COMP 322: Fundamentals of Parallel Programming

Lecture 34: Task Affinity with Places

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/

COMP 322 Lecture 34 10 April 2017

Worksheet #33: Combining Task and MPI parallelism

Name: ______________________ Net ID: ______________________

Compute the critical path length for the MPI program shown on the right in pseudocode, assuming that it is executed with 2 processes/ranks. (Assume that the send/recv calls in lines 5 & 10 match with each other.)

```
1.  main() {
2.    if (my rank == 0)
3.      finish { // F1
4.        async await(req) doWork(1);
5.        MPI_Irecv(rank 1, … , req);
6.        doWork(1);
7.      }
8.    else {
9.      doWork(1);
10.     MPI_Send(rank 0, …);
11.   }
12. } // main
```

CPL = 2

Organization of a Distributed-Memory Multiprocessor

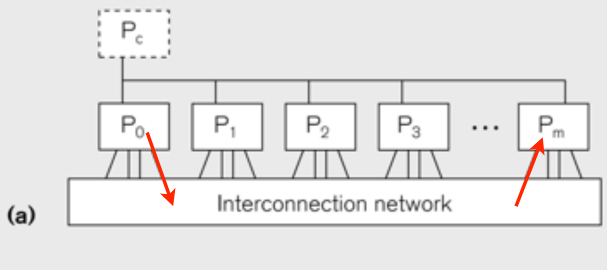
Figure (a)

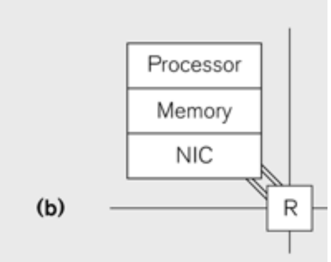
- Host node (P_c) connected to a cluster of processor nodes (P_0 … P_m)
- Processors P_0 … P_m communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

Figure (b)

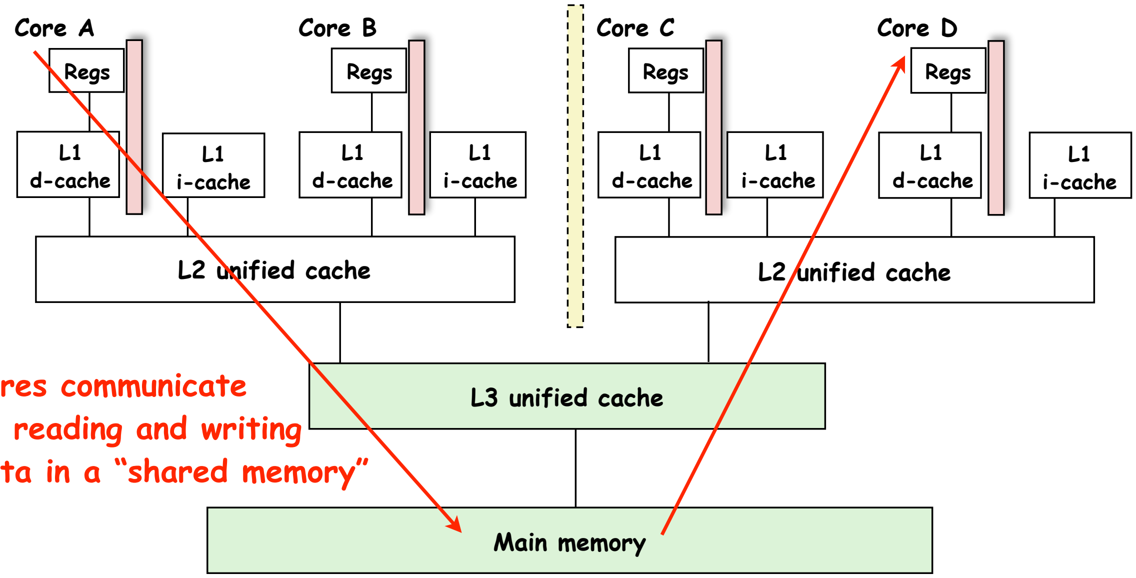
- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect

