Slide 1

Administrivia

- Reminder: Homework 3 due today.
- Homework 4 to be on the Web soon (tomorrow?). I will send mail.

Slide 2

Example Application — Mergesort

- Mergesort should be familiar from other courses. Sequential algorithm is divide-and-conquer, and solution of subproblems is independent, so this fits our *Divide and Conquer* pattern.
- (Review code.)

Slide 3

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 (assuming 0-based indexing):

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j}$$

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

- Simple approach to calculating this is obvious — just do the above calculation for all i and j between 0 and $N - 1$.
- 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 4

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 5

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 6

Parallelization — *Algorithm Structure* (Shared Memory)

- 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. Block-based program is similar.
- 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, and group tasks by — what? for simple approach, two obvious choices; for block-based approach, more. Could try several and see which seems to work best.

Slide 7

Parallelization — *Supporting Structures* and Code (Shared Memory)

- For program structure, *Loop Parallelism* makes sense.
- Code in OpenMP is very straightforward (see example code).

Slide 8

Parallelization — *Algorithm Structure* (Distributed Memory)

- For distributed memory 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.
- 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.

Slide 9

Parallelization — *Algorithm Structure* (Distributed Memory), Continued

- 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. (Compare how this fits overall idea of *Geometric Decomposition* with how heat-diffusion example fits.)
- This tells us what kind of communication we need. (Simple approach is to broadcast two blocks at each step, one for "row" and one for "column". More complex, but more efficient, version involves rotating blocks among processes.)

Slide 10

Parallelization — *Supporting Structures* (Distributed Memory)

- For program structure, we probably want *SPMD* (especially if using MPI or similar programming environment).
- *Distributed Array* is relevant, especially for parts of sample/test program that initialize and print array (since they use each array element's global indices).

Slide 11

Parallelization — Code (Distributed Memory)

- If we distribute all three arrays (which seems like a good idea), we have to make changes in code to initialize and print as well as matrix-multiplication. As is often the case with programs using *Distributed Array*, the ideas are simple but the code inclined to be messy.
- For actual multiplication, each process will update one "chunk", doing the same computation done in the block-based sequential program, but with communication operations to broadcast two blocks per step.
- Look at example code . . . .

Slide 12

Minute Essay

- None — sign in.