

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

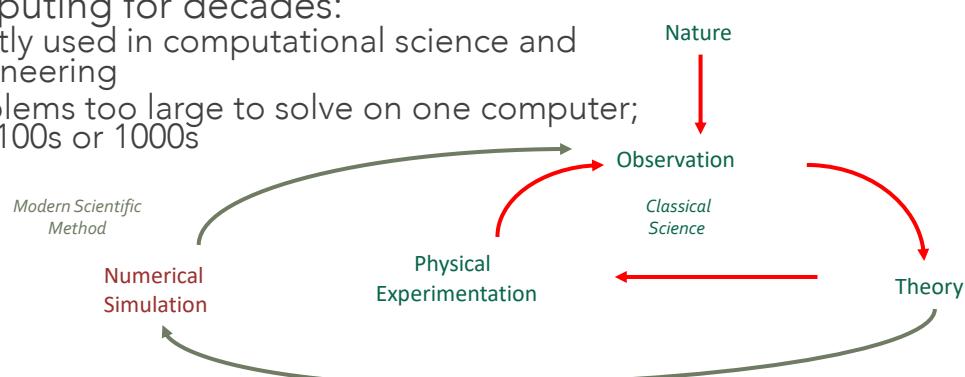
Why Parallel Computing?

Haidar M. Harmanani

Spring 2019

Why Parallel Computing Now?

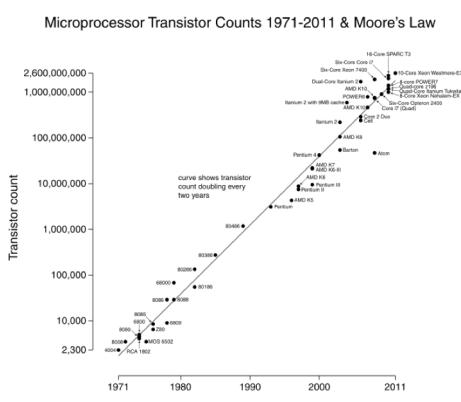
- Researchers have been using parallel computing for decades:
 - Mostly used in computational science and engineering
 - Problems too large to solve on one computer; use 100s or 1000s



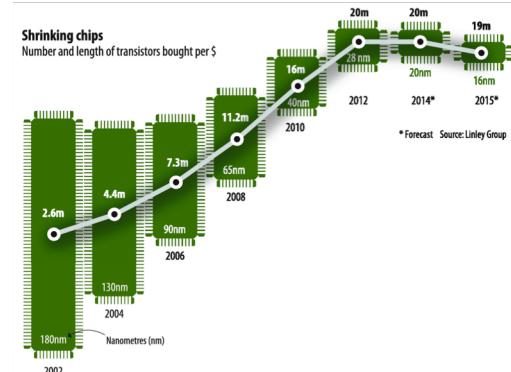
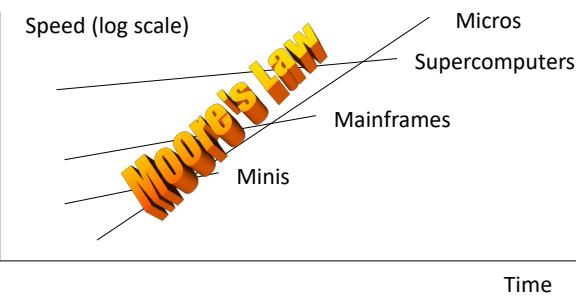
Why Parallel Computing Now?

- Many companies in the 80s/90s “bet” on parallel computing and failed
 - Computers got faster too quickly for there to be a large market
- Why an undergraduate course on parallel programming?
 - Because the entire computing industry has bet on parallelism
 - There is a desperate need for parallel programmers
- There are 3 ways to improve performance:
 - Work Harder
 - Work Smarter
 - Get Help
- Computer Analogy
 - Using faster hardware
 - Optimized algorithms and techniques used to solve computational tasks
 - Multiple computers to solve a particular task