Processors communicate by sending messages via an interconnect





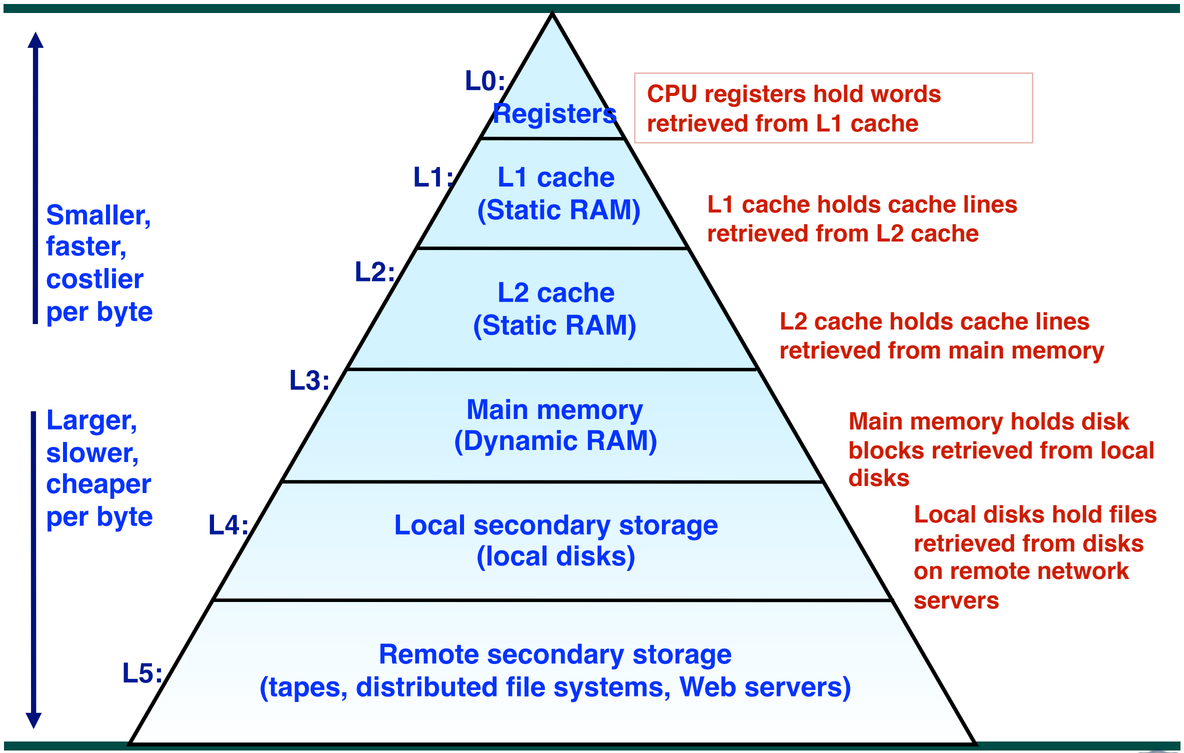
Organization of a Shared-Memory Multicore Symmetric Multiprocessor (SMP)



Cores communicate by reading and writing data in a “shared memory”

- Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip
  - —A STIC node contains TWO such chips, for a total of 8 cores

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



Smaller, faster, costlier per byte

Larger, slower, cheaper per byte

L0: Registers

L1: L1 cache (Static RAM)

L2: L2 cache (Static RAM)

L3: Main memory (Dynamic RAM)

L4: Local secondary storage (local disks)

L5: Remote secondary storage (tapes, distributed file systems, Web servers)

CPU registers hold words retrieved from L1 cache

L1 cache holds cache lines retrieved from L2 cache

L2 cache holds cache lines retrieved from main memory

Main memory holds disk blocks retrieved from local disks

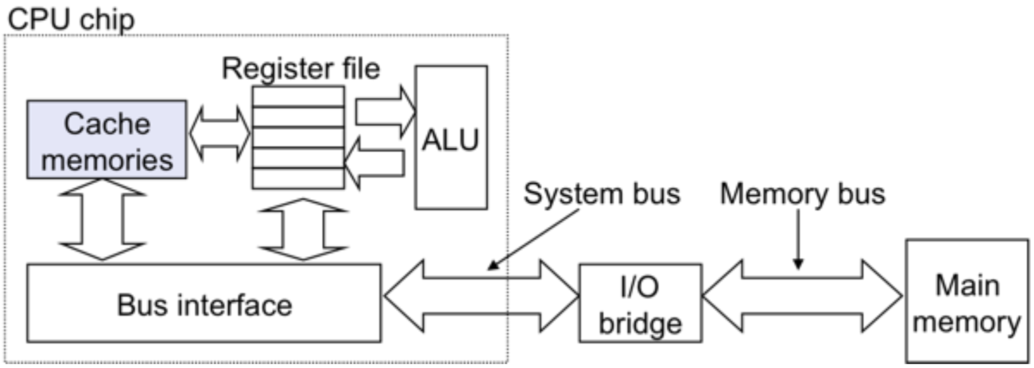
Local disks hold files retrieved from disks on remote network servers

