Knowing When to Parallelize

Rules-of-Thumb based on User Experiences

Cherri M. Pancake

Department of Computer Science

Oregon State University

pancake@cs.orst.edu

Oregon State University

What is Parallelism?

- Consider your favorite computational application
  - One processor can give me results in N hours
  - Why not use N processors
    -- and get the results in just one hour?

The concept is simple:

Parallelism = applying multiple processors to a single problem

- Reasons for using parallelism
  - Get results faster
  - Solve bigger problems
  - Run simulations at finer resolutions
  - Model physical phenomena more realistically

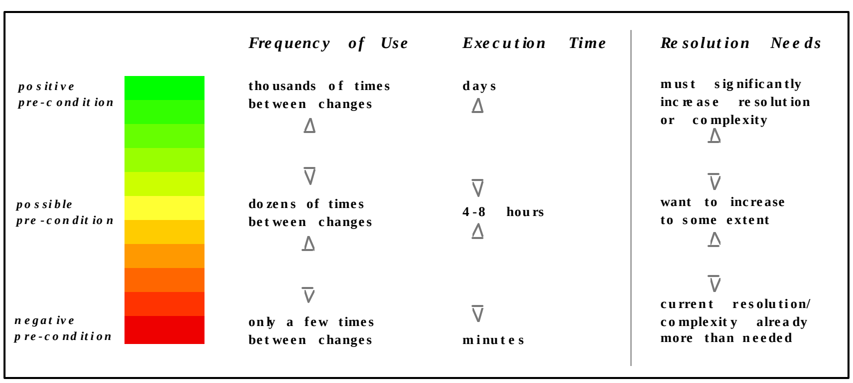
Parallelism Carries a Price Tag

- Parallel programming
  - Involves a steep learning curve
  - Is effort-intensive
- Parallel computing environments are unstable and unpredictable
  - Don't respond to many serial debugging and tuning techniques
  - May not yield the results you want, even if you invest a lot of time

Will the investment of your time be worth it?

- Rules-of-thumb
  - Drawn from the experiences of hundreds of computational scientists and engineers
  - Encapsulate their "tricks" for knowing when to parallelize (and when to keep their applications serial)

Test the "Preconditions for Parallelism"



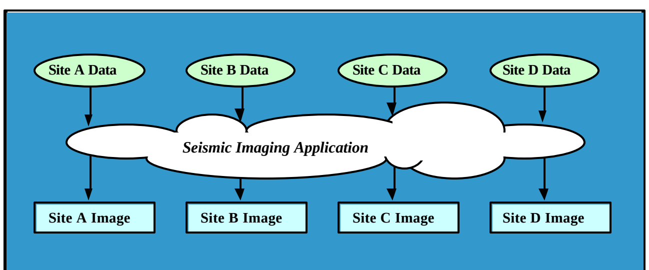
- According to experienced parallel programmers:
  - *no green* - Don't even consider it
  - *one or more red* - Parallelism may cost you more than you gain
  - *all green* - You need the power of parallelism (but there are no guarantees)

How Your Problem Affects Parallelism

- The nature of your problem constrains how successful parallelization can be
- Consider your problem in terms of
  - When data is used, and how
  - How much computation is involved, and when
- Geoffrey Fox identified the importance of <u>problem architectures</u>
  - Perfectly parallel
  - Fully synchronous
  - Loosely synchronous
- A fourth problem style is also common in scientific problems
  - Pipeline parallelism

Perfect Parallelism

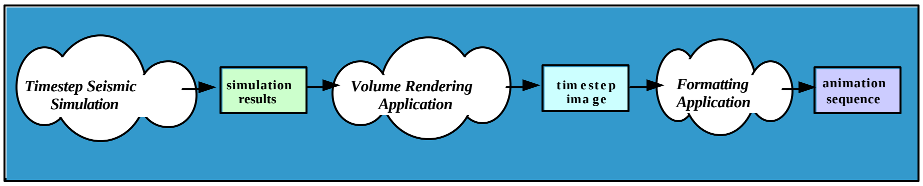
- Scenario: seismic imaging problem
  - Same application is run on data from many distinct physical sites
  - Concurrency comes from having multiple datasets processed at once
  - Could be done on independent machines (if data can be available)