Technology Trends: Microprocessor Capacity

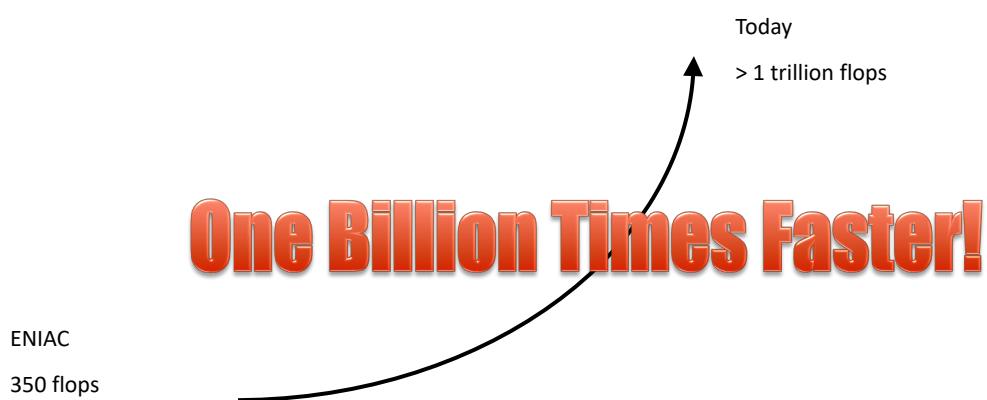


Technology Trends: Microprocessor Capacity



Microprocessors have become smaller, denser, and more powerful.

50 Years of Speed Increases



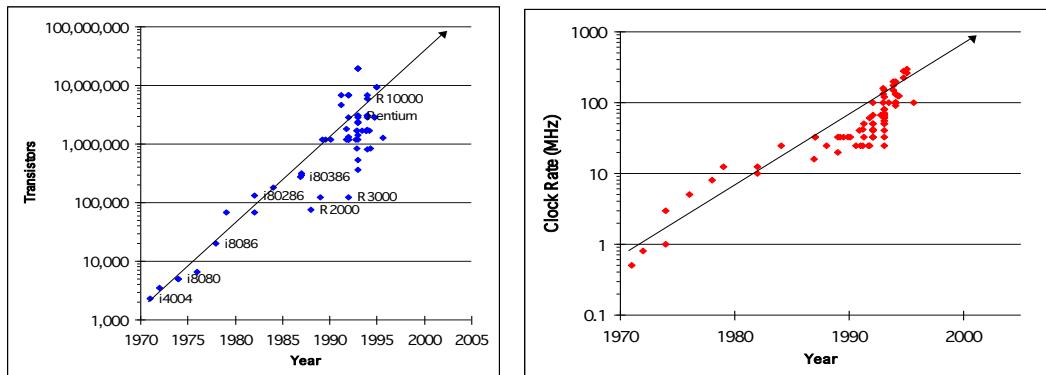
CPUs 1 Million Times Faster

- Faster clock speeds
- Greater system concurrency
 - Multiple functional units
 - Concurrent instruction execution
 - Speculative instruction execution

Systems 1 Billion Times Faster

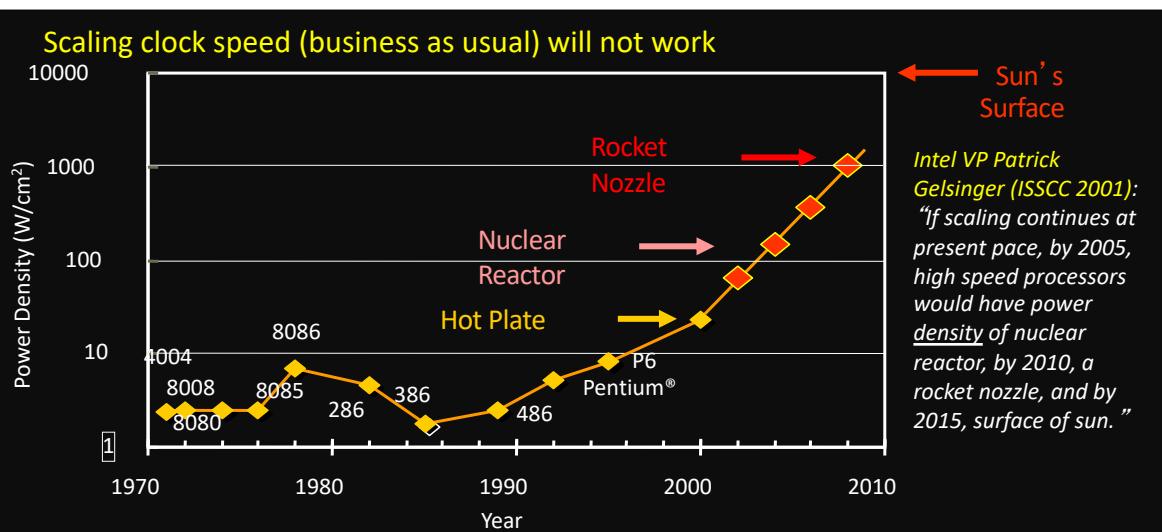
- Processors are 1 million times faster
- Combine thousands of processors
- Parallel computer
 - Multiple processors
 - Supports parallel programming
- Parallel computing = Using a parallel computer to execute a program faster

