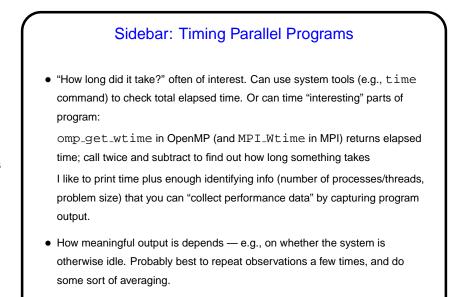
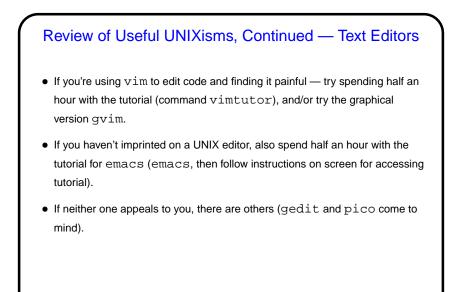


Homework 2, Again
For Homework 2 you are revising your programs from Homework 1 and trying to be more systematic about evaluating them.
So, some comments about tools, etc., that may be useful ...



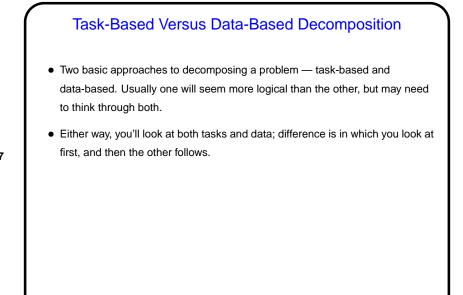
Review of Useful UNIXisms
Remember that you can capture output of program foo in file foo.out by typing: foo >foo.out (to overwrite) foo >>foo.out (to append) Append 2>&1 to capture standard error too.
Slide 4
You can have output go both to the screen and a file by typing: foo | tee foo.out (to overwrite) foo | tee foo.out (to overwrite) foo | tee -a foo.out (to append)
Remember that if you have commands you want to execute multiple times, you can make a script — e.g., put the commands in file bar and execute by typing sh bar.
Script mpirun-on-upmachines on sample programs page may help you, if you want to use some of the dual-boot lab machines.



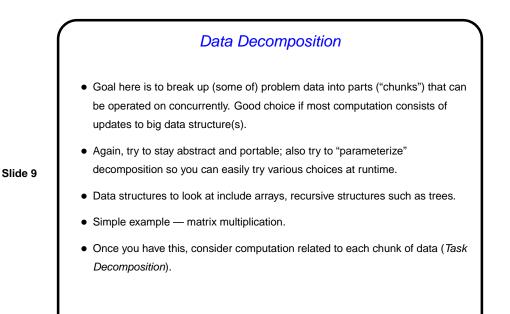
Finding Concurrency Design Space

Starting point in our grand strategy for developing parallel applications.
 Overall idea — capture how experienced parallel programmers think about initial design of parallel applications. Might not be necessary if clear match between application and an *Algorithm Structure* pattern.

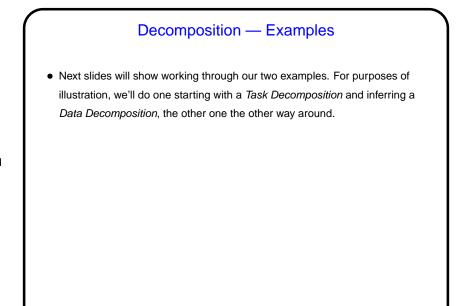
- Idea is to work through three groups of patterns in sequence (possibly with backtracking):
 - Decomposition patterns (*Task Decomposition*, *Data Decomposition*): Break problem into tasks that maybe can execute concurrently.
 - Dependency analysis patterns (Group Tasks, Order Tasks, Data Sharing):
 Organize tasks into groups, analyze dependencies among them.
 - Design Evaluation: Review what you have so far, possibly backtrack.
- Keep in mind best to focus attention on computationally intensive parts of problem.