- This is the simplest style of problem
- Key characteristic: <u>calculations for each data set are independent</u>
  - Could divide/replicate data into files and run as independent serial jobs
  - (also called "job-level parallelism")

Pipeline Parallelism

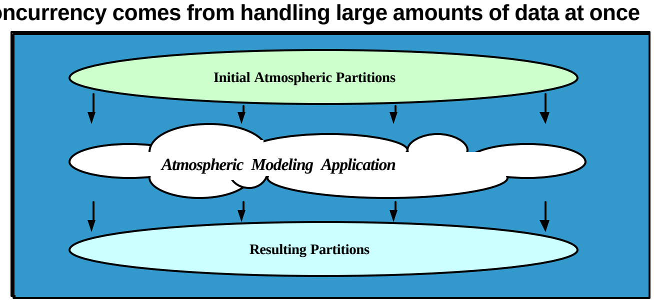
- Scenario: seismic imaging problem
  - Data from different time steps used to generate series of images
  - Job can be subdivided into phases which process the output of earlier phases
  - Concurrency comes from overlapping the processing for multiple phases



- Key characteristic: only need to pass results one-way
  - Can delay start-up of later phases so input will be ready
- Potential problems
  - Assumes phases are computationally balanced
  - (or that processors have unequal capabilities)

Fully Synchronous Parallelism

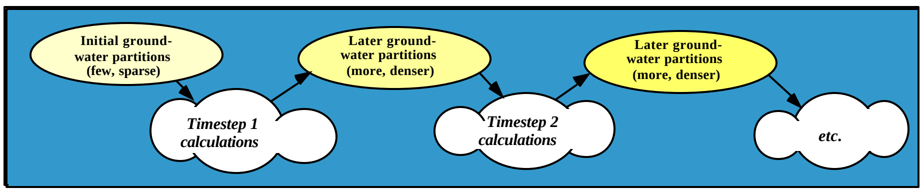
- Scenario: atmospheric dynamics problem
  - Data models atmospheric layer; highly interdependent in horizontal layers
  - Same operation is applied in parallel to multiple data
  - Concurrency comes from handling large amounts of data at once



- Key characteristic: Each operation is performed on all (or most) data
  - Operations/decisions depend on results of previous operations
- Potential problems
  - Serial bottlenecks force other processors to "wait"

Loosely Synchronous Parallelism

- Scenario: diffusion of contaminants through groundwater
  - Computation is proportional to amount of contamination and geostructure
  - Amount of computation varies dramatically in time and space
  - Concurrency from letting different processors proceed at their own rates

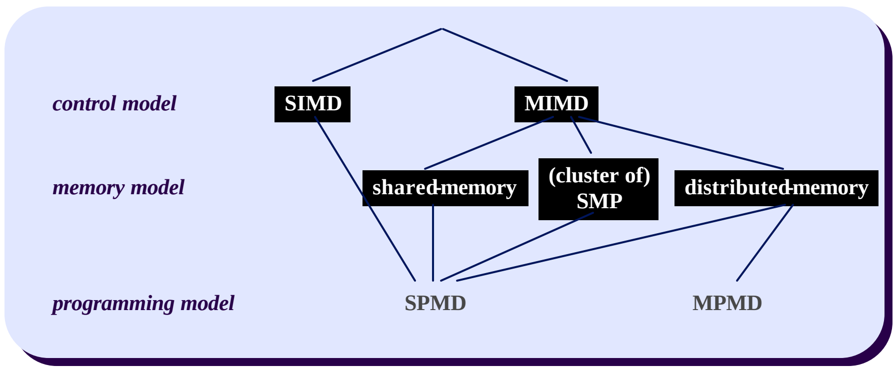


- Key characteristic: Processors each do small pieces of the problem, <u>sharing information only intermittently</u>
- Potential problems
  - Sharing information requires "synchronization" of processors (where one processor will have to wait for another)