Microprocessor Transistors and Clock Rate

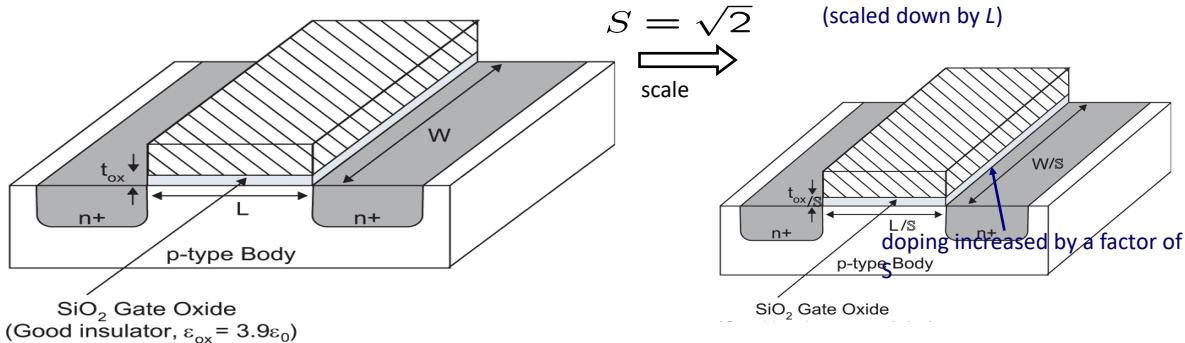


Why bother with parallel programming? Just wait a year or two...

Limit #1: Power density



Device scaling



Increasing the channel doping density decreases the depletion width
 \Rightarrow improves isolation between source and drain during OFF status
 \Rightarrow permits distance between the source and drain regions to be scaled

(very idealistic NMOS transistor)

Parallelism Saves Power

- Exploit explicit parallelism for reducing power

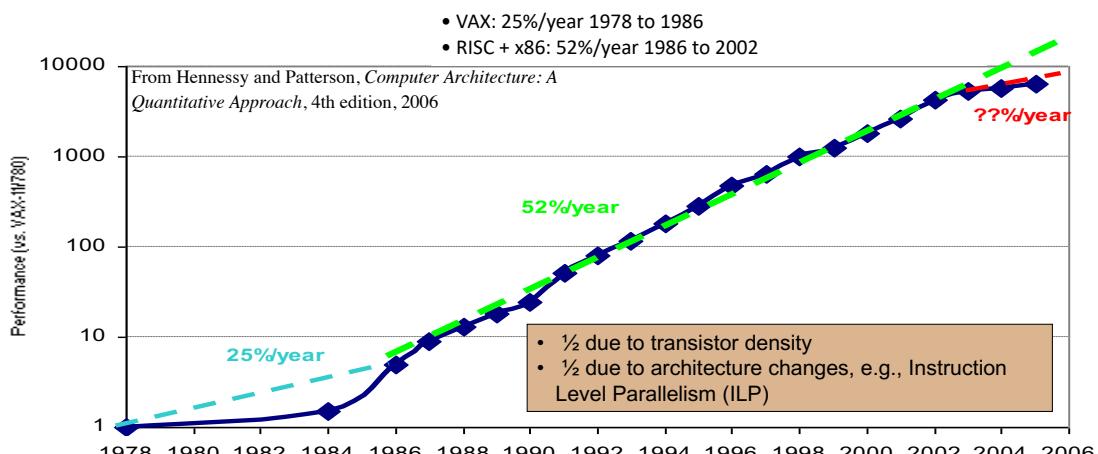
$$\text{Power} = (C * V^2 * F)/4 \quad \text{Performance} = (\text{Cores} * F)*1$$

Capacitance Voltage Frequency

- Using additional cores
 - Increase density (= more transistors = more capacitance)
 - Can increase cores (2x) and performance (2x)
 - Or increase cores (2x), but decrease frequency (1/2): same performance at ¼ the power
- Additional benefits
 - Small/simple cores \rightarrow more predictable performance

