COMP 322: Fundamentals of Parallel Programming

Lecture 18: Task Affinity with Places (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments

- Supercomputing 2007 tutorial on "Programming using the Partitioned Global Address Space (PGAS) Model" by Tarek El-Ghazawi and Vivek Sarkar
  - —http://sc07.supercomputing.org/schedule/event_detail.php?evid=11029

Places in HJ

here = place at which current task is executing

place.MAX_PLACES = total number of places (runtime constant)
Specified by value of `p` in runtime option, `-places p:w`

place.factory.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

async at(P) S

- Creates new task to execute statement S at place P
- **async S** is equivalent to **async at(here) S**
- Main program task starts at **place.factory.place(0)**

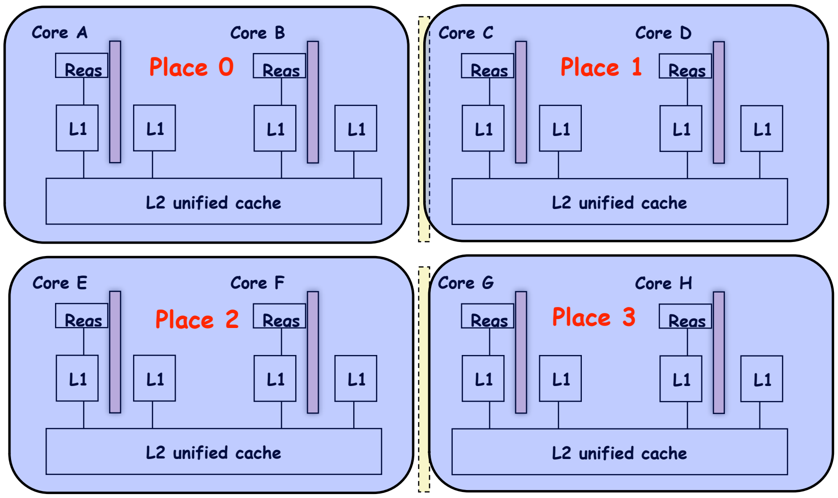
Note that **here** in a child task refers to the place P at which the child task is executing, not the place where the parent task is executing

---

Example of –places 4:2 option on a SUG@R node (4 places w/ 2 workers per place)

```
// Main program starts at place 0
async at(place.factory.place(0)) S1;
async at(place.factory.place(0)) S2;
```

```
async at(place.factory.place(1)) S3;
async at(place.factory.place(1)) S4;
async at(place.factory.place(1)) S5;
```



```
async at(place.factory.place(2)) S6;
async at(place.factory.place(2)) S7;
async at(place.factory.place(2)) S8;
```

```
async at(place.factory.place(3)) S9;
async at(place.factory.place(3)) S10;
```

Example HJ program with places

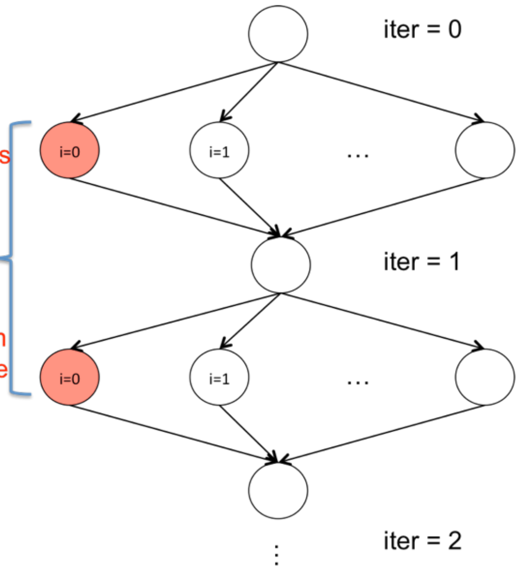
```
class T1 {
  final place affinity;
  . . .
  // T1's constructor sets affinity to place where instance was created
  T1() { affinity = here; ... }
  . . .
}
. . .
finish { // Inter-place parallelism
  System.out.println("Parent␣place␣=␣", here); // Parent task s place
  for (T1 a = . . .) {
    async at (a.affinity) { // Execute async at place with affinity to a
      a.foo();
      System.out.println("Child␣place␣=␣", here); // Child task's place
    } // async
  } // for
} // finish
. . .
```