Rules-of-Thumb Based on Type of Problem

- If your application fits the model of perfect parallelism
  → the parallelization task is relatively straightforward and likely to achieve respectable performance
- If your application is an example of pipeline parallelism
  → you have to do more work. If you can't balance the computational intensity, it may not prove worthwhile
- If your application is fully synchronous
  → a significant amount of effort is required and payoff may be minimal. The decision to parallelize should be based on how uniform computational intensity is likely to be
- A loosely synchronous application is the most difficult to parallelize
  → it's probably not worthwhile unless the points of CPU interaction are very infrequent

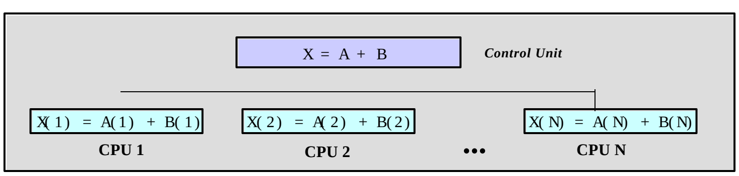
How the Machine Affects Parallelism



"Genealogy" of parallel computing systems

SIMD Computer (Processor Array)

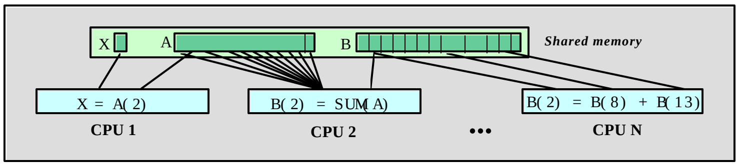
- All processors execute the same instruction in "lockstep"
  - Examples: Maspar, Thinking Machines (CM-2)



- Major programming hurdle:
  - Must use Fortran-90 style array operations efficiently
- Highlights:
  - Efficient use of memory
  - Relatively easy to program
- Lowlights:
  - Programming is difficult or impossible if application isn't fully synchronous
  - All processors perform every operation (even scalar addition or conditional op)

Shared-Memory MIMD Computer

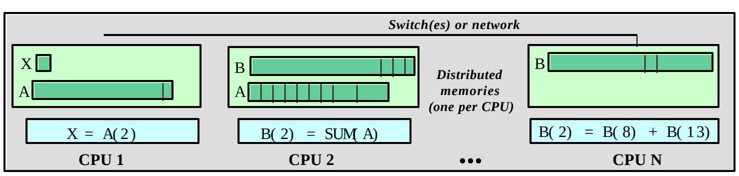
- Each processor executes its own instruction
  - Processors interact by accessing shared memory locations
  - Examples: Cray Y/MP and C-90/J-90, Fujitsu, IBM ES/9000



- Major programming hurdle:
  - Must use compiler directives to protect access to shared data locations
- Highlights:
  - Blindingly fast
  - Large memory
- Lowlights:
  - Very expensive
  - Can be hard or impossible to restructure computational loops effectively

Distributed-Memory MIMD Computer

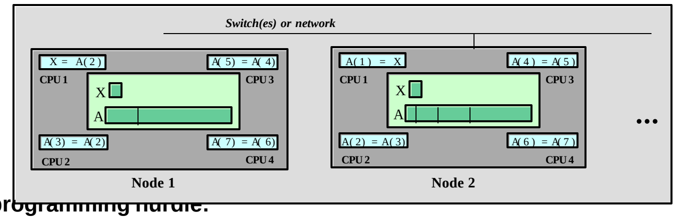
- Each processor executes its own instruction
  - Processors interact via a communication system
  - Examples: IBM SP2, Intel, Meiko, SGI/Cray T3 series