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

---

Cache Memories

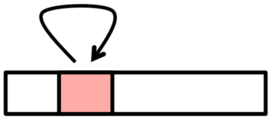
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - —Hold frequently accessed blocks of main memory
- CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.
- Typical system structure:



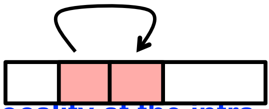
Source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

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

---

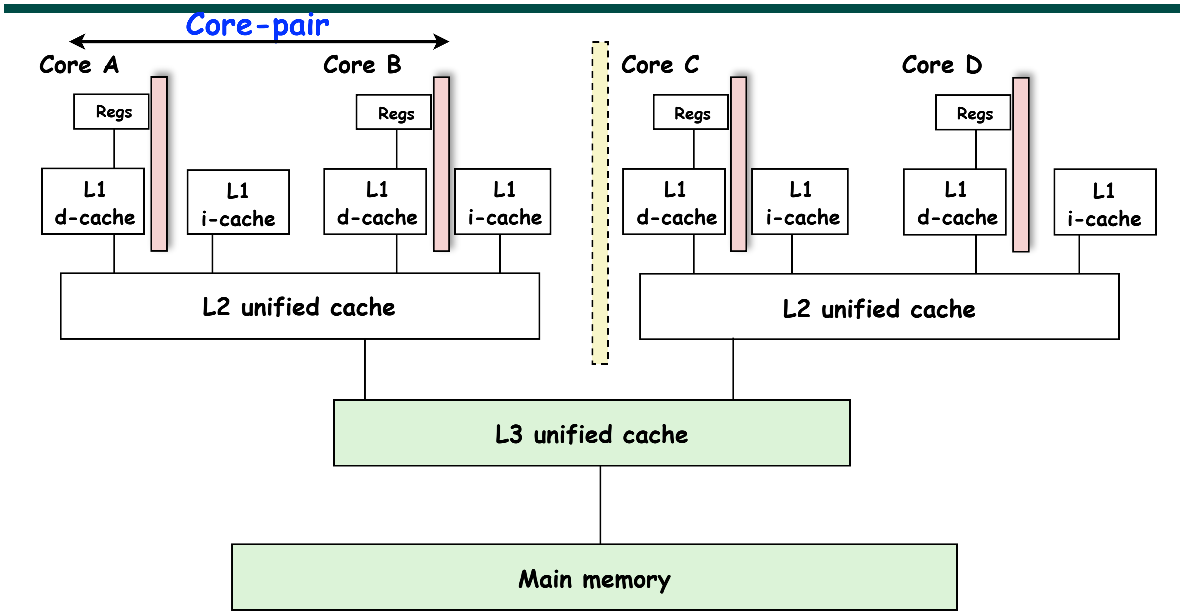
Locality Example

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

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

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

Memory Hierarchy in a Multicore Processor



Core-pair

Core A

Core B

Core C

Core D

Regs

L1 d-cache

L1 i-cache

L2 unified cache

L3 unified cache

Main memory

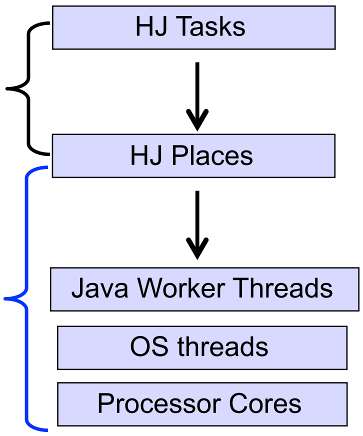
- **Memory hierarchy for a single Intel Xeon (Nehalem) Quad-core processor chip**
  - —A STIC node contains TWO such chips, for a total of 8 cores

---

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
  - — Current HJlib implementation supports one level of locality via places, but future HJlib versions will support hierarchical places