*Task Decomposition*Goal here is to break up (some of) computation into "tasks" — logical elements of overall computation that might be independent enough to do concurrently. At this stage, try to stay abstract and portable; also try to identify lots of tasks (can always recombine them later if too many), as independent of each other as possible. Places to look for tasks include groups of function calls (e.g., in divide-and-conquer strategy), loop iterations (e.g., many examples we've discussed). Simple example — matrix multiplication. Once you have this, consider data related to each task (*Data Decomposition*).



Examples (from 2/17) — Review Simplified generic molecular dynamics application — timestepped simulation, where in each step we compute forces on all atoms (O(N²) unless we ignore distant particles) and then use forces to recompute positions and velocities. Heat-diffusion equation — repeatedly compute new values for each point based on old value of it and neighbors, continuing until we get convergence.

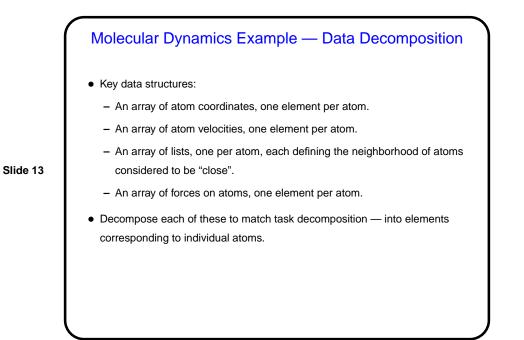


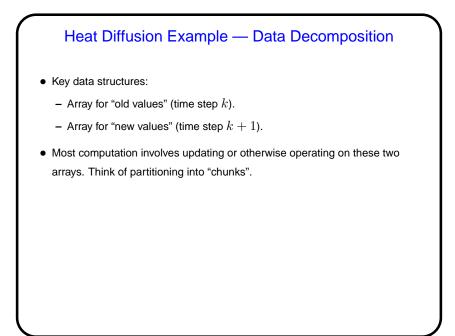
Molecular Dynamics Example — Task Decomposition

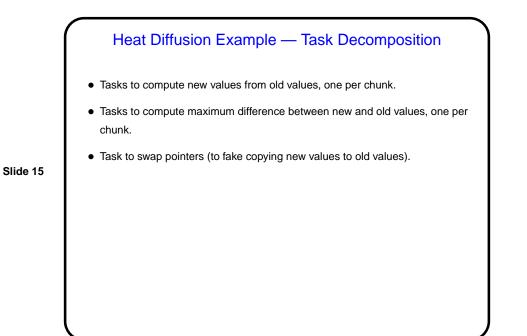
- Tasks that find the vibrational forces on an atom.
- Tasks that find the rotational forces on an atom.

(Together, these are tasks to compute "bonded forces" — those due to chemical bonds.)

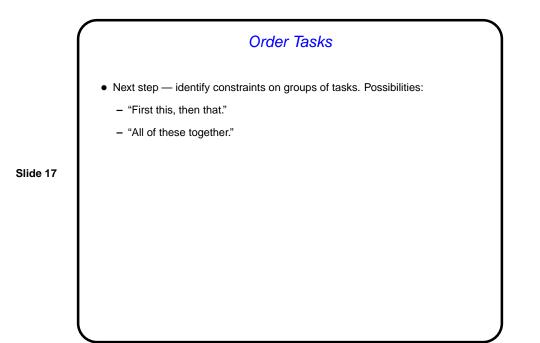
- Tasks that find the non-bonded forces on an atom (the ones due to electrical charges).
- Tasks that update the position and velocity of an atom.
- Tasks that update the neighbor list for an atom. (Or we could consider updating all the neighbor lists as one task, as in the book, if we think it won't be done very often and therefore is not worthwhile to parallelize.)