- Major programming hurdle:
  - Must use message-passing (or equivalent) and minimize communications
- Highlights:
  - Versatile
  - Cost-effective
- Lowlights:
  - Hard to use efficiently
  - Can be very hard to debug race conditions and deadlocks

SMP (Symmetric Multiprocessor) Clusters

- Cross between shared- and distributed- memory systems
  - Small group of processors share a common memory (SMP "node")
  - Clustered into larger configurations using a communication system
  - Examples: SGI PowerChallenge & Origin, HP/Convex Exemplar , Sun SPARCServer

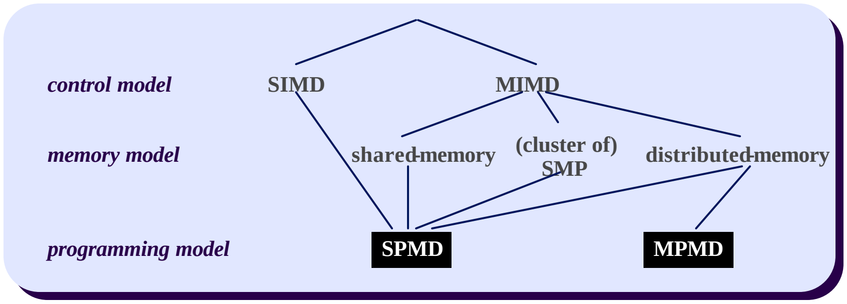


- Major programming hurdle:
  - (Within a node) must protect access to shared data
  - (Off-node) must minimize amount of communication
- Highlights:
  - Versatile
  - Cost-effective
- Lowlights:
  - Hard to use efficiently
  - Problems with race conditions, deadlock

Rules-of-Thumb Based on Type of Machine

- If your application is perfectly parallel
  - → it will probably perform reasonably well on any MIMD architecture, but may be difficult to adapt to a SIMD computer
- If your application is pipeline parallelism
  - → it will probably perform best on a shared-memory machine or clustered SMP (where a given stage fits on a single SMP)
  - → it should be adaptable to a distributed-memory computer as well, as long as the communication network is fast enough to pipe the data sets from one stage to another
- If your application is fully synchronous
  - → it will perform best on a SIMD computer, if you can exploit array operations
  - → it may be respectable on a shared-memory computer (or clustered SMP, if a small number of CPUs is sufficient), but only if the computations are fairly independent
- If your application is loosely synchronous
  - → it will perform best on a shared-memory computer (or clustered SMP, if a small number of CPUs is sufficient)
  - → it may be respectable on a distributed-memory computer, but only if there are many computations between CPU interactions

How Programming Language Affects Parallelism



SPMD model

- (Functionally equivalent to MPMD)
- Each processor executes same object code
- Data storage areas and instructions must be resident on all CPUs
- "Natural" model for SIMD machines
- Convenient for MIMD compiler/tool writers

MPMD model

- Each processor can execute different object code
- Each CPU has only the data/instructions it will need to access
- "Natural" model for MIMD machines (but supported on only a few)
- Convenient for MIMD users

Varieties of Programming Languages

Control-parallel: computational work subdivided across CPUs, which periodically synchronize their activities

- Examples: VP Fortran, Cray Autotasking, *ANSI X3H5 Fortran, OpenMP*
- Model: SPMD on shared-memory computers

Data-parallel: data domain subdivided across CPUs, which provide copies of data they "own" to other CPUs

- Examples: CM-Fortran, C*, MasPar's MPF, *HPF*, *Data Parallel C*
- Model: SIMD (first three), SPMD on distributed-memory computers (last two)

Message-passing: Each CPU executes independently; messages are sent when they need to share data or synchronize activities

- Examples: *PVM*, *MPI*, Intel's NX, p4, Express, Fortran M
- Model: MPMD on distributed-memory computers (first five), SPMD (Fortran M)

Combined: Hybrid of 2 or more (e.g., control-parallel subroutines that send messages to each other)

- Examples: pC++, Convex Fortran, Convex C
- Model: MPMD on distributed-memory computers (pC++), SPMD (last two)