Places in HJlib


HJ Tasks

HJ Places

Java Worker Threads

OS threads

Processor Cores

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

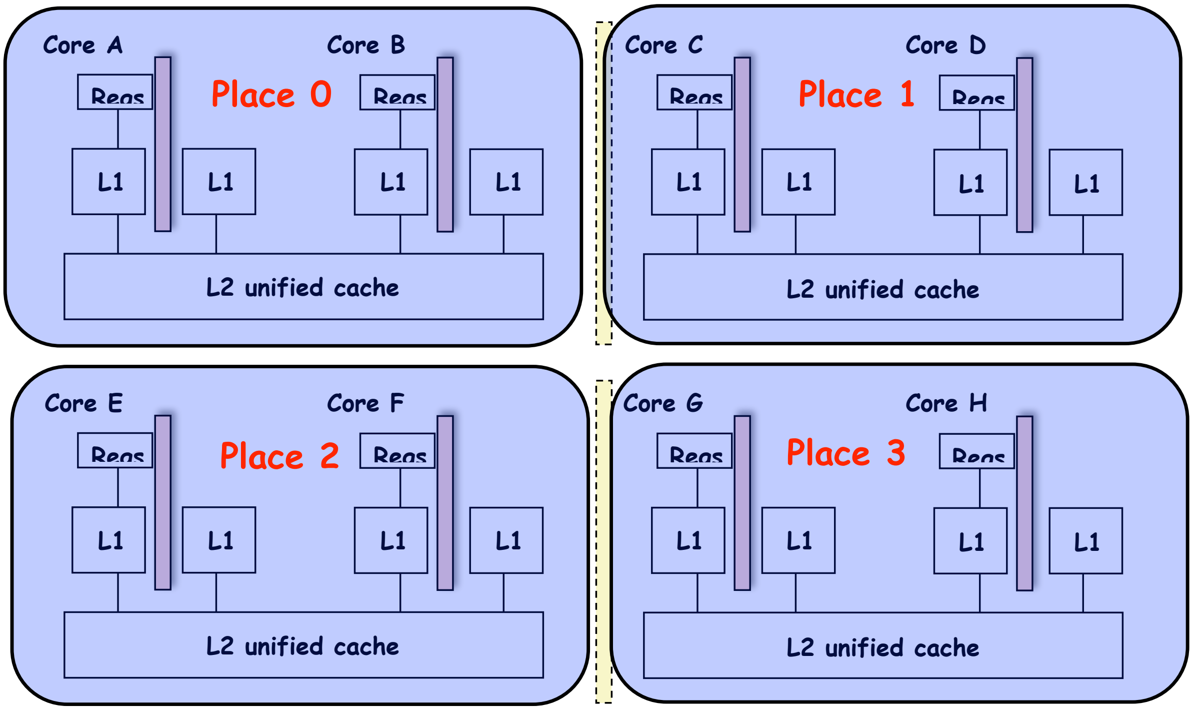
```
HjSystemProperty.numPlaces.set(p);
HjSystemProperty.numWorkers.set(w);
```

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)`

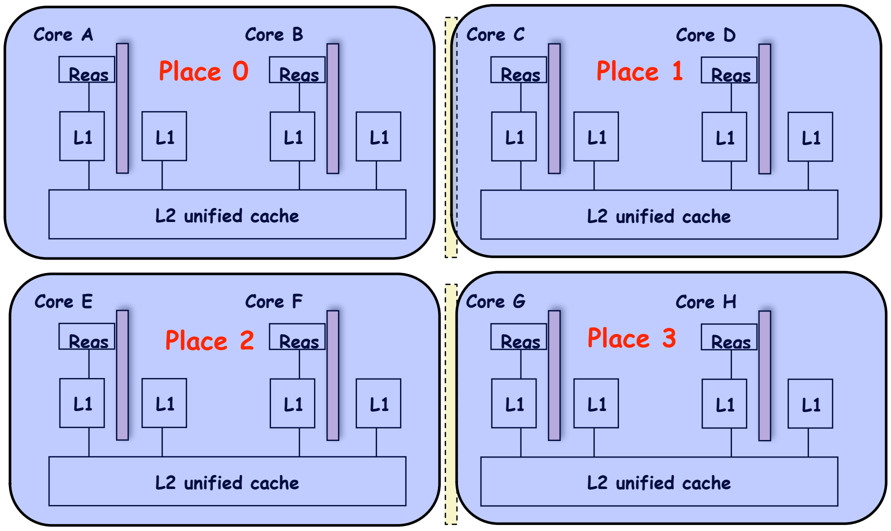
Note that `here()` in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

