

COMP 322: Fundamentals of Parallel Programming

Lecture 32: Task Affinity with Places

(Start of Module 3 on Distribution & Locality)

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

<http://comp322.rice.edu/>

COMP 322

Lecture 32

6 April 2016



Worksheet #31: Solution Finding maximal index of goal in matrix

Below is a code fragment intended to find the maximal index of goal that occurs multiple times in the input matrix. What log

```
1. class AsyncFinishEurekaSearchMaxIndexOfGoal {
2.     HjEureka eurekaFactory() {
3.         comparator = (cur, newVal) -> { //
4.             (cur.x==newVal.x) ? (cur.y - newVal.y) : 0;
5.         }
6.     int[] doWork(matrix, goal) {
7.         val eu = eurekaFactory()
8.         finish (eu, () -> { // eureka registration
9.             forasync (0, matrix.length - 1, (r) ->
10.                 procRow(matrix(r), r, goal));
11.         });
12.         return eu.get()
13.     }
14.     void procRow(array r, goal) {
15.         for (int c = array.length() - 1; c >= 0; c--)
16.             check([r, c]) // terminate if comparator returns negative
17.             if goal.match(array(c)) offer([r, c]) // updates cur in eureka
18.     } }
```

	0	...	10	...	15	...
...						
5			M			
...						
10	M				M	

The task terminates when `check([r,c])` is called and the comparator has `cur` smaller than `[r,c]`. We need to ensure the iteration order in our code is such that the comparator returning negative means we cannot produce an offer `([r',c'])` where `[r', c']` is greater than the value of `cur`.



Organization of a Distributed-Memory Multiprocessor

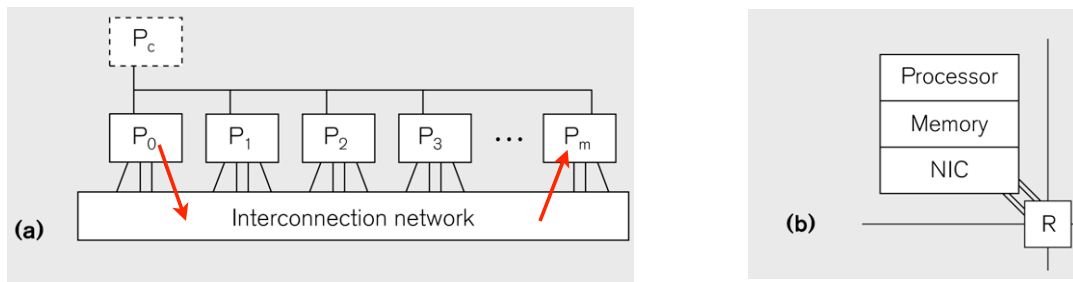
Figure (a)

- Host node (P_c) connected to a cluster of processor nodes ($P_0 \dots P_m$)
- Processors $P_0 \dots P_m$ 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)

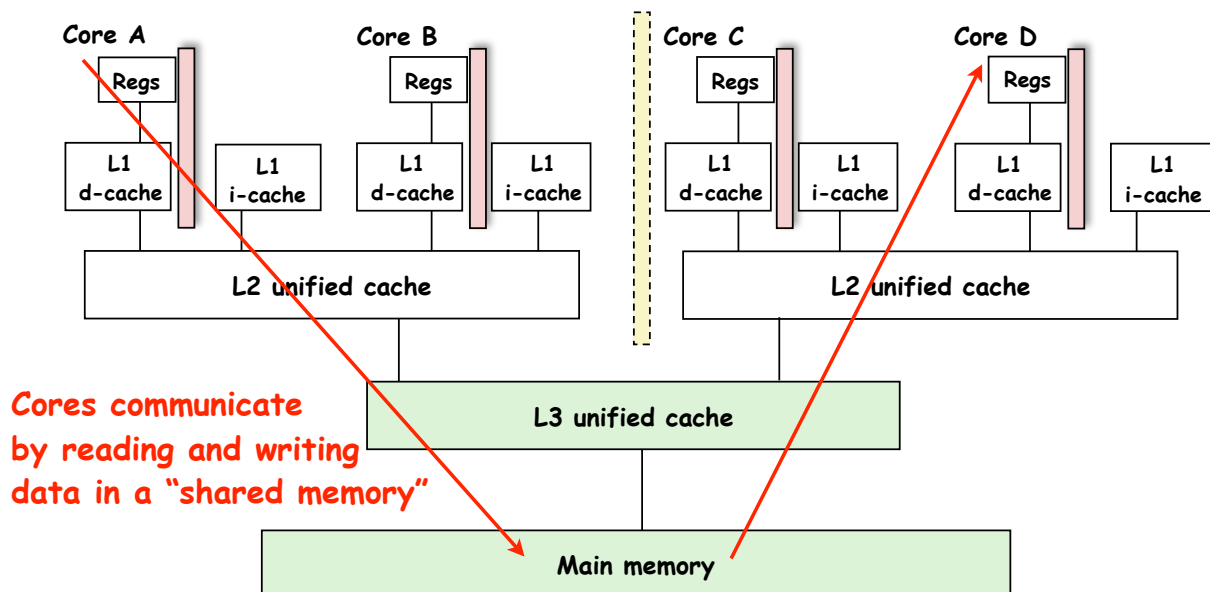
Figure (b)