(Italicized examples are *standards*, intended to be supported across multiple vendors)

Rules-of-Thumb Based on Type of Language

Pie-in-the-sky viewpoint:
Any problem can be programmed in any language, for execution on any parallel computer

Realistic viewpoint:
No current machine offers much choice among compilers
Programmers are usually comfortable with 1 or 2 languages
Libraries or associated applications don't interface with just any language

- With few exceptions, you don't really choose a parallel language
  → it chooses you

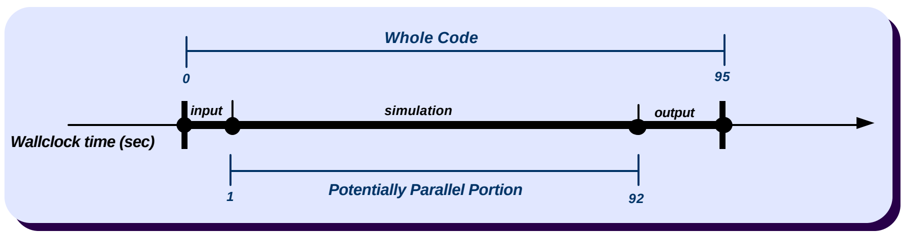
Setting Realistic Expectations

Nobody wants parallelism ... what we want is performance
- *Ken Neves (Boeing)*

- Suppose a serial application has been parallelized to run on 50 CPUs
  - It's using computing resources
    - If results aren't ready much, much faster, resources are being wasted
  - It took somebody a lot of time to parallelize it
    - If performance isn't reasonable, it's a waste of human productivity, too
- How can you estimate whether your efforts will be wasted?
  - Assess your application's <u>potential</u> before committing to parallelism
- Should a parallel program be built from scratch?
  - Computer scientists say "yes"
  - Only 1/3 of parallel programmers report doing so (primarily computer scientists or mathematicians)
- An existing, well-written serial application can facilitate the parallelization process
  - Baseline for checking the validity of parallel program results
  - Baseline for measuring performance improvements
  - ... and some (or most) of the code can be cut-and-pasted into the parallel program

Time the Performance of Your Baseline

- Use your baseline program to estimate its potential parallel performance
  (If it's implemented sloppily, clean it up first!)
- Insert calls to timing routines as the program's first and last statements, to acquire <u>whole-code</u> time
- Insert calls to timing routines just before and after each section with potential for parallelism
  - Collectively, these represent the <u>potentially-parallel</u> code
  - Exclude all potential <u>serial bottlenecks</u>
    - Input or output phases
    - Inherently serial operations (e.g., global summations)



Estimate the Effects of Parallelism

- Goal: reduce the whole-code time so results are produced faster
- Calculate the program's <u>parallel content</u>

$$p = \frac{\text{potentially parallel time}}{\text{whole code time}} = \frac{90}{93} = 0.9677$$

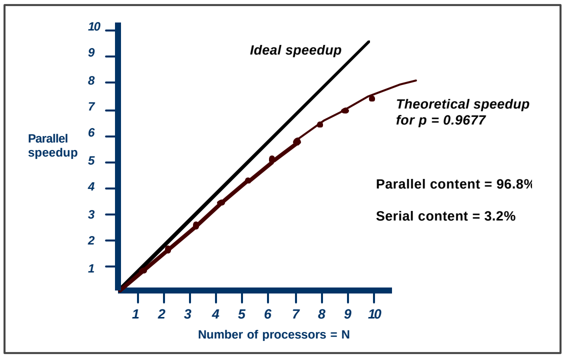
Parallel content indicates what proportion of code can be parallelized
  - 96.77% of the code is potentially parallelizable
  - 3.2% must run serially
- Apply Amdahl's law to calculate <u>theoretical speedup</u>

$$\text{theoretical speedup} = \frac{1}{(1-p) + (p/N)} = \frac{1}{0.323 + (0.9677/N)}$$

as a function of the number (N) of CPUs that will be used
- Ideally, applying N CPUs to a program should cause it to run N times faster
- Theoretical speedup shows the effects of even a small proportion of serial content

Estimate the Effects of Parallelism (cont.)

