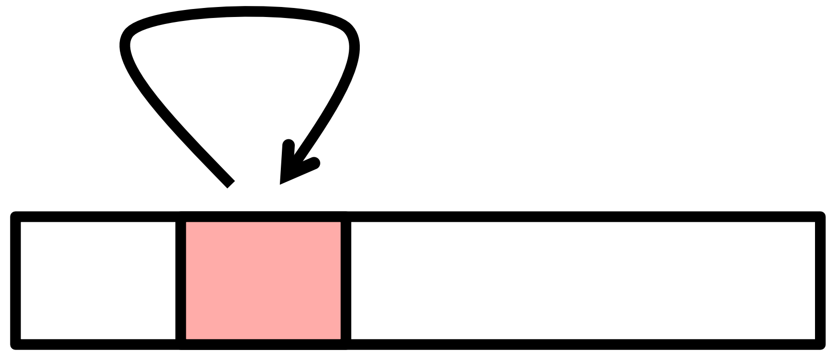



Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

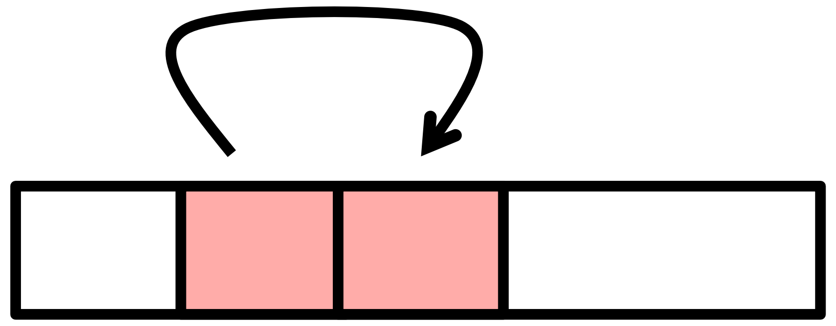
# Locality

- Principle of Locality:
  - Empirical observation: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



  - A Java programmer can only influence spatial locality at the intra-object level
