Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Homework 5 coming soon. (Preview: Parallelize Java quicksort. I will show Java mergesort in class soon.)

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 (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 3

Sidebar: Info About Hardware

- My Web site

  `https://sites.google.com/trinity.edu/csci-department-computers/specifics`

  has some information about our equipment.
- Linux pseudofile `/proc/cpuinfo` has more about processors.
- Command `lstopo` shows info about caches etc. (First load module `hwloc-latest`.)

Slide 4

Matrix Multiplication — Code

- (We looked at code last time, some.)
- 2D array represented as 1D array, to make it easier(?) to locate elements of blocks: Each row of a block is contiguous, and rows are separated by "stride".
- Functions to work on blocks (one to clear a block, one to multiply and add to running total) use as arguments block start, size, stride.
- Timing can be tedious — since how this works depends on some details of hardware (e.g., cache), try more than one type of machine, different problem sizes. Shell scripts help! (Review data I collected. Some results make the point. Note that times on some platforms vary from execution to execution. No idea why! Others puzzling but at least show why multiple executions advisable.)

Slide 5

Parallelization — Understanding the Problem (Review)

- In simple approach, code is just nested loops over the elements of C. A block-based approach is slightly more complicated, though not a lot.
- Consider parallelizing for first shared-memory and then distributed-memory environments.

Slide 6

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 7

Parallelization — *Algorithm Structure* (Shared Memory)

- For simple approach, 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. Simple static assignment seems right, but details — ?
- For simple approach, could group tasks by rows and assign rows to UEs.

Slide 8

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

- For block-based approach, probably want to assign (groups of) blocks to UEs: Multithreading within block seems like might lose improvement in use of cache?
- Giving each UE rows of blocks is simple but may limit concurrency too much.
- Giving each UE individual blocks is more trouble but less limiting.

Slide 9

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

- For program structure, *Loop Parallelism* makes sense.
- Code in OpenMP is straightforward.
- (Look at code, timing data.)

Slide 10

Parallelization — *Algorithm Structure* (Distributed Memory)

- For distributed memory, 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 11

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 12

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 13

Parallelization — Code (Distributed Memory)

- If we distribute all three arrays (which seems like a good idea), 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*, ideas are simple but code inclined to be messy!
- For actual multiplication, each process will update one "chunk", doing same computation done in the block-based sequential program, but with communication operations to broadcast two blocks per step.
- (Look at code, timing data. Note printing strategy: Each process prints its data, but with global indices; then can use external tools (`cat`, `sort`) to combine. Good idea to do a small test run for correctness in addition to timing.)

Slide 14

Minute Essay

- I had planned for the last assignment in the course to be a project, but we're running out of time. Should we just not do this, or do something small-scale? (Try to think about learning outcomes as well as workload!)