- 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)



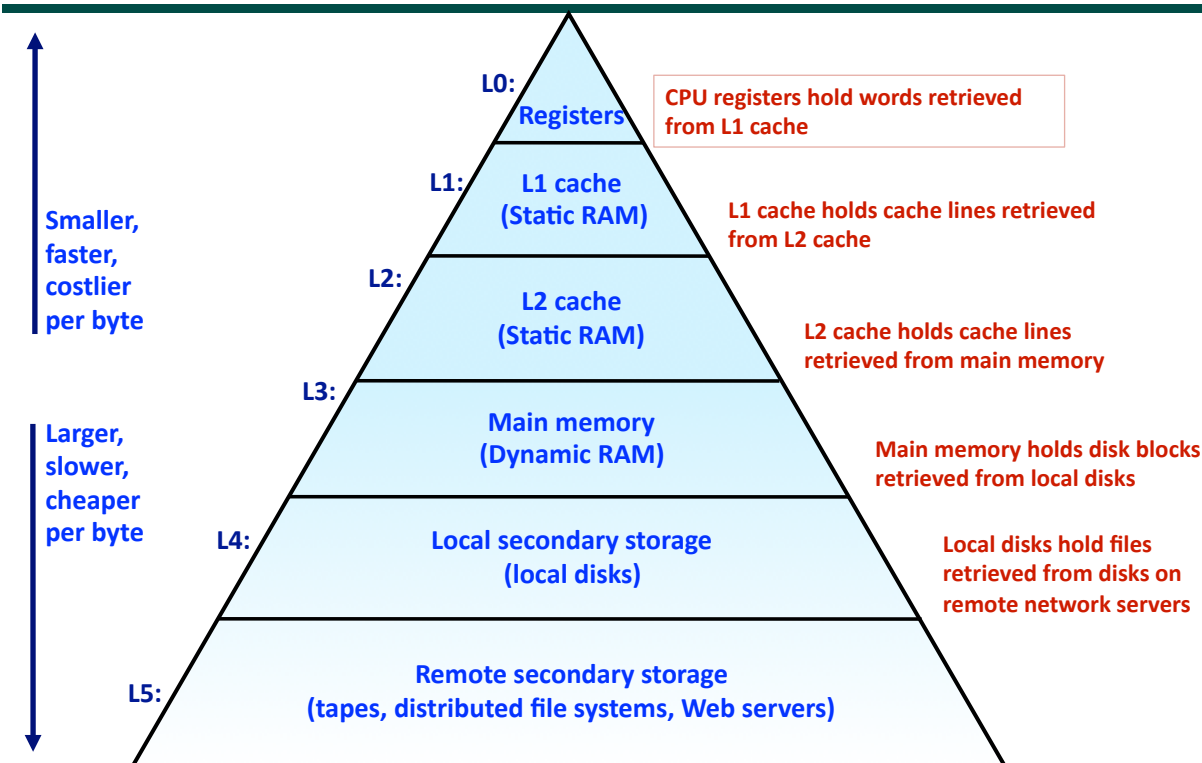
Cores communicate by reading and writing data in a "shared memory"