    - The garbage collector and memory management system determines inter-object placement



Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

# Locality Example

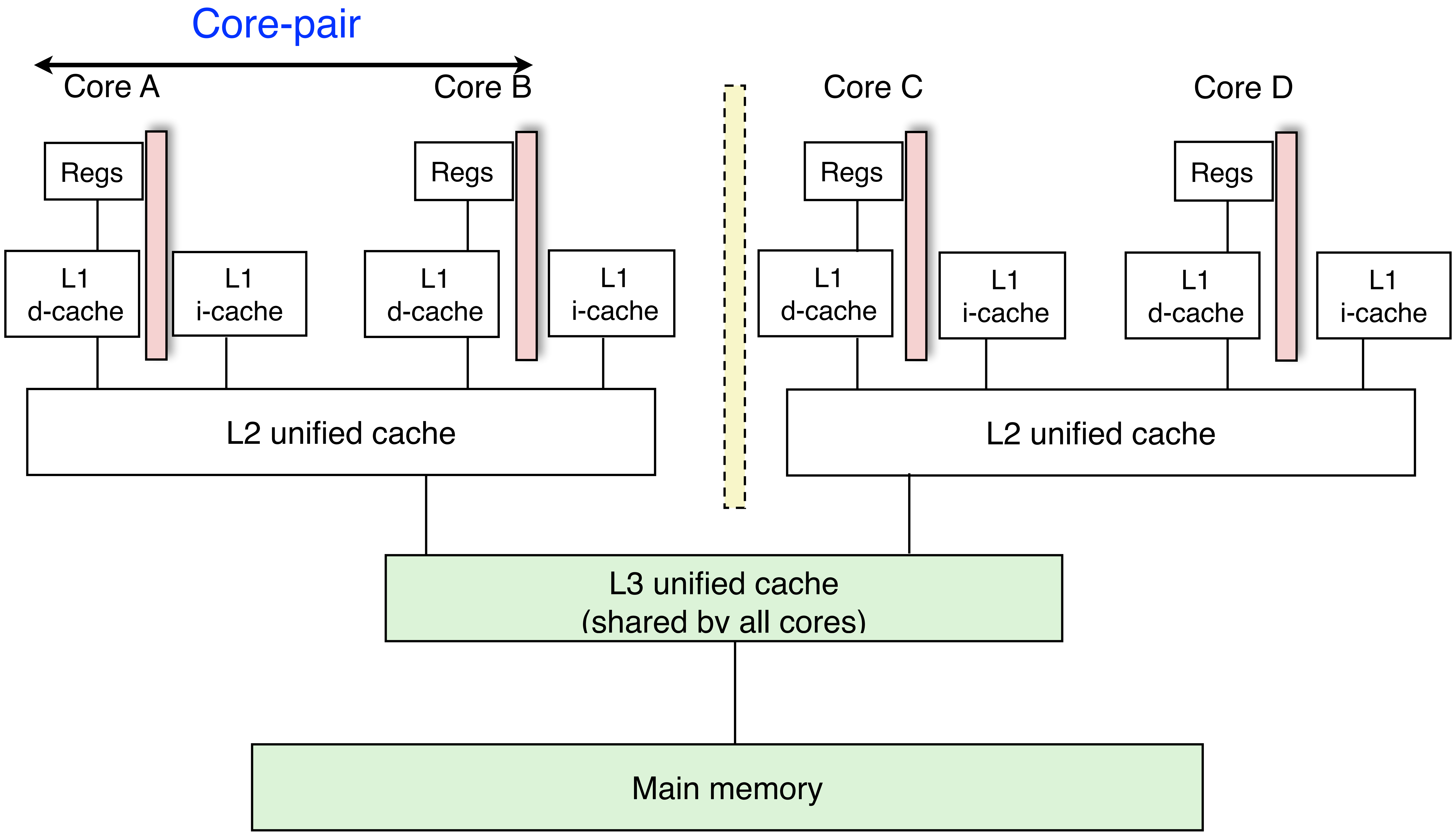
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern).
  - Reference variable sum each iteration.
- Instruction references
  - Cycle through loop repeatedly.
  - Reference instructions in sequence.



Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

# Memory Hierarchy in a Multicore Processor

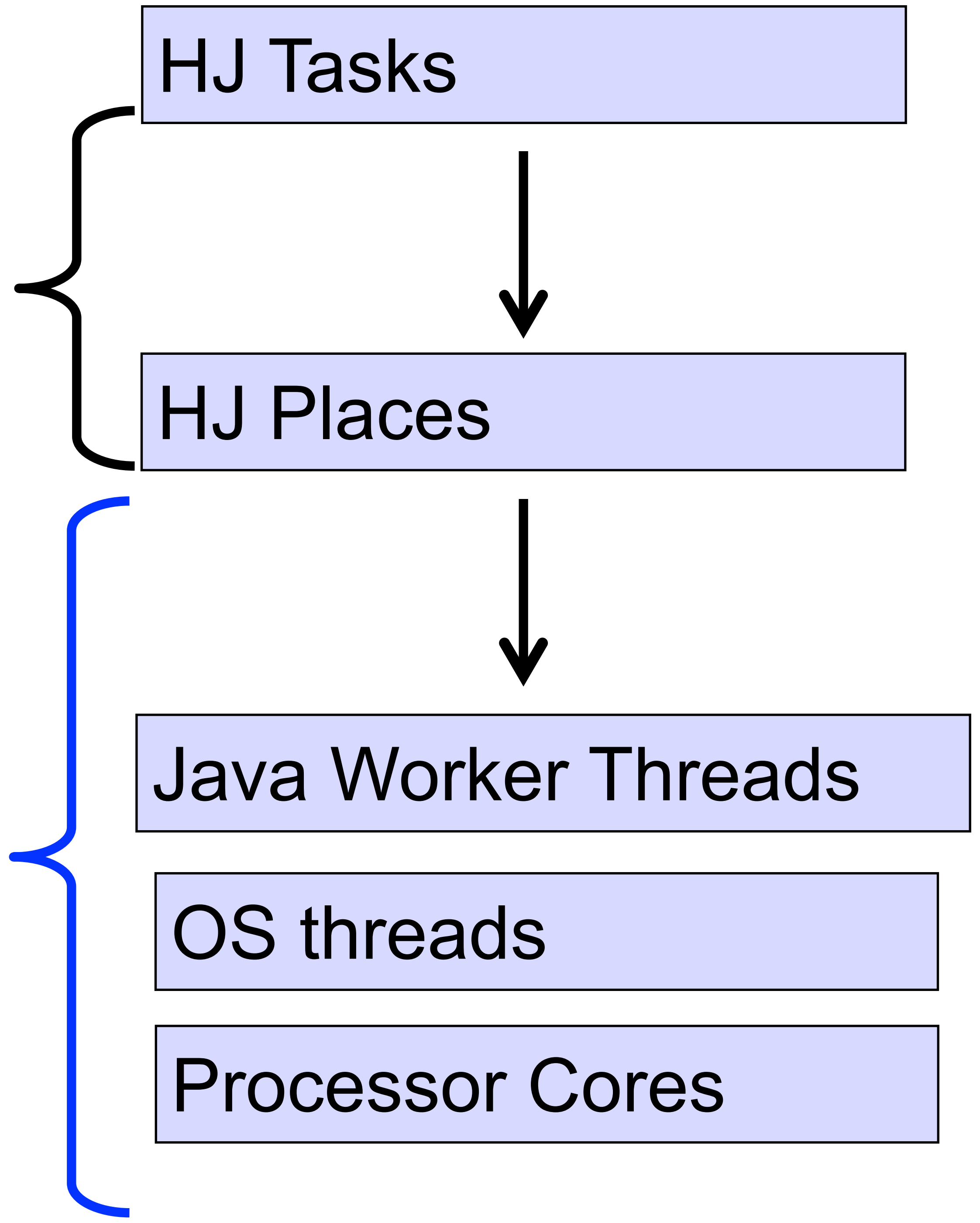



- Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip

# Programmer Control of Task Assignment to Processors

- The parallel programming constructs that we've studied thus far result in tasks that are assigned to processors *dynamically* by the HJ runtime system
  - Programmer does not worry about task assignment details