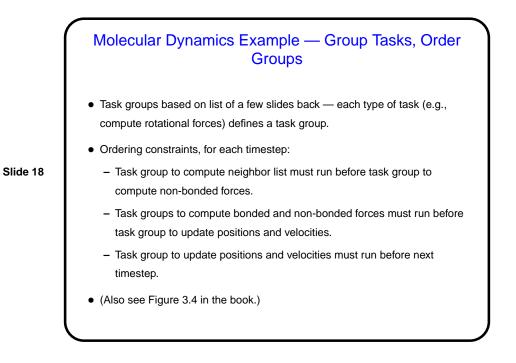


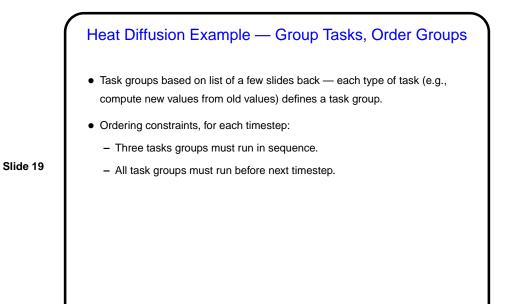


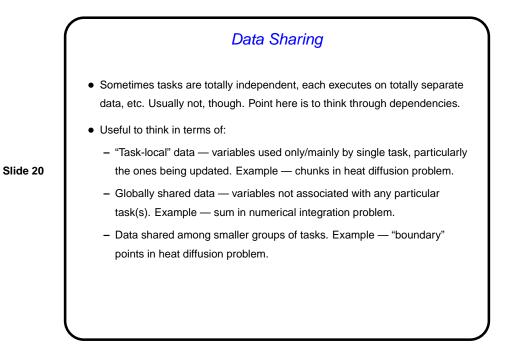


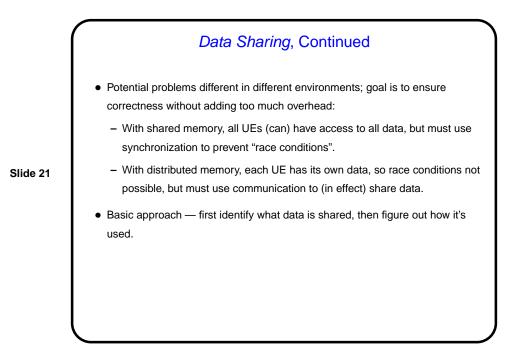
Group Tasks • Once you've broken down problem into tasks / data chunks, need to put it back together as design for parallel algorithm. • First step — look for "groups of tasks" — logically related, or interdependent, or all with same constraints, etc. Often just one group. Slide 16

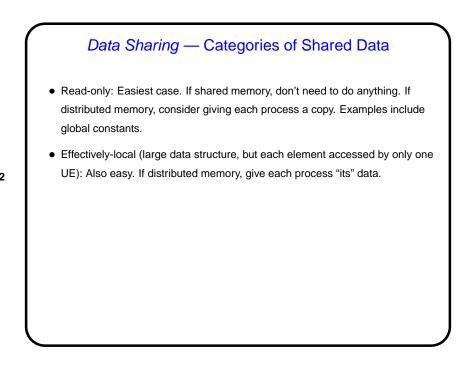












Data Sharing — Categories of Shared Data, Continued
Read-write (accessed by more than one task, at least one changing it): Can be arbitrarily complicated, but some common cases aren't too bad:

"Accumulate" (variable(s) used to accumulate result — usually a reduction). Example — sum in numerical integration problem. Give each task (or each UE) a copy and combine at end.
"Multiple-read/single-write" (multiple tasks need initial value, one task computes new value). Example — points near boundaries of chunks in heat diffusion problem. Create at least two copies, one for task that computes new value, other(s) to hold initial value for other tasks.

