coursera

3.5 Lecture Summary

3 Loop Parallelism

3.5 Iteration Grouping: Chunking of Parallel Loops

Lecture Summary: In this lecture, we revisited the vector addition example:

```
forall (i : [0:n-1]) a[i] = b[i] + c[i]
```

We observed that this approach creates *n* tasks, one per *forall* iteration, which is wasteful when (as is common in practice) *n* is much larger than the number of available processor cores.

To address this problem, we learned a common tactic used in practice that is referred to as *loop chunking* or *iteration grouping*, and focuses on reducing the number of tasks created to be closer to the number of processor cores, so as to reduce the overhead of parallel execution:

With iteration grouping/chunking, the parallel vector addition example above can be rewritten as follows:

```
forall (g:[0:ng-1])
  for (i : mygroup(g, ng, [0:n-1])) a[i] = b[i] + c[i]
```

Note that we have reduced the degree of parallelism from *n* to the number of groups, *ng*, which now equals the number of iterations/tasks in the *forall* construct.

There are two well known approaches for iteration grouping: *block* and *cyclic*. The former approach (*block*) maps consecutive iterations to the same group, whereas the latter approach (*cyclic*) maps iterations in the same congruence class (mod *ng*) to the same group. With these concepts, you should now have a better understanding of how to execute *forall* loops in practice with lower overhead.

For convenience, the PCDP library provides helper methods, *forallChunked()* and *forall2dChunked()*, that automatically create one-dimensional or two-dimensional parallel loops, and also perform a block-style iteration grouping/chunking on those parallel loops. An example of using the *forall2dChunked()* API for a two-dimensional parallel loop (as in the matrix multiply example) can be seen in the following Java code