- Sometimes, programmer control of task assignment can lead to significant performance advantages due to improved locality
- Motivation for HJ "places"
  - Provide the programmer a mechanism to restrict task execution to a subset of processors for improved locality

# Places in HJlib




**HJ programmer defines mapping from HJ tasks to set of places**

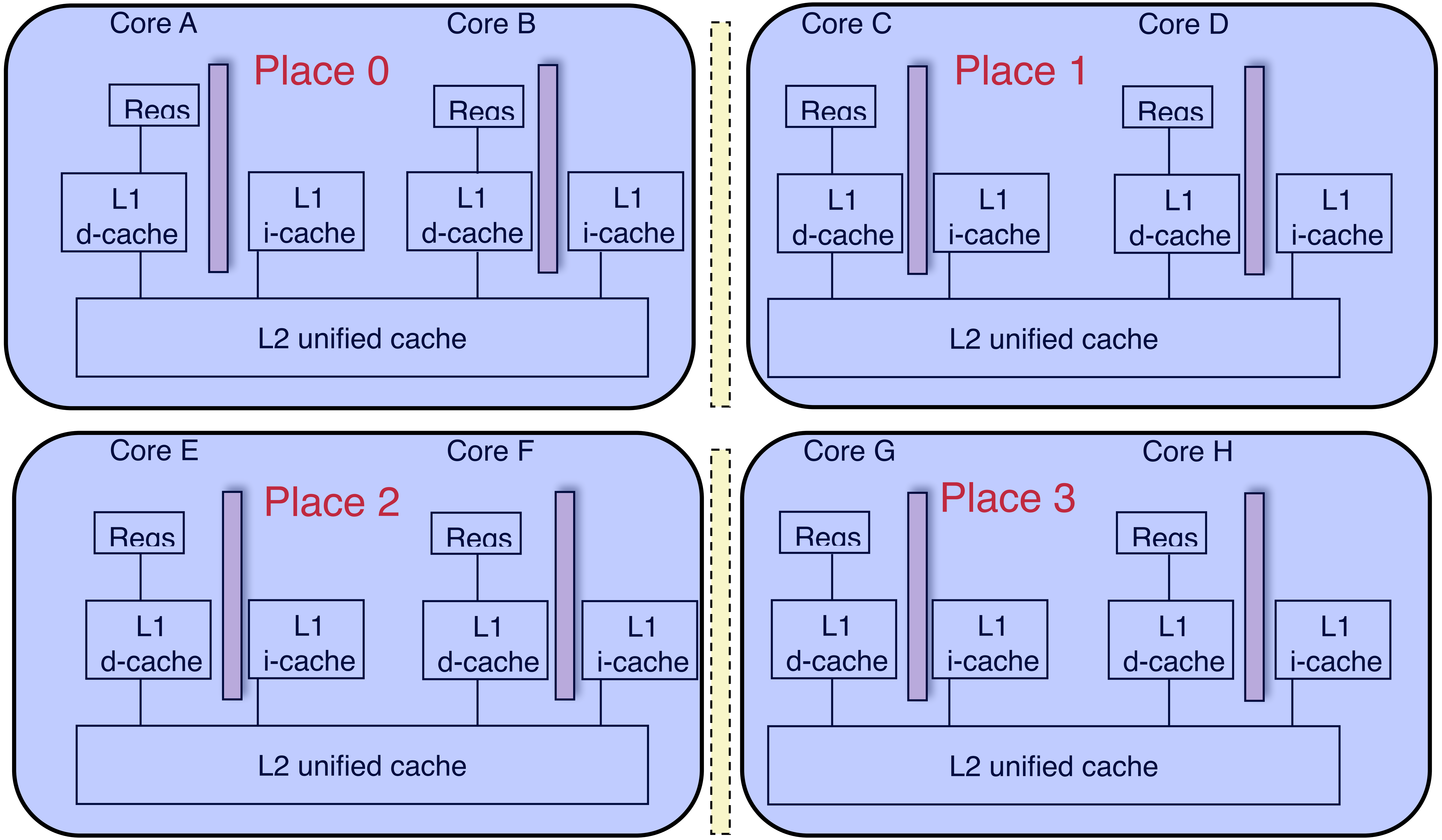
HJ runtime defines mapping from places to one or more worker Java threads per place