- 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



5

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



Storage Trends

SRAM

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	19,200	2,900	320	256	100	75	60	320
access (ns)	300	150	35	15	3	2	1.5	200

DRAM

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	8,000	880	100	30	1	0.1	0.06	130,000
access (ns)	375	200	100	70	60	50	40	9
typical size (MB)	0.064	0.256	4	16	64	2,000	8,000	125,000

Disk

Metric	1980	1985	1990	1995	2000	2005	2010	2010:1980
\$/MB	500	100	8	0.30	0.01	0.005	0.0003	1,600,000
access (ms)	87	75	28	10	8	4	3	29
typical size (MB)	1	10	160	1,000	20,000	160,000	1,500,000	1,500,000

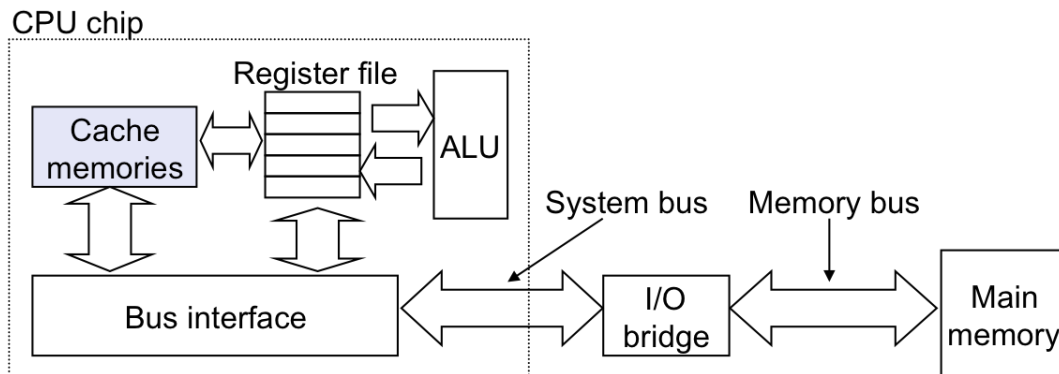
6

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:



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



Examples of Caching in the Hierarchy

Hierarchy Level		ed by
Registers	<p>Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.</p> <p>A. W. Burks, H. H. Goldstine, and J. von Neumann <i>Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (1946)</i></p>	
TLB		
L1 cache		
L2 cache		
Virtual		
Buffer		
Disk cache		
Network		
Browser cache		
Web cache		

Ultimate goal: create a large pool of storage with average cost per byte that approaches that of the cheap storage near the bottom of the hierarchy, and average latency that approaches that of fast storage near the top of the hierarchy.

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



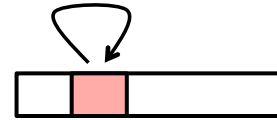
Locality

- **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:**

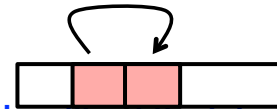
- Recently referenced items are likely to be referenced again in the near future



- **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

9

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.

Spatial locality

Temporal locality

- **Instruction references**

- Reference instructions in sequence.

- Cycle through loop repeatedly.

