Slide 1

Administrivia

- (Review minute essay from previous lecture.)

Slide 2

Example Application: Matrix Multiplication

- Basic problem is straightforward: For two N by N matrices A and B, compute the matrix product C with elements defined thus:

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$$

  (Actually A and B don't have to be square and the same size, but for the moment let's assume they are.)

- Less obvious approach: Decompose A, B, and C into blocks and think of the calculation in terms of these blocks (equation similar to the above, but for blocks rather than individual elements).

  Why? often makes better use of cache and therefore is faster.

Slide 3

Parallelization — Understanding the Problem

- In the simple approach, the code is just nested loops over the elements of C. A block-based approach is slightly more complicated, but not a great deal. (Look at example code, performance results.)
- Consider parallelizing for first shared-memory and then distributed-memory environments.

Slide 4

Parallelization — *Finding Concurrency*

- Obvious decomposition for simple approach is task-based, with one task per point. Tasks are completely independent.
- For block-based approach, may make more sense to think in terms of decomposing data into blocks; then tasks correspond to computing blocks of C. Again, though, they're independent.

Slide 5

Parallelization — *Algorithm Structure*

- For the simple approach, we have many mostly-independent tasks, forming a flat set rather than a hierarchy, so *Task Parallelism* seems like a good choice.
- Key design decision is how to assign tasks to UEs.
- Probably makes sense to group tasks by rows rather than individual points and to use a simple static assignment of tasks to UEs.
- For shared memory, that's about it.

Slide 6

Parallelization — *Algorithm Structure*, Continued

- For distributed memory, however, we have to think about how to distribute C and how to duplicate/distribute A and B. Might work better to think in terms of block-based approach and data decomposition — so *Geometric Decomposition* might be a better fit.

  (Recall — *Geometric Decomposition* focuses on simultaneous update of chunks of a large data structure. Compare how this problem fits that with how the heat-diffusion problem fits it.)
- Key design decisions here are how to decompose data and assign chunks to UEs, and then how to manage synchronization/communication for update operation.
- Probably makes sense to decompose data so we can assign one block of C to each UE — amount of work per block is pretty much constant.
- For each block of C, computation can be thought of a sequence of update

operations, each involving a different combination of blocks of A and B. This tells us what kind of communication we need.

Slide 7

Parallelization — *Supporting Structures*

- For shared memory, *Loop Parallelism* makes sense; for distributed memory, we probably want *SPMD*.
- Probably we will also want to look at *Distributed Array*.

Slide 8

Slide 9

Parallelization — Code

- (Look at code, multiple versions.)

Slide 10

Minute Essay

- TBA