The API calls
HjSystemProperty.numPlaces.set(p);
HjSystemProperty.numWorkers.set(w);

when executing an HJ program can be used to specify
- p, the number of places
- w, the number of worker threads per place

we will abbreviate this as p:w

# Example of 4:2 option on an 8-core node
# (4 places w/ 2 workers per place)



# Places in HJlib

here() = place at which current task is executing

numPlaces() = total number of places (runtime constant)

Specified by value of p in runtime option:

HjSystemProperty.numPlaces.set(p);

place(i) = place corresponding to index i

<place-expr>.toString() returns a string of the form “place(id=0)”

<place-expr>.id() returns the id of the place as an int

asyncAt(P, () -> S)

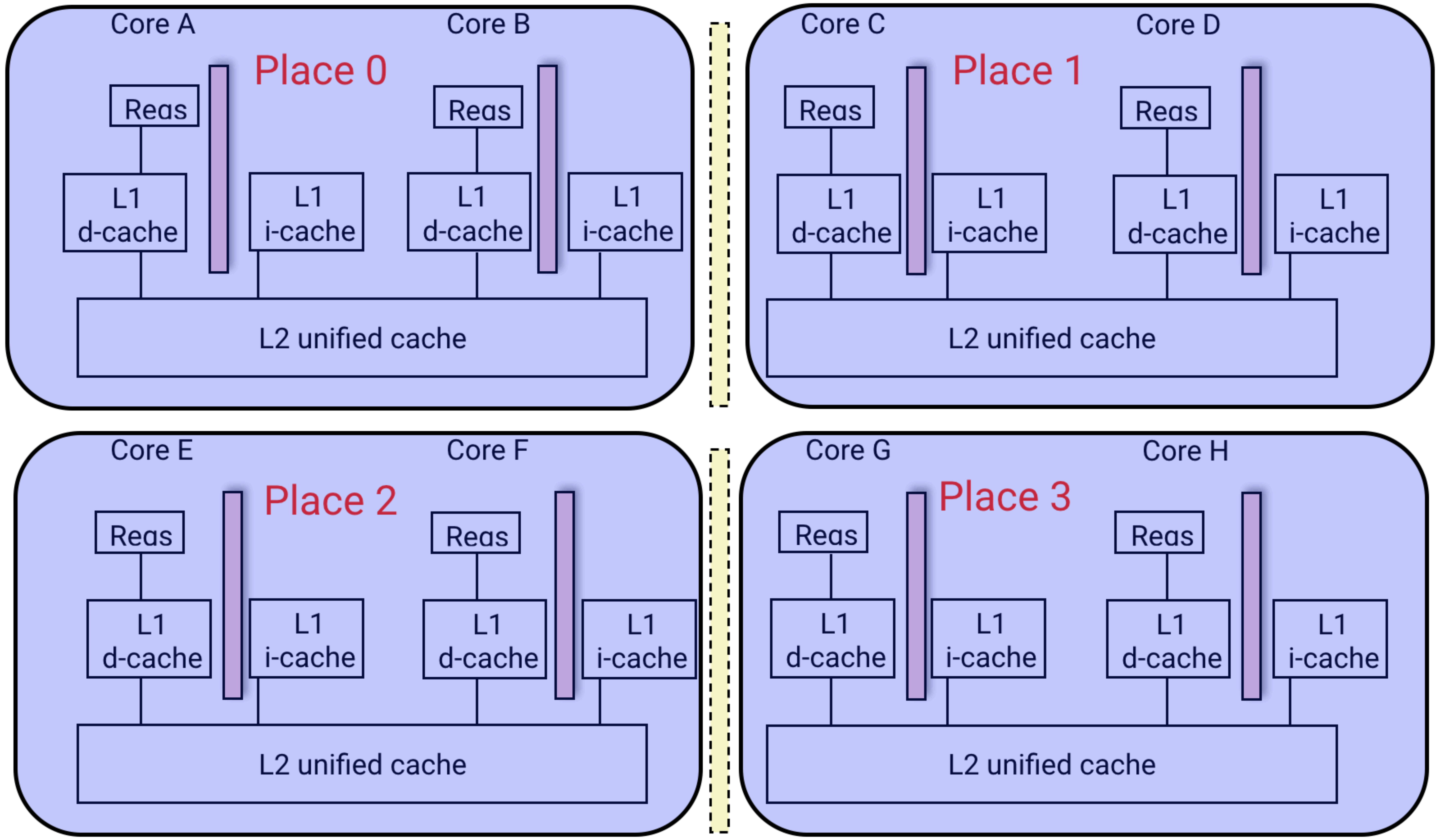
- Creates new task to execute statement S at place P
- async(() -> S) is equivalent to asyncAt(here(), () -> S)
- Main program task starts at place(0)

Note that here() in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

# Example of 4:2 option on an 8-core node
# (4 places w/ 2 workers per place)

```
// Main program starts at place 0
asyncAt(place(0), () -> S1);
asyncAt(place(0), () -> S2);
```