Molecular Dynamics Example — Analyze Task/Data Dependencies

- Arrays of atom positions, velocities:
 - Read-only for most groups of tasks but tasks may need access to many elements, so for distributed memory might want to duplicate.

- Updated by one group of tasks, but each task updates its own element(s)
 "effectively local".
- Array of forces:
 - Read-only for group of tasks that update positions and velocities, and each task needs access only to "local" data.
 - Updated by several groups of tasks, but updates fit "accumulate data" model.



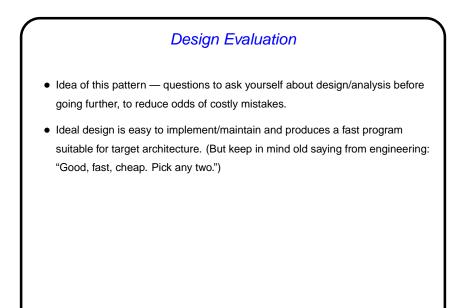
- Array of neighbor lists:
 - Read-only for group of tasks that compute "non-bonded" forces, and each task needs access only to local data.
 - Updated by one group of tasks, but each task updates its own element(s).
- (Also see Figure 3.5 in book.)

Heat Diffusion Example — Analyze Task/Data Dependencies

- Arrays of old, new values:
 - Old values read-only for all groups of tasks, and each task needs access mostly to local data — plus "boundary values" for neighboring tasks.
 - New values updated by one group of tasks, and each task computes values only for "its" elements.

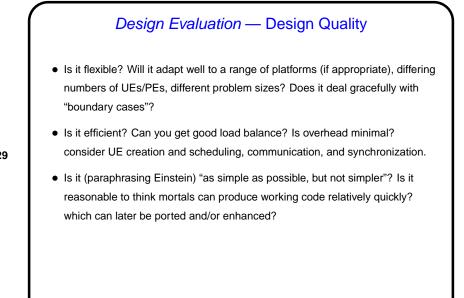
For distributed memory, could distribute among processes, with extra variable(s) to hold copy of boundary values.

- Maximum difference between old, new values is "accumulate data" in one group of tasks, read-only elsewhere.
- Pointers to old/new values changed at end of time step by one task, read-only elsewhere. Could duplicate for distributed memory.



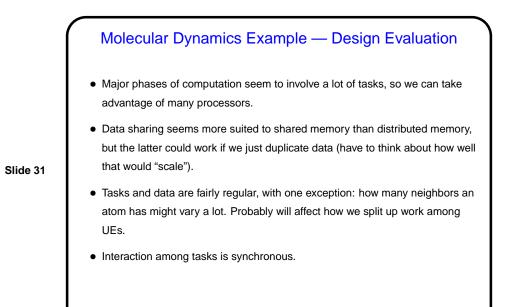
Design Evaluation — Suitability for Target Platform

- How many processing elements (PEs) are available? Need at least one task per PE, often want many more — unless we can easily get exactly one task per PE at runtime, with good load balance.
- How are data structures shared among PEs? If there's a lot of shared data, or sharing is very "fine-grained", implementing for distributed memory will likely not be easy or fast.
- How many UEs are available and how do they share data? Similar to previous questions, but in terms of UEs — with some architectures, can have multiple UEs per PE, e.g., to hide latency. For this to work, "context switching" must be fast, and problem must be able to take advantage of it.
- How does time spent doing computation compare to overhead of synchronization/communication, on target platform? May be a function of problem size relative to number of PEs/UEs.



Design Evaluation — Preparation for Next Phase

- How regular are tasks and their data dependencies?
- Are interactions between tasks (or groups of tasks) synchronous or asynchronous?
- Are tasks grouped in the best way?



Heat Diffusion Example — Design Evaluation

- Major phases of computation seem to involve a lot of tasks, so we can take advantage of many processors.
- Data sharing seems suitable for either shared or distributed memory.
- Tasks and data are very regular, interaction is synchronous.