Limit #2: Hidden Parallelism Tapped Out

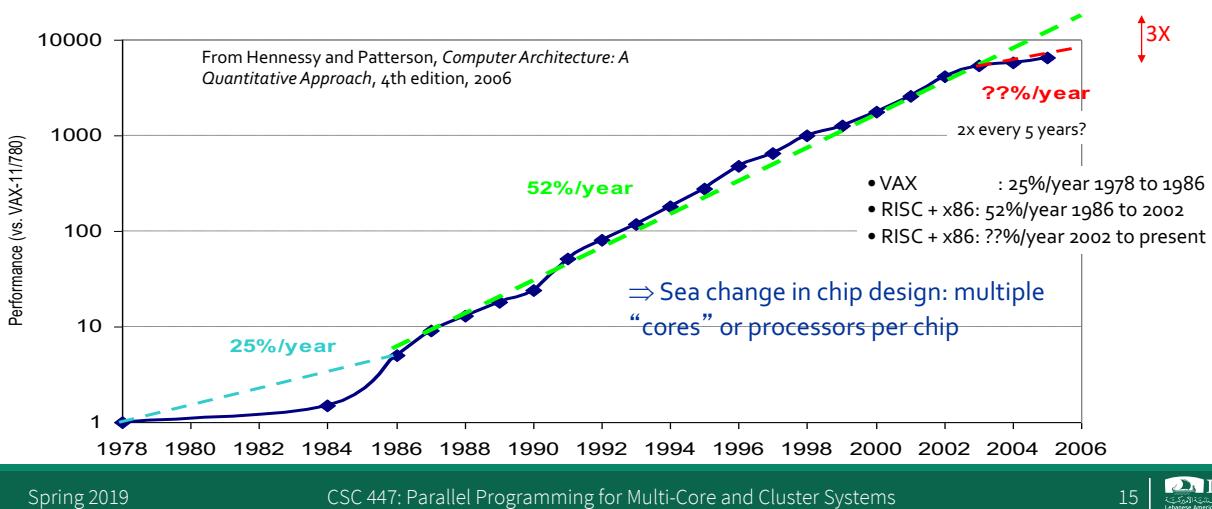
- Application performance was increasing by 52% per year as measured by the SpecInt benchmarks here



Limit #2: Hidden Parallelism Tapped Out

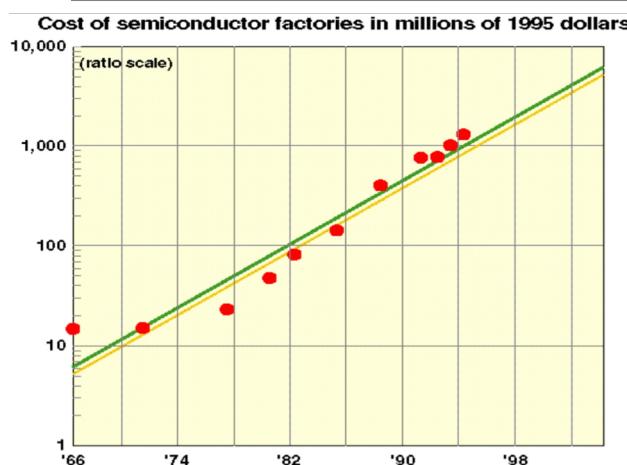
- Superscalar (SS) designs were the state of the art; many forms of parallelism not visible to programmer
 - multiple instruction issue
 - dynamic scheduling: hardware discovers parallelism between instructions
 - speculative execution: look past predicted branches
 - non-blocking caches: multiple outstanding memory ops
- You may have heard of these in CSC320, but you haven't needed to know about them to write software
- Unfortunately, these sources have been used up

Uniprocessor Performance (SPECint) Today



Limit #3: Chip Yield

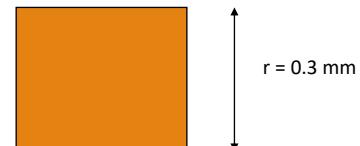
Manufacturing costs and yield problems limit use of density



- Moore's (Rock's) 2nd law: fabrication costs go up
- Yield (% usable chips) drops
- Parallelism can help
 - More smaller, simpler processors are easier to design and validate
 - Can use partially working chips:
 - E.g., Cell processor (PS3) is sold with 7 out of 8 "on" to improve yield

