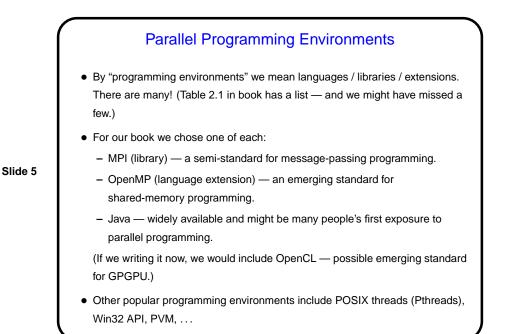


What Programming Languages Support This?, Continued

- A regular sequential language with a parallelizing compiler: Attractive, but such compilers are not easy.
- Slide 4 Per
- A language designed to support parallel programming (Java, Ada, PCN): Perhaps the most expressive, but more work for programmers and implementers.
 - A regular sequential language plus calls to parallel library functions (PVM, MPI, Pthreads): More familiar for users, easier to implement.
 - A regular sequential language with some added features (CC++, OpenMP): Also familiar for users, can be difficult to implement.



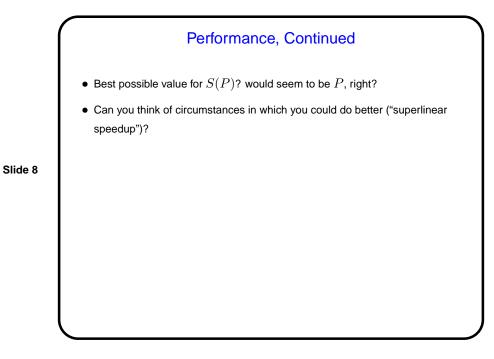
Sketch of Parallel Algorithm Development

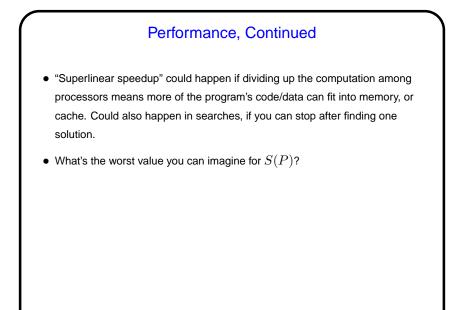
- Start with understanding of problem to be solved / application.
- Decompose computation into "tasks" snippets of sequential code that you might be able to execute concurrently.
- Analyze tasks and data how do tasks depend on each other? what data do they access (local to task and shared)?

(Or start with decomposition of data and infer tasks from that.)

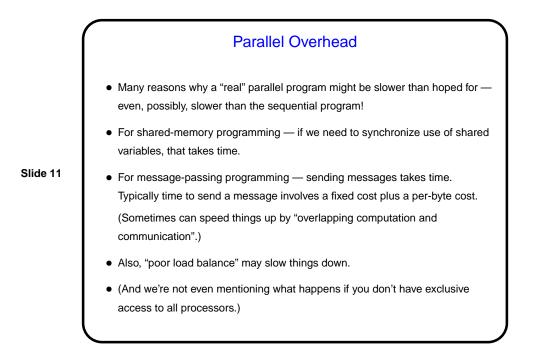
- Plan how to map tasks onto "units of execution" (threads/processes) and coordinate their execution. Also plan how to map these onto "processing elements".
- Translate this design into code.
- Our book organizes all of this into four "design spaces", corresponding to (we think) steps in program design / development.

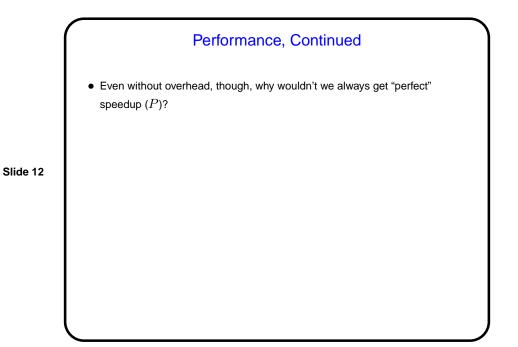
A Few Words About Performance • If the point is to "make the program run faster" — can we quantify that? • Sure. Several ways to do that. One is "speedup" — $S(P) = \frac{T_{total}(1)}{T_{total}(P)}$ • What's the best possible value you can imagine for S(P)?





Performance, Continued
Worst possible value would seem to be 1, right?
Can you think of circumstances in which you'd do worse? (Hint: What do you know so far about how the parts of the program running on different cores/processors/machines interact?)
Slide 10





Amdahl's Law

 And most "real programs" have some parts that have to be done sequentially. Gene Amdahl (principal architect of early IBM mainframe(s)) argued that this limits speedup — "Amdahl's Law":

If γ is the "serial fraction", speedup on P processors is (at best — this ignores overhead)

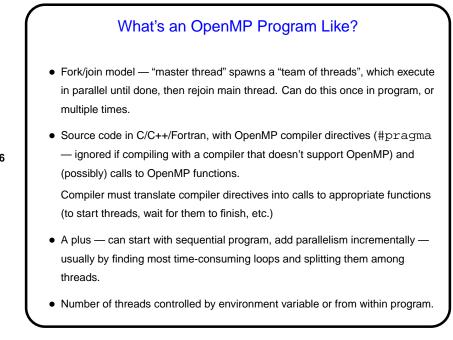
$$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

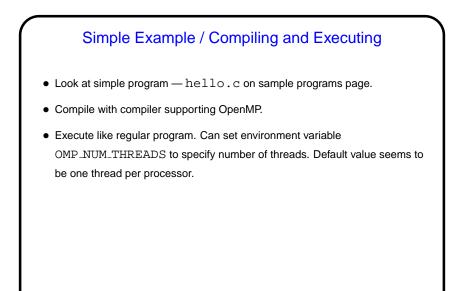
and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup. (Details of math in chapter 2.)

What's Next - Nuts and Bolts

• So we can start writing programs as soon as possible, next topic will be a fast tour through the three programming environments we will use for writing programs. (Some possibility of also including OpenCL.)

OpenMP
Early work on message-passing programming resulted in many competing programming environments — but eventually, MPI emerged as a standard.
Similarly, many different programming environments for shared-memory programming, but OpenMP may be emerging as a standard.
In both cases, idea was to come up with a single standard, then allow many implementations. For MPI, standard defines concepts and library. For OpenMP, standard defines concepts, library, and compiler directives.
First release 1997 (for Fortran, followed in 1998 by version for C/C++).
Several production-quality commercial compilers available. Early on, free compilers were, um, "research software" or in work. Recent versions of GNU compilers, though, offer support. !!





Sidebar — make and makefiles
Compiling with non-default options (as you must do to compile OpenMP programs with gcc) can become tedious.
make can help. Briefly — it's a very old UNIX tool intended to help automate building large programs. Can be used in different ways, but one of them is simply to make it easy to compile with non-default options.
To use make, set up Makefile (example on from "Sample programs" Web page), and then type make foo to compile foo.c to foo.