---

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);
```

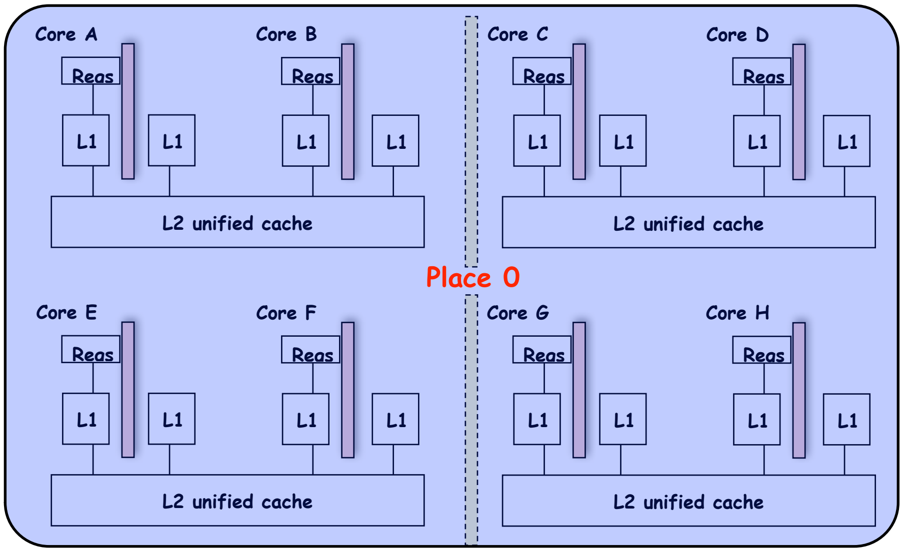


```
asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);
```

```
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

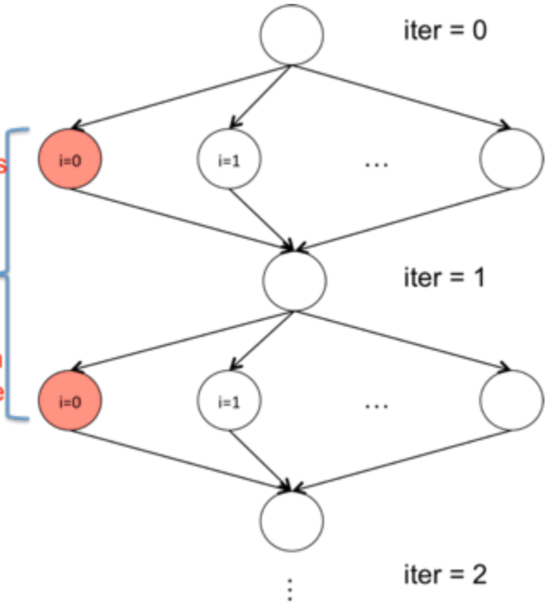
```
private static class T1 {
  final HjPlace affinity;

  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 saync at place with affinity to a
    asyncAt(a.affinity, () -> {
      println("Child place: " + here()); // Child task's place
      a.foo();
    });
  }
});
```

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(() -> {
      forseq (0, numChunks - 1, (jj) -> {
        asyncAt(dist.get(jj), () -> {
          forseq (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

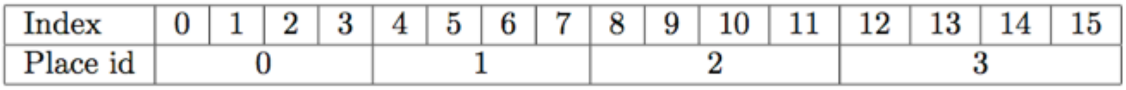
Analyzing Locality of Fork-Join Iterative Averaging Example with Places



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

Block Distribution

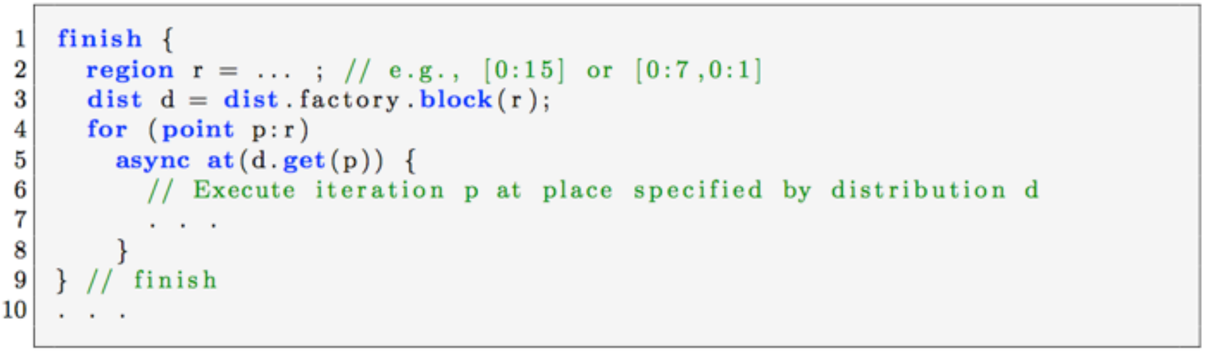
- A block distribution splits the index region into contiguous subregions, one per place, while trying to keep the subregions as close to equal in size as possible.
- Block distributions can improve the performance of parallel loops that exhibit spatial locality across contiguous iterations.
- Example: 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
. . .
```

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