COMP 322: Fundamentals of Parallel Programming

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

Lecture 23: Places and Distributions

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

Acknowledgments for Today's Lecture

- Lecture 23 handout
- 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
- "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
  - Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

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**

Note that **here** in a child task for an async/future computation will refer to the place P at which the child task is executing, not the place where the parent task is executing

Listing 1: Batched Async-Finish Iterative Averaging Example with Places

```
for (point [iter] : [0:iterations-1]) {
  finish for (point [i] : [0:tasks-1]) {
    async at(place.factory.place(i % place.MAX_PLACES)) {
      int start = i * batchSize + 1;
      for (point [j] : [start:Math.min(start+batchSize-1,n)]) {
        myNew[j] = (myVal[j-1] + myVal[j+1]) / 2.0;
      }
    } // async
  } // finish for
  double[] temp = myNew; myNew = myVal; myVal = temp;
}
```

- Assume a –places 4:4 configuration with 4 places and 4 workers per places for execution on a 16-core machine
  - Set tasks = 16 so as to create one async per worker
  - Use i % place.MAX_PLACES to compute destination place for each async

  ➔ Each subarray is processed at same place for successive iterations of for-iter loop

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

- 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 as follows
  - 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.
- Example in Table 1: dist.factory.block([0:15]) for 4 places

<table>
<tr><td>Index</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td>Place id</td><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">2</td><td colspan="4">3</td></tr>
</table>

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

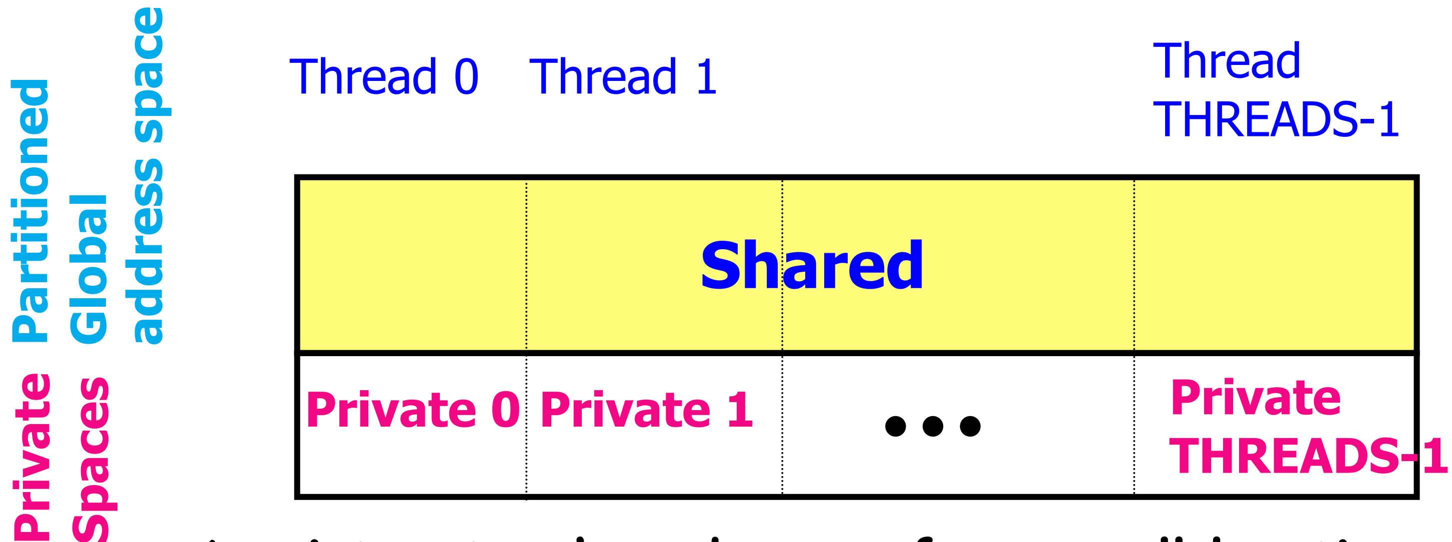
Block-Cyclic Distribution

- 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 distri-bution 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 with block size b = 2

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 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

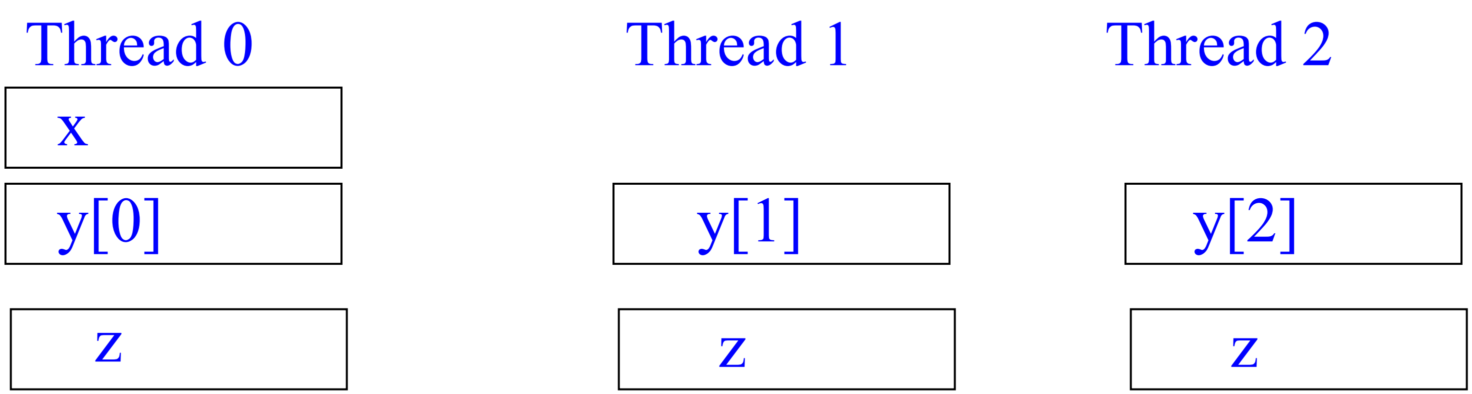
Shared and Private Data

Examples of Shared and Private Data Layout:

Assume THREADS = 3

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

will result in the layout:



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]