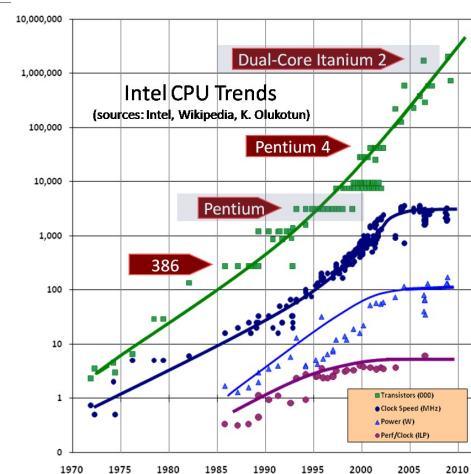
Limit #4: Speed of Light (Fundamental)

- Consider the 1 Tflop/s sequential machine:
 - Data must travel some distance, r , to get from memory to CPU.
 - To get 1 data element per cycle, this means 1012 times per second at the speed of light, $c = 3 \times 10^8$ m/s.
 - Thus $r < c/1012 = 0.3$ mm.
- Now put 1 Tbyte of storage in a 0.3 mm \times 0.3 mm area:
 - Each bit occupies about 1 square Angstrom, or the size of a small atom.
- No choice but parallelism



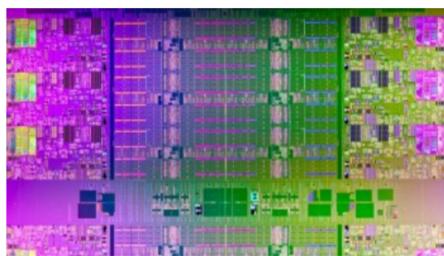
Revolution is Happening Now

- Chip density is continuing increase $\sim 2x$ every 2 years
 - Clock speed is not
 - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software



The death of CPU scaling: From one core to many — and why we're still stuck

By Joel Hruska on February 1, 2012 at 2:31 pm | [86 Comments](#)



Share This Article



consult the following chart. It shows transistor counts, clock speeds, power consumption, and instruction-level parallelism (ILP). The doubling of transistor counts every two years is known as Moore's law, but over time, assumptions about performance and power consumption were also made and shown to advance along similar lines. Moore got all the

It's been nearly eight years since Intel canceled Tejas and announced its plans for a new multi-core architecture. The press wasted little time in declaring conventional CPU scaling dead — and while the media has a tendency to bury products, trends, and occasionally people well before their expiration date, this is one declaration that's stood the test of time.

To understand the magnitude of what happened in 2004 it may help to

Parallel Programming Work Flow

- Identify compute intensive parts of an application
- Adopt scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code portability and maintainability

So what is the problem?

Writing (fast) parallel programs
is hard!

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
 - Granularity
 - Locality
 - Load balance
 - Coordination and synchronization
- All of these things makes parallel programming even harder than sequential programming.

Finding Enough Parallelism (Amdahl's Law)

- Suppose only part of an application seems parallel
- Amdahl's law
 - let s be the fraction of work done sequentially, so $(1-s)$ is fraction parallelizable
 - P = number of processors

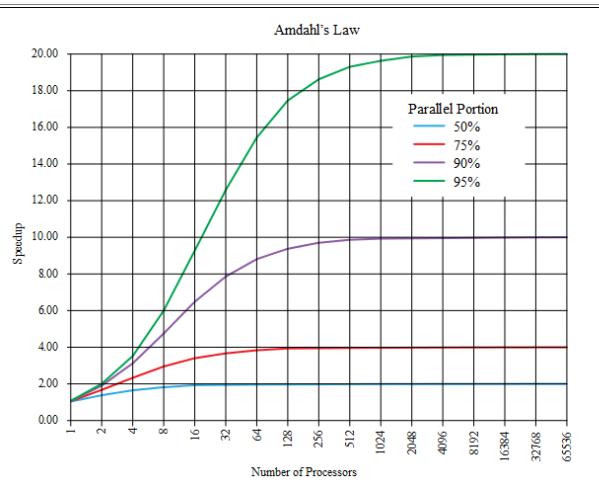
$$\text{Speedup}(P) = \frac{\text{Time}(1)}{\text{Time}(P)}$$