- Gap between ideal and theoretical speedup widens as N increases
  - Gap is solely a function of the program's serial content
  - For every program and problem size, it is not worthwhile to go beyond some value of N

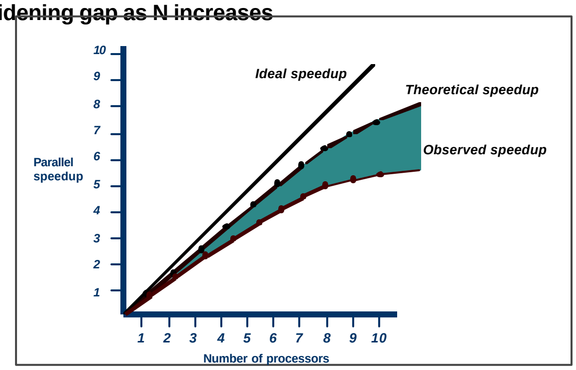


Number of CPUs	Theoretical Speedup
1	1.000
2	1.937
3	2.818
4	3.647
5	4.428
6	5.167
7	5.863
8	6.525
9	7.152
10	7.752
...	...
¥	30.959

- Suppose we greatly increase the size of the problem to be solved
  - How does this affect potential parallel content?
  - Does it change the theoretical speedup curve?

Theory versus Reality in Parallel Execution

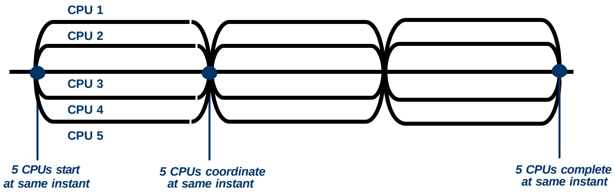
- Observed speedup is even less than theoretical speedup
  - Again, a widening gap as N increases



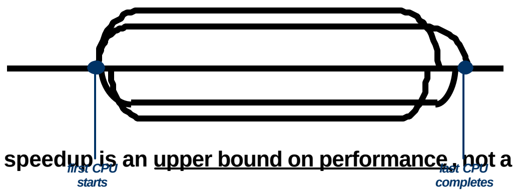
- Theoretical curve (based on Amdahl's Law) does not take into account the overhead of parallelism
  - CPU cycles spent managing parallelism
  - delays or wasted time (waiting for I/O or communications, competition from OS)

Theory versus Reality (cont.)

- Theoretical speedup assumes <u>perfect concurrency</u>
  - All CPUs begin, interact, and complete at the same time



- Real applications are subject to subtle variations in timing



- Theoretical speedup is an <u>upper bound on performance</u>, not a realistic estimate

Estimate the Effects of Program Granularity

- Concurrency worsens as the number of CPU interactions increases
- <u>Granularity</u>: rough measure of how many computations occur between CPU interactions
  - <u>Coarse-grained</u> programs execute many computations between interactions
  - <u>Fine-grained</u> programs interact frequently, with relatively few intervening computations
- For shared-memory computers, it's hard to estimate how coarse-grained the program must be to perform well
- For distributed-memory computers and SMP clusters, it's possible to calculate the <u>message-equivalent</u>
  - Based on machine properties that are generally available
  - Provides an indication of how many computations need to occur be tween CPU interactions

Estimate the Effects of Program Granularity (cont.)

Latency: **Time (in microseconds) required to initiate the transmission of data**

Bandwidth: **Speed (in megabytes/second) at which data are transmitted**

- **Together, they indicate the cost of CPU interactions**
  - **Latency is "fixed overhead" (same cost, regardless of amount of data sent)**
  - **Bandwidth is "variable overhead" (cost is a function of how much data is sent)**
- **Nominal cost of sending a message (or other CPU interactions)**

$$\textit{message time} = \textit{latency} + \frac{\textit{message size}}{\textit{bandwidth}}$$

- **Real cost is the amount of time "lost" as your application waits for a CPU interaction to complete**

Estimate the Effects of Program Granularity (cont.)

