Slide 1

Administrivia

- (None?)

Slide 2

Example Application: Mandelbrot Set (Review)

- For each point $c = a + bi$ in the complex plane, look at the sequence $z_0, z_1, z_2, \ldots$, where

$$\begin{aligned} z_0 &= 0 \\ z_{k+1} &= {z_k}^2 + c \end{aligned}$$

- For some points, this sequence is "quasi-stable" ($|z_k|$ bounded); for others, it's not.
- We can get interesting pictures by discretizing and then computing, for each point, how long it takes this sequence to "diverge".

Parallelization — Understanding the Problem

- Code is nested loops over points in a 2D space, where at each point we calculate until divergence / maximum iterations and then plot the result (to something implicitly or explicitly shared).
- Consider parallelizing. . .

Slide 3

Parallelization — *Finding Concurrency*

- Task-based decomposition seems logical. (Why not geometric? could do that too, though it sort of comes to the same thing.) Consider calculations for one point as a task.
- How do the tasks depend on each other? they don't really, except that "plotting" a result means doing something with a shared resource.

Slide 4

Slide 5

Parallelization — *Algorithm Structure*

- Many mostly-independent tasks, forming a flat set rather than a hierarchy, so *Task Parallelism* seems like a good choice.
- Key design decisions are how to assign tasks to UEs, how to manage "plotting".
- Probably makes sense to group tasks by rows rather than individual points. We could try a simple static distribution, but might have to do something more complex if that doesn't give good load balance.
- Managing plotting?

  With shared memory, should work if we make sure only one thread at a time can plot.

  With distributed memory, might make sense to just assign plotting to a process that does nothing else.

Slide 6

Parallelization — *Supporting Structures*

- For shared memory, *Loop Parallelism*.
- For distributed memory, *SPMD*, but with elements of *Master/Worker* (a master process to manage the computation and the displays, and workers to do the calculations).
- (Look at code.)

Slide 7

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 8

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.
- Consider parallelizing for first shared-memory and then distributed-memory environments.
- (To be continued next time.)

Minute Essay

- None really — just sign in, unless questions?

Slide 9