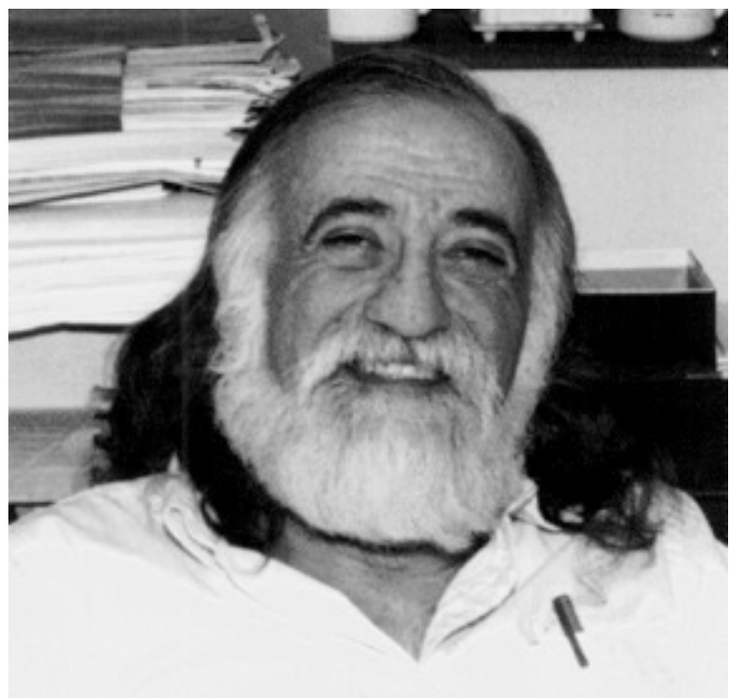
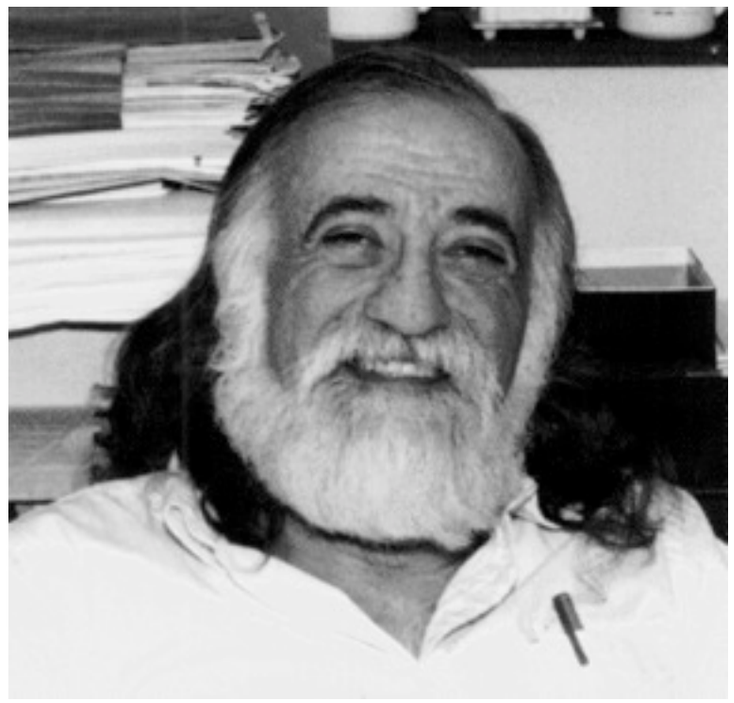
# Bottom-Up Multicore Programming

# Bottom-Up Approach Inspiration



# Bottom-Up Approach Inspiration

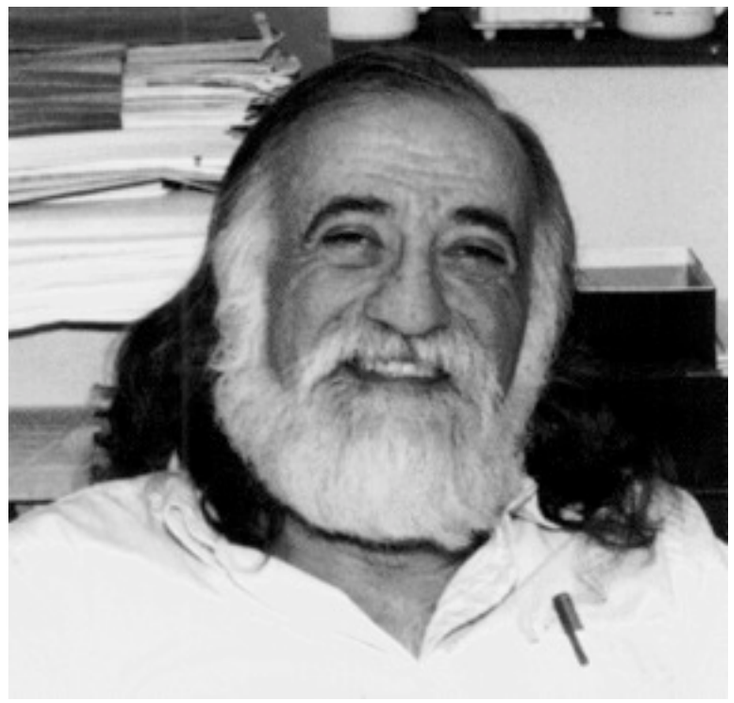
- Robert Lee Moore
  - Mathematician
    - Penn (1911-1920)
    - U. of Texas (1920-1969)
  - Developed "Moore Method" or "Texas Method"
    - A unique style of teaching mathematics
    - Start with axioms, have class build up the framework during the semester
    - "Looking ahead" forbidden