Spatial locality

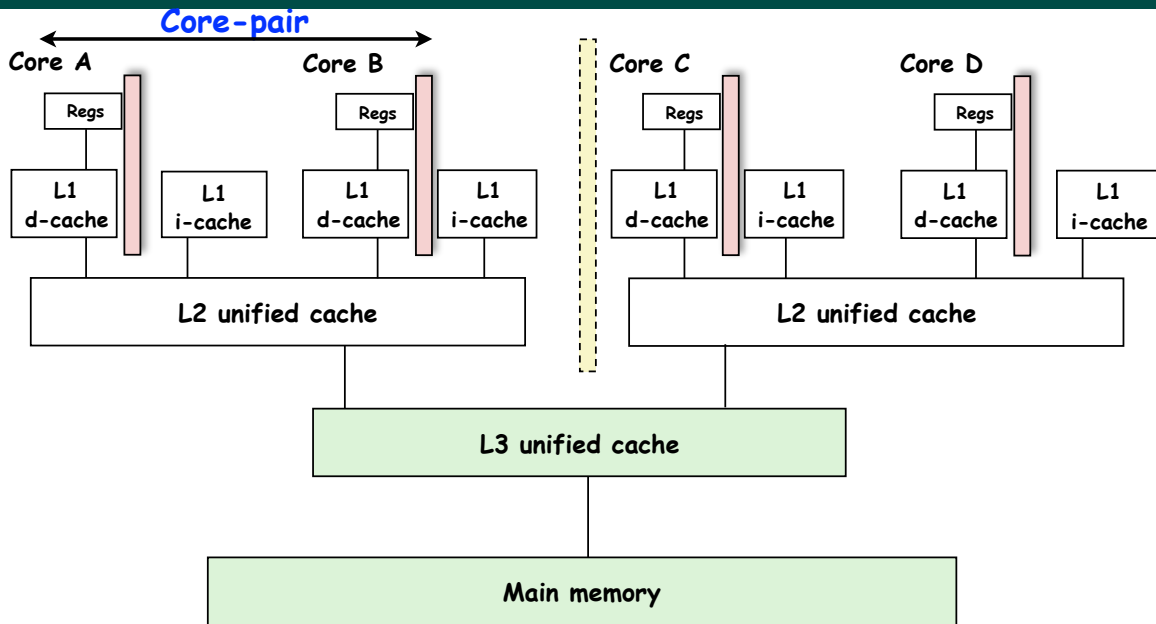
Temporal locality

10

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
 - 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 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

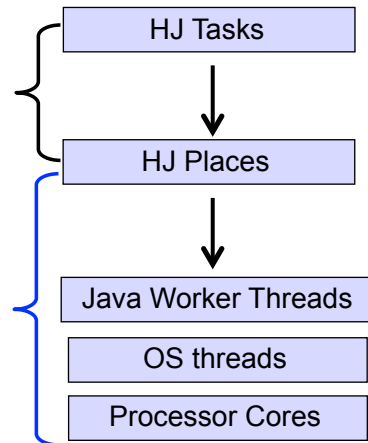
```
HjSystemProperty.numPlaces.set(p);  
HjSystemProperty.numWorkers.set(w);
```