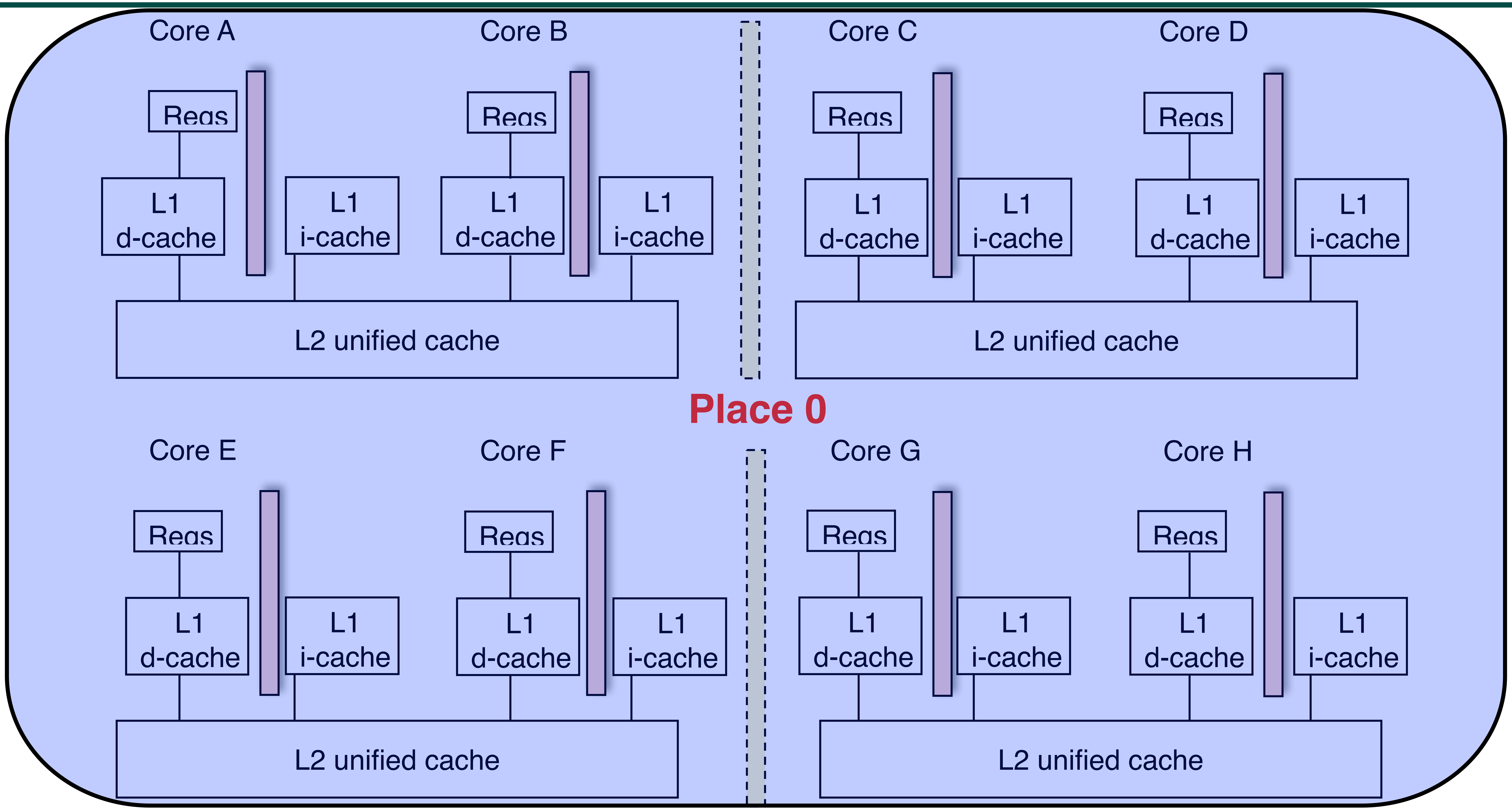
```
asyncAt(place(1), () -> S3);
asyncAt(place(1), () -> S4);
asyncAt(place(1), () -> S5);
```




```
asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);
```

```
asyncAt(place(3), () -> S9);
asyncAt(place(3), () -> S10);
```

# Example of 1:8 option (1 place w/ 8 workers per place)




All async's run at place 0 when there's only one place!

# HJ program with places

```
private static class T1 {
  final HjPlace affinity;

  public T1(HjPlace affinity) {
    // set affinity of instance to place where it is created
    this.affinity = here();
    ...
  }
  public void foo() { ... }
}

finish(() -> {
  println("Parent place: " + here());
  for (T1 a : t1Objects) {
    // Execute async at place with affinity to a
    asyncAt(a.affinity, () -> {
      println("Child place: " + here()); // Child task's place
      a.foo();
    });
  }
});
```

# Block Distribution

- A block distribution splits the index region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example: dist.get(index) for a block distribution on 4 places, when index is in the range, 0…15

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Place id | 0 | | | | 1 | | | | 2 | | | | 3 | | | |

# Distributed Parallel Loops

- The pseudocode below shows the typical pattern used to iterate over an input region r, while creating one async task for each iteration p at the place dictated by distribution d i.e., at place d.get(p).
- This pattern works correctly regardless of the rank and contents of input region r and input distribution d i.e., it is not constrained to block distributions

```
finish {
  region r = ... ; // e.g., [0:15] or [0:7,0:1]
  dist d = dist.factory.block(r);
  for (point p:r)
    async at(d.get(p)) {
      // Execute iteration p at place specified by distribution d
      . . .
    }
} // finish
. . .
```