Chunked Fork-Join Iterative Averaging Example with Places

```
public void runDistChunkedForkJoin(int iterations,
                                   int numChunks, dist d) {
  for (int iter = 0; iter < iterations; iter++) {
    finish for (point [jj] : [0:numChunks-1])
      async at(d.get(jj) {
        for (point [j] : getChunk([1:n],numChunks,jj))
          myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
    } // finish-for-async
    double[] temp = myNew; myNew = myVal; myVal = temp;
  } // for iter
} // runDistChunkedForkJoin
```

- Chunk jj is always executed in the same place for each iteration, iter
- Method runDistChunkedForkJoin can be called with 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

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

Distributions --- hj.lang.dist

- A distribution maps points in a rectangular index space (region) to places e.g.,
  - i → place.factory.place(i % place.MAX_PLACES-1)
- Programmers are free to create any data structure they choose to store and compute these mappings
- For convenience, the HJ language provides a predefined type, hj.lang.dist, to simplify working with distributions
- Some public members available in an instance d of hj.lang.dist are:
  - d.rank = number of dimensions in the input region for distribution d
  - d.get(p) = place for point p mapped by distribution d. It is an error to call d.get(p) if p.rank != d.rank.
  - d.places() = set of places in the range of distribution d
  - d.restrictToRegion(pl) = region of points mapped to place pl by distribution d

Block Distribution

- dist.factory.block([lo:hi]) creates a block distribution over the one-dimensional region, lo:hi.
- A block distribution splits the 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.

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

Block Distribution (contd)

- If the input region is multidimensional, then a block distribution is computed over the linearized one-dimensional version of the multidimensional region
- Example in Table 2: dist.factory.block([0:7,0:1]) for 4 places

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0				1				2				3			

Distributed Parallel Loops

- Listing 2 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

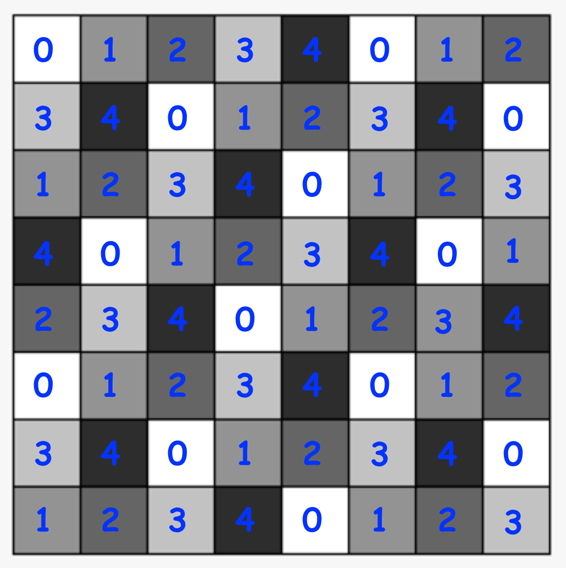
- dist.factory.cyclic([lo:hi]) creates a cyclic distribution over the one-dimensional region, lo:hi.
- A cyclic distribution "cycles" through places 0 … place.MAX PLACES – 1 when spanning the input region
- Cyclic distributions can improve the performance of parallel loops that exhibit load imbalance
- Example in Table 3: dist.factory.cyclic([0:15]) for 4 places

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

- Example in Table 4: dist.factory.cyclic([0:7,0:1]) for 4 places

Index	[0,0]	[0,1]	[1,0]	[1,1]	[2,0]	[2,1]	[3,0]	[3,1]	[4,0]	[4,1]	[5,0]	[5,1]	[6,0]	[6,1]	[7,0]	[7,1]
Place id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Figure 1: Cyclic distribution for a 8×8 sized region (e.g., [1:8,1:8]) mapped on to 5 places



0	1	2	3	4	0	1	2
3	4	0	1	2	3	4	0
1	2	3	4	0	1	2	3
4	0	1	2	3	4	0	1
2	3	4	0	1	2	3	4
0	1	2	3	4	0	1	2
3	4	0	1	2	3	4	0
1	2	3	4	0	1	2	3