when executing an HJ program can be used to specify

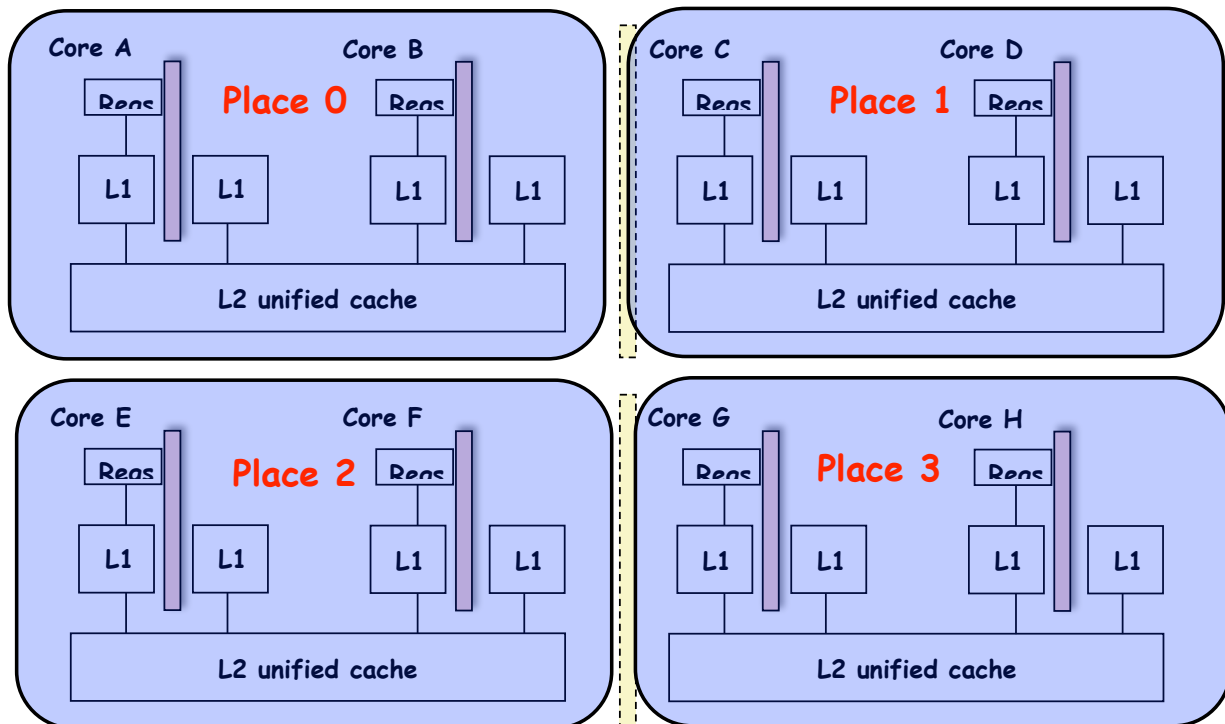
p, the number of places

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)`

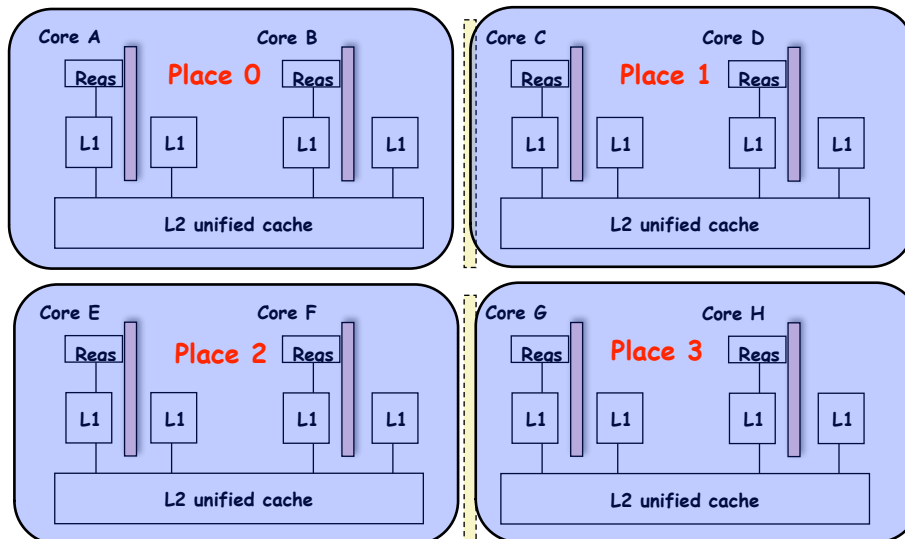
Note that `here()` in a child task refers to the place `P` at which the child task is executing, not the place where the parent task is executing



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);
```



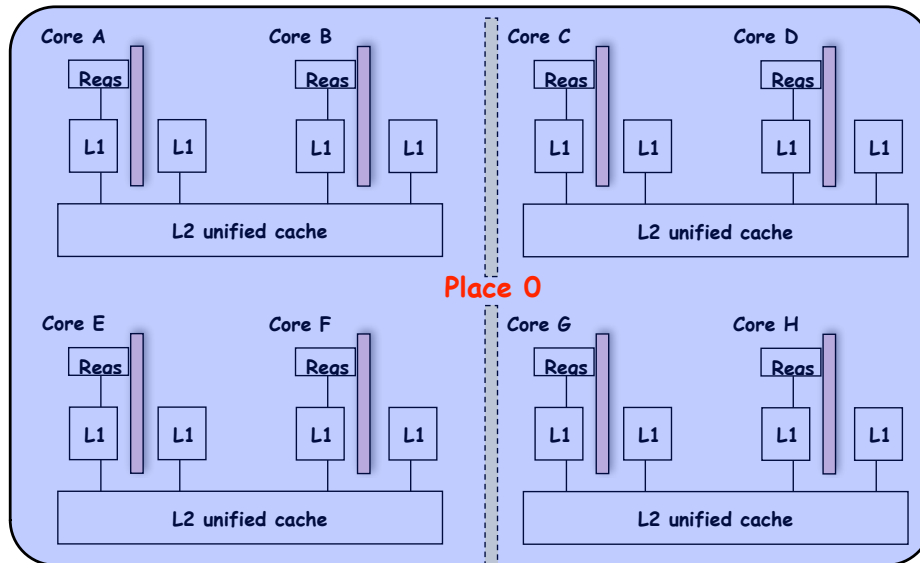
```
asyncAt(place(2), () -> S6);  
asyncAt(place(2), () -> S7);  
asyncAt(place(2), () -> S8);
```

```
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!