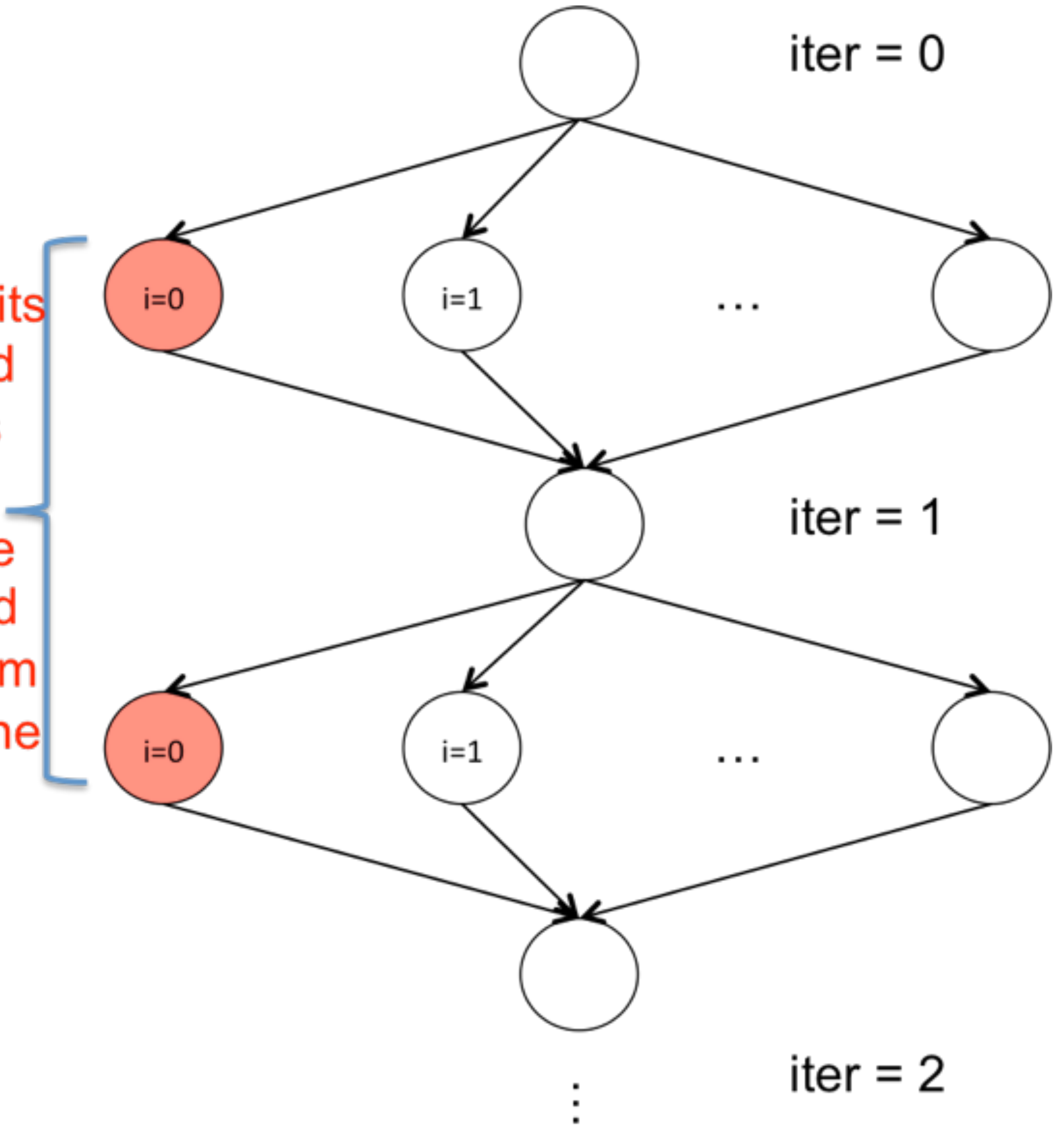
# Chunked Fork-Join Iterative Averaging Example with Places

```
public void runDistChunkedForkJoin(
  int iterations, int numChunks, Dist dist) {
  // dist is a user-defined map from int to HjPlace
  for (int iter = 0; iter < iterations; iter++) {
    finish(() -> {
      for (0, numChunks - 1, (jj) -> {
        asyncAt(dist.get(jj), () -> {
          for (getChunk(1, n, numChunks, jj), (j) -> {
            myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
          }
        });
      });
    });
    double[] temp = myNew; myNew = myVal; myVal = temp;
  } // for iter
}
```

- Chunk jj is always executed in the same place for each iter
- Method runDistChunkedForkJoin can be called with different values of distribution parameter d

# Analyzing Locality of Fork-Join Iterative Averaging Example with Places

Locality benefits will be realized if all instances of chunk 0 execute on the same core and reuse data from the same cache



# Cyclic Distribution

- A cyclic distribution "cycles" through places 0 … place.MAX PLACES – 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example: dist.get(index) for a cyclic distribution on 4 places, when index is in the range, 0…15

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Place id | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |