# Introduction to the course

Eric J. Kostelich

Arizona State University
SCHOOL OF MATHEMATICAL & STATISTICAL SCIENCES

Aug. 20, 2018

## Survey and goals of the course

- Programming for scientific computing, especially floating-point intensive code
- Good programming practices, code and data structures
- Fortran 2003/2008, C/C++, POSIX shell
- Models of parallelism: vectorization, OpenMP, MPI, coarrays
- Applications to PDE, linear algebra, and others
- Programs will be run on the Agave cluster

## What the course does not cover

- MapReduce, Hadoop, databases, and the like
- Numerical methods for PDEs, including finite-element and spectral methods (see instead APM 522, 524, 526 and MAE 502, 527, 546, 561)
- Software for genomic analysis (BLAST and similar)
- CUDA and similar frameworks (but if you can vectorize, you can run on a GPU)
- POSIX threads
- Quantum computing

## Synopsis of the syllabus

- Programming projects and homework are 75% of the grade
- Final project is 25% of the grade
- Collaboration with others is permitted provided that you acknowledge your collaborators
- The syllabus contains links to many online resources

## Personal survey

- Your goals for taking the course
- Your previous programming experience in coursework and research
- Your previous professional programming experience (if any)
- What languages you feel most comfortable programming in
- What languages you have had some exposure to
- Your current or proposed thesis work, as applicable

## Consideration #1: Computer memory has levels

- Registers: ($$$$) access time 1 cycle; 16–256 registers in most modern cpu designs
- Cache (in 1 to 3 levels): ($$$) access times from 5 to 40 cycles; typically 32K–512K of level-1 cache to several MB of level-3 cache
- Main memory: ($$) access times $\sim$ 250 cycles; typically 16GB–1028GB per machine in modern designs
- Virtual memory: ($) uses disk; virtually unlimited in size but access times are millions of cycles

## Consideration #2: Amdahl's Law

- Let $p$ be the proportion of the program's cycles that are spent in parallelizable code
- The maximum possible speedup on $n$ cpus is

$$\frac{1}{(1-p)+(p/n)}$$

- Example: if $p = 1/2$, then the maximum speedup is 2, no matter how many processors are used!
- We say that a code scales linearly if the speedup is a multiple of $n$ with $n$ processors

## Models of parallelism

- SIMD: Single Instruction Multiple Data (a.k.a. vectorization)
- SPMD: Single Process Multiple Data (a.k.a. multicore)
- MIMD: Multiple Instruction Multiple Data (very long instruction word, message passing, etc.)
- SMP: Symmetric Multi-Processing (multiple cores, single memory image)
- MPI: Message-Passing Interface (multiple cores, distributed memory)

## Vectorization

- If $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ then $\mathbf{x} + \mathbf{y}$ conceptually is a single operation
- Cray-1, XMP, YMP vector machines: 8 vector registers of $k = 64$ operands each
- Faded in the 1990s and now resurrected in many modern processors
- x86 AVX-512: 32 registers of 512 bits each (up to 8 double-precision operands)
- NEC SX-Aurora: 64 vector registers, each with 256 operands (up to 32 results per cycle)

## Vectorization targets innermost loops

- SAXPY operation:

```
do j=1, n
    y(j) = y(j) + a * x(j)
enddo
```

- Fused multiply-add (FMA) computes $ax_j + y_j$ in one step with one rounding
- The compiler issues $n/4$ FMA instructions if x and y are single precision and 128-bit vector registers are available

## General requirements

- The loop count must be fixed at the start
- The order of operations must not matter
- x and y must not overlap in memory
- Only simple conditionals allowed

```
do j=1, n
    y(j) = max(y(j), x(j))
enddo
```

## Vectorization provides a limited paradigm

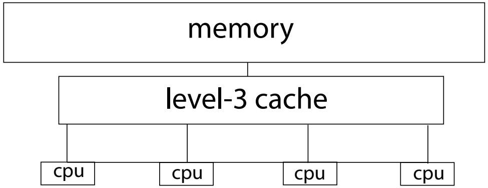
- Useful for “loopy” code that performs the same set of operations on large blocks of data
- Example: SAXPY operations: `y(i) = y(i) + a * x(i)`
- Not useful for:
  - Overlapping computation with i/o
  - Using multiple cores on compute-bound codes
  - Keeping a GUI responsive by spawning a separate task for a compute-intensive requests

## Multicore computers are now common

- Increasing the clock speed on a chip raises energy consumption and heat dissipation
- Instead, chip designers put more than one compute core on a single microprocessor
- While one core is busy, the others can do other tasks
- Can yield good performance at lower clock speeds while using less energy
- Individual cores can be powered up or down as needed

## The multicore programming paradigm

- Divides computations among several cpus, each of which has a common view of memory
- Ranges from 2-core laptop to the NASA Ames Columbia Supercomputer (SGI Altix with 10,240 Itanium processors running as a single system image)
- Smartphones use 2–8 cores for various functions

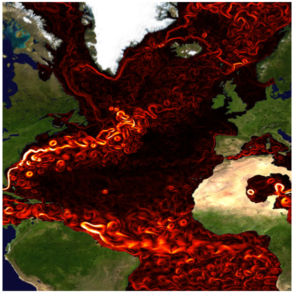


## Multicore is a limited paradigm

- Advantage: Relatively easy to program
- Disadvantage: Memory bandwidth is limited
- Consequently, multicore performance does not scale to more than a few processors

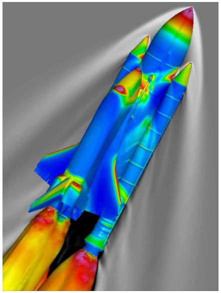
## Examples at NASA Ames

- The Parallel Ocean Program (part of the Community Climate System Model)
- The NASA Finite-Volume General Circulation Model (fvGCM)



## The Overflow code at NASA Ames

- Space Shuttle launch vehicle at Mach 2.46 at 66,000 feet
- Colors show pressure on the vehicle's surface
- Gray contours show density of the surrounding air



## Unix processes

- Starting a program in a Unix/Linux environment involves many steps
- Dynamic linker gets code from shared libraries to resolve undefined references
- The kernel allocates a process id, an address space, and a scheduling slot
- Loader and kernel allocate a stack, a heap, and virtual memory and initializes the program counter
- The ps command-line utility shows the status of some or all running processes

## Lightweight processes (“threads”)

- Starts a partial copy of the main program
- Shares the same process id and code space of the parent
- Non-stack (e.g., static and global) variables are shared among threads
- Each thread maintains its own stack and runs autonomously
- When threads exit, their stack is reclaimed but the main program continues

## Example usage

- If i/o and computation take about the same amount of time, this code can run in half the usual wall-clock time, even on a single cpu:

```
read first file
do while(more files remain to be processed)
        spawn thread to compute on current file
        spawn thread to read next file
        wait
        spawn thread to write results
        current file = next file
        increment next file
enddo
```

## Limitations and complications

- Amdahl's Law: if $p$ is the fraction of the code that is parallelizable, then given $n$ cores, the maximum speedup is

$$\frac{1}{(1-p)+(p/n)}$$

- Serial portions of the program become a bottleneck
- CPU utilization is limited by main memory bandwidth once cache is full
- Main memory is much slower than the CPU!

## Access to shared variables is complicated

- If `x` is a global variable, then any thread can alter its value
- Since the time scheduling of threads is indeterminate, so is the final value of `x` unless precautions are taken
- Locks and mutexes are code and data constructs to ensure that only one thread has access to `x` at any given time
- Code in locked regions is effectively serialized
- Examples: Heap managers, i/o with common files

## The C++ STL is a bottleneck

- The C++ Standard Template Library (STL) is poorly suited for multithreaded and multicore parallelism
- vectors with the default constructor require serialized heap access
- The iterator construct

  ```
  for(p = vec.begin(); p != vec.end(); ++p)
  ```

  does not lend itself to vectorization
- Fortran permits variable-sized stack-based arrays and vector constructs
- However: thread stack space is limited (typically 2–64 MB)

## Algorithm considerations

- Sorting, searching, and summation are tasks for which the optimum choice of algorithm depends on data size and number of processors
- Serial programs that maintain a lot of “state” may require significant redesign to execute efficiently in a parallel environment (e.g., a weather forecast model that updates a global atmosphere)
- Incompatible template libraries reduce code portability and reuse
- Standard C++ has 3 incompatible ways to declare an array: `array`, `vector`, and `valarray` (also `tuple` and `pair`)

## The POSIX threading library

- The “pthreads” library provides routines to spawn and manage threads
- Programmer has complete control but also has complete responsibility for managing the threads and associated variables (no compiler support)
- Thread-safe code permits concurrent execution by multiple cpus
- Requires stack-based variables and a minimum of serialized accesses to be efficient
- Other program libraries must be thread safe for efficiency

## Some other libraries and languages

- ScaLAPACK provides parallelized linear algebra routines
- Intel Threaded Building Blocks is a proprietary C++ library for many tasks, including image, video, and audio processing, cryptography, and signal processing
- Some experimental languages: X10 (IBM), Chapel (Cray), Fortress (Sun/Oracle), Julia (MIT/open source), Unified Parallel C (Berkeley)
- Recommended reference: Clay Breshears, *The Art of Concurrency*, O'Reilly