Figure source: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder, http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

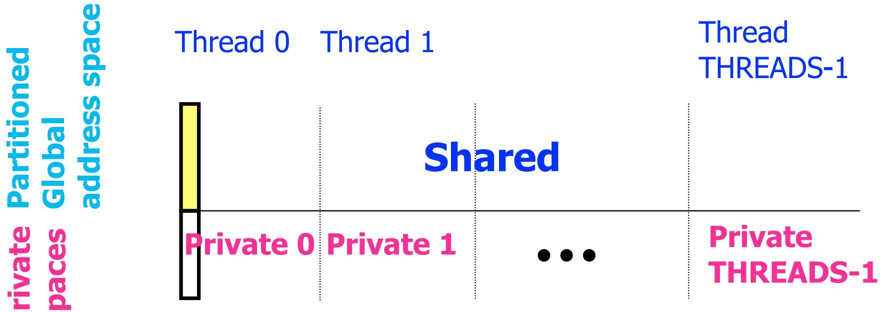
Block-Cyclic Distribution

- dist.factory.blockCyclic([lo:hi],b) creates a block-cyclic distribution over the one-dimensional region, lo:hi.
- A block-cyclic distribution combines the locality benefits of the block distribution with the load-balancing benefits of the cyclic distribution by introducing a block size parameter, b.
- The linearized region is first decomposed into contiguous blocks of size b, and then the blocks are distributed in a cyclic manner across the places.
- Example in Table 5: dist.factory.blockCyclic([0:15]) for 4 place

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

Data Distributions

- In HJ, distributions are used to guide computation mappings for affinity
- The idea of distributions was originally motivated by mapping data (array elements) to processors
- e.g., Unified Parallel C (UPC) language for distributed-memory parallel machines (Thread = Place)



- A pointer-to-shared can reference all locations in the shared space, but there is data-thread affinity

Affinities for Shared and Private Data in UPC

Examples of Shared and Private Data Layout:

Assume THREADS = 3

```
shared int x;   /*x will have affinity to thread 0 */
shared int y[2*THREADS]; /* cyclic distribution by default
   */
int z; /* private by default */
```

will result in the layout:

Thread 0	Thread 1	Thread 2
☐ x		
☐ y[0]	☐ y[1]	☐ y[2]
☐ y[3]	☐ y[4]	☐ y[5]
☐ z	☐ z	☐ z

Shared and Private Data

`shared int A[4][THREADS];`

will result in the following data layout:

Thread 0	Thread 1	Thread 2
A[0][0]	A[0][1]	A[0][2]
A[1][0]	A[1][1]	A[1][2]
A[2][0]	A[2][1]	A[2][2]
A[3][0]	A[3][1]	A[3][2]

Block-Cyclic Distributions for Shared Arrays

- Default block size is 1
- Shared arrays can be distributed on a block per thread basis, round robin with arbitrary block sizes.
- A block size is specified in the declaration as follows:
  - `shared [block-size] type array[N];`
  - **e.g.:** `shared [4] int a[16];`

Shared and Private Data

Assume THREADS = 4

`shared [3] int A[4][THREADS];`

will result in the following data layout:

Thread 0	Thread 1	Thread 2	Thread 3
A[0][0]	A[0][3]	A[1][2]	A[2][1]
A[0][1]	A[1][0]	A[1][3]	A[2][2]
A[0][2]	A[1][1]	A[2][0]	A[2][3]
A[3][0]	A[3][3]		
A[3][1]			
A[3][2]			

Announcements (REMINDER)

- Homework 3 due on Wednesday, Feb 22nd
  - —Performance results for parts 2 and 3 of assignment must be obtained on Sugar (see Section 4)
- No lab this week
  - —Use the time for HW3 and to prepare for Exam 1
- Exam 1 will be held in the lecture on Friday, Feb 24th
  - —Closed book 50-minute exam
  - —Scope of exam includes lectures up to Monday, Feb 20th
  - —Feb 22nd lecture will be a midterm review before exam