$$<= \frac{1}{s + (1-s)/P}$$

$$<= \frac{1}{s}$$

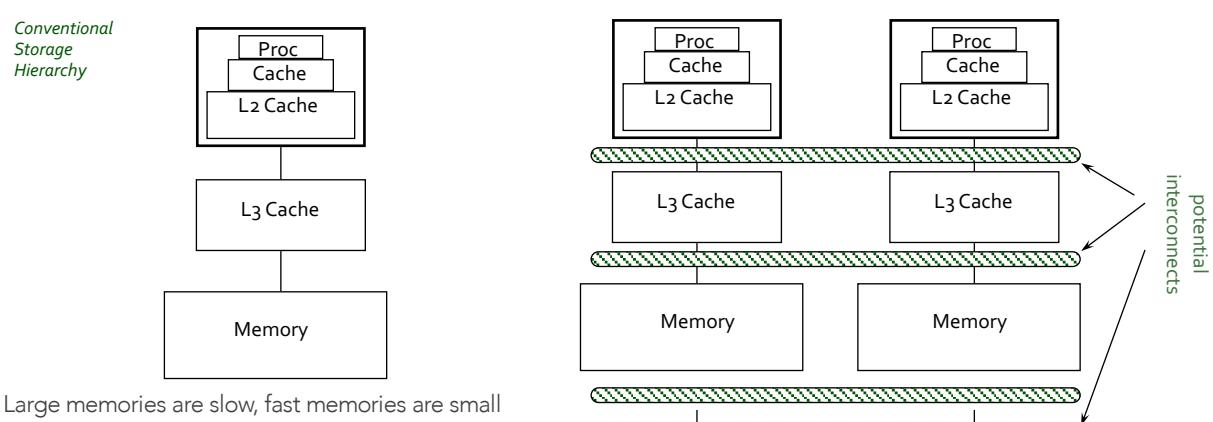
- Even if the parallel part speeds up perfectly performance is limited by the sequential part

Finding Enough Parallelism (Amdahl's Law)



Granularity

- Given enough parallel work, this is the biggest barrier to getting desired speedup
- Parallelism overheads include:
 - Cost of starting a thread or process
 - Cost of communicating shared data
 - Cost of synchronizing
 - Extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work



Large memories are slow, fast memories are small

Storage hierarchies are large and fast on average

Parallel processors, collectively, have large, fast cache

- the slow accesses to "remote" data we call "communication"

Algorithm should do most work on local data

Locality and Parallelism

Load Balance/Imbalance

- Load imbalance is the time that some processors in the system are idle due to
 - insufficient parallelism (during that phase)
 - unequal size tasks
- Examples of the latter
 - adapting to “interesting parts of a domain”
 - tree-structured computations
 - fundamentally unstructured problems
- Algorithm needs to balance load

Load Balance



Synchronization

- Need to manage the sequence of work and the tasks performing
- Often requires "serialization" of segments of the program
- Various types of synchronization maybe involved
 - Locks/Semaphores
 - Barrier
 - Synchronous Communication Operations

Performance Modeling

- Analyzing and tuning parallel program performance is more challenging than for serial programs.
- There is a need for parallel program performance analysis and tuning.

So how do we do parallel computing?

Strategy 1: Extend Compilers

- Focus on making sequential programs parallel
- Parallelizing compiler
 - Detect parallelism in sequential program
 - Produce parallel executable program
- Advantages
 - Can leverage millions of lines of existing serial programs
 - Saves time and labor
 - Requires no retraining of programmers
 - Sequential programming easier than parallel programming
- Disadvantages
 - Parallelism may be irretrievably lost when programs written in sequential languages
 - Performance of parallelizing compilers on broad range of applications still up in air

Strategy 2: Extend Language

- Add functions to a sequential language
 - Create and terminate processes
 - Synchronize processes
 - Allow processes to communicate
- Advantages
 - Easiest, quickest, and least expensive
 - Allows existing compiler technology to be leveraged
 - New libraries can be ready soon after new parallel computers are available
- Disadvantages
 - Lack of compiler support to catch errors
 - Easy to write programs that are difficult to debug

Strategy 3: Add a Parallel Programming Layer

- Lower layer
 - Core of computation
 - Process manipulates its portion of data to produce its portion of result
- Upper layer
 - Creation and synchronization of processes
 - Partitioning of data among processes
- A few research prototypes have been built based on these principles

Strategy 4: Create a Parallel Language

- Develop a parallel language "from scratch"
 - occam is an example
- Add parallel constructs to an existing language
 - Fortran 90
 - High Performance Fortran
 - C*
- Advantages
 - Allows programmer to communicate parallelism to compiler
 - Improves probability that executable will achieve high performance
- Disadvantages
 - Requires development of new compilers
 - New languages may not become standards
 - Programmer resistance