- **Message-equivalent indicates how many floating-point operations could be done in the time needed to send one 1,024-byte message**

$$\textit{message equivalent} = \textit{CPU speed} * [\textit{latency} + (1K / \textit{bandwidth})]$$

CPU speed **is the so-called "peak speed" of a single CPU (in MFLOPS)**

	Peak CPU (MFLOPS)	*Latency (microsec)*	*Bandwidth (MB/sec)*	*Message-equivalent*
System A	100	2000	1	300,000
System B	200	300	8	85,000
System C	100	20	50	4,000
System D	150	5	30	5,700
System E	150	25	10	18,750

- **Good performance requires that computation exceed the message-equivalent on a regular basis**
  - **Very coarse-grained programs will succeed anywhere**
  - **None of the example systems would tolerate a medium- or fine-grained program**

Rules-of-Thumb Based on Assessment

- If you have a "clean" serial application
  - timing it will provide you with a solid starting point for estimating potential payoffs
- If the parallel content of your application is less than 95%, you probably shouldn't consider parallelizing it
  - unless you're already experienced in parallel programming, *or*
  - unless you'll be able to dramatically reduce the serial content by substituting a parallel algorithm that has been proven to perform well
- Apply your knowledge of the application to estimate how theoretical speedup will change as problem size grows
  - you will certainly observe less speedup than that (since theoretical speedup is an upper bound on what is possible)
- If your application is coarse-grained
  - it will perform relatively well on any parallel computer
- If your application is fine-grained
  - it will probably won't perform unless you can run it on a SIMD computer
- If you will be using a distributed-memory computer or SMP cluster
  - calculate the message-equivalent to see how many thousands of FLOPs your application needs to perform between each CPU interaction point

Parallel Performance - Fact or Fantasy?

- How much performance can we get?
- Bicycle analogy:
  - I can't ride my bicycle faster than 30 MPH (peak)
  - Speed on an average ride depends on environmental conditions
  - I typically achieve 10 MPH (sustained)
- Parallel computing equivalent:
  - Vendor X claims that the HypoMetaStellar is a 200 GFLOPS machine
  - Shows benchmark results that the HypoMetaStellar is worth 10 Crays
  - What counts is the fraction of peak performance that can be sustained

For real applications, that value is probably only 10% - 20%

Is Parallelism for You?

- Actual performance will depend on 5 critical factors
  (1) inherent parallelism in the application
  (2) multiprocessor architecture
  (3) how well the compiler or parallel library exploits the architecture
  (4) how the program maps the problem to the machine
  (5) scheduling policies on the machine
- An application's parallel content constrains even its theoretical performance
  - If there's more than a tiny fraction of serial content, parallelism almost certainly won't pay off
  - Changing the problem or the algorithm to reduce serial content will have more impact than whatever effort you can put into tuning
- The parallel machine and the runtime environment are probably out of your control
- The efficiency of the language and runtime system are beyond *any* programmer's control
- That leaves the efficiency of your program in mapping your problem to the parallel computer ...

How much effort are you willing to invest?

Is Parallelism for You? (cont.)

- Consider what you hope to gain and how much it will buy you in time or quality
- Consider the propensity your application seems to have for parallelism
- Estimate the best performance you could possibly get through parallelization
- Factor in how well you think your own efforts need to pay off
- (Assuming there are no counter-indications, such as a mismatch between your problem architecture and the type of machine available to you) Make sure the upper-bound estimate on future performance is at least 5-10 times bigger your "bottom-line price"
- *Theoretically,* any problem can be programmed in any language for execution on any parallel computer
- *Realistically* ...

**If a problem doesn't lend itself to parallelism,
or if it doesn't match your computer's capabilities,
parallelization simply won't be worth the effort**