17

COMP 322, Spring 2016 (V. Sarkar, S. Imam)



HJ program with places

```
1. private static class T1 {
2.     final HjPlace affinity;

4.     public T1(HjPlace affinity) {
5.         // set affinity of instance to place where it is created
6.         this.affinity = here();
7.         ...
8.     }
9.     public void foo() { ... }
10. }
11.
12. finish(() -> {
13.     println("Parent place: " + here());
14.     for (T1 a : t1Objects) {
15.         // Execute saync at place with affinity to a
16.         asyncAt(a.affinity, () -> {
17.             println("Child place: " + here()); // Child task's place
18.             a.foo();
19.         });
20.     }
21. });
```

18

COMP 322, Spring 2016 (V. Sarkar, S. Imam)



Chunked Fork-Join Iterative Averaging Example with Places

```
1. public void runDistChunkedForkJoin(  
2.     int iterations, int numChunks, Dist dist) {  
3.     // dist is a user-defined map from int to HjPlace  
4.     for (int iter = 0; iter < iterations; iter++) {  
5.         finish(() -> {  
6.             forseq (0, numChunks - 1, (jj) -> {  
7.                 asyncAt(dist.get(jj), () -> {  
8.                     forseq (getChunk(1, n, numChunks, jj), (j) -> {  
9.                         myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;  
10.                    }  
11.                });  
12.            });  
13.        });  
14.        double[] temp = myNew; myNew = myVal; myVal = temp;  
15.    } // for iter  
16. }
```

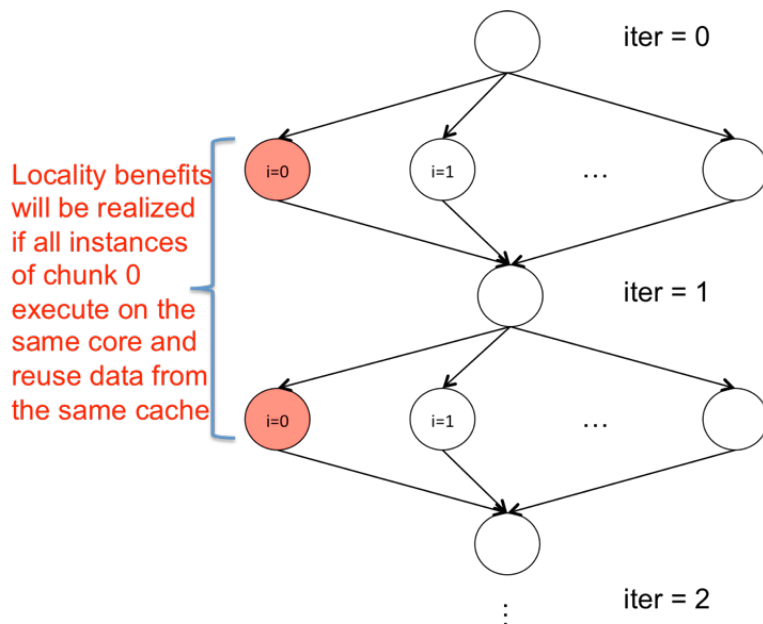
- Chunk jj is always executed in the same place for each iter
- Method `runDistChunkedForkJoin` can be called with different values of distribution parameter d

19

COMP 322, Spring 2016 (V. Sarkar, S. Imam)



Analyzing Locality of Fork-Join Iterative Averaging Example with Places



20

COMP 322, Spring 2016 (V. Sarkar, S. Imam)



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

```
1 finish {
2   region r = ... ; // e.g., [0:15] or [0:7,0:1]
3   dist d = dist.factory.block(r);
4   for (point p:r)
5     async at(d.get(p)) {
6       // Execute iteration p at place specified by distribution d
7       . . .
8     }
9 } // finish
10 . . .
```



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