# Bottom-Up Approach Inspiration

- Robert Lee Moore
  - Mathematician
    - Penn (1911-1920)
    - U. of Texas (1920-1969)
  - Developed "Moore Method" or "Texas Method"
    - A unique style of teaching mathematics
    - Start with axioms, have class build up the framework during the semester
    - "Looking ahead" forbidden

- Yale Patt
  - Computer Architect
  - Renowned teacher and researcher
  - "Bits and Bytes to C and Beyond" textbook



# Bottom-up Multicore Programming

# Bottom-up Multicore Programming

- We’re going to follow a “bottom-up” approach
  - We’re going to start with the lowest-level primitives
  - Build up high abstractions using these primitives

# Bottom-up Multicore Programming

- We're going to follow a "bottom-up" approach
  - We're going to start with the lowest-level primitives
  - Build up high abstractions using these primitives
- Why?
  - No-magic understanding
  - Gain performance intuition

# Bottom-up Multicore Programming

- We're going to follow a "bottom-up" approach
  - We're going to start with the lowest-level primitives
  - Build up high abstractions using these primitives
- Why?
  - No-magic understanding
  - Gain performance intuition

- Discarded alternative #1: "top-down"
  - Start with high-level parallelism libraries and languages
    - As "black boxes"
  - Then try to explain how they work
    - And their odd performance characteristics

# Bottom-up Multicore Programming

- We're going to follow a "bottom-up" approach
  - We're going to start with the lowest-level primitives
  - Build up high abstractions using these primitives
- Why?
  - No-magic understanding
  - Gain performance intuition
- Discarded alternative #1: "top-down"
  - Start with high-level parallelism libraries and languages
    - As "black boxes"
  - Then try to explain how they work
    - And their odd performance characteristics
- Discard alternative #2: "top-up"

# What Bottom-Up Means for You

# What Bottom-Up Means for You

- Need to "forget" (for now) some things
  - Put aside any thoughts of higher-level constructs
  - Forget anything you might know of
    - **TBB**, **Cilk**, **OpenMP**, Blocks, Grand Central Dispatch, Ct, Java JSR-166, ZPL, NESL, Fortress, X10, Chapel, StreamIT, Brook, CUDA, **OpenCL**, **memory consistency**, etc.

# What Bottom-Up Means for You

- Need to "forget" (for now) some things
  - Put aside any thoughts of higher-level constructs
  - Forget anything you might know of
    - **TBB**, **Cilk**, **OpenMP**, Blocks, Grand Central Dispatch, Ct, Java JSR-166, ZPL, NESL, Fortress, X10, Chapel, StreamIT, Brook, CUDA, **OpenCL**, **memory consistency**, etc.
- Need to actively engage with material
  - Engage in class discussion, in-class exercises
  - Spend time outside of class on material
  - Embrace the homework assignments
    - More gritty, more effort at the start

# What Bottom-Up Means for You

- Need to "forget" (for now) some things
  - Put aside any thoughts of higher-level constructs
  - Forget anything you might know of
    - **TBB**, **Cilk**, **OpenMP**, Blocks, Grand Central Dispatch, Ct, Java JSR-166, ZPL, NESL, Fortress, X10, Chapel, StreamIT, Brook, CUDA, **OpenCL**, **memory consistency**, etc.
- Need to actively engage with material
  - Engage in class discussion, in-class exercises
  - Spend time outside of class on material
  - Embrace the homework assignments
    - More gritty, more effort at the start
- Less spoon-feeding, more self-learning

# What Bottom-Up Means for You

- Need to "forget" (for now) some things
  - Put aside any thoughts of higher-level constructs
  - Forget anything you might know of
    - **TBB**, **Cilk**, **OpenMP**, Blocks, Grand Central Dispatch, Ct, Java JSR-166, ZPL, NESL, Fortress, X10, Chapel, StreamIT, Brook, CUDA, **OpenCL**, **memory consistency**, etc.
- Need to actively engage with material
  - Engage in class discussion, in-class exercises
  - Spend time outside of class on material
  - Embrace the homework assignments
    - More gritty, more effort at the start
- Less spoon-feeding, more self-learning
- Best way I know how to teach this course

# Simple Parallel Work Decomposition

# Static Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - void each_thread(int thread_id):
    - segment_size = SIZE / number_of_threads
    - assert(SIZE % number_of_threads == 0)
    - my_start = thread_id * segment_size
    - my_end = my_start + segment_size
    - for (int i=**my_start**; i<**my_end**; i++)
      - calculate(i, …, …, …)
- Hey, its a parallel program!

# Static Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - void each_thread(int thread_id):
    - segment_size = SIZE / number_of_threads
    - assert(SIZE % number_of_threads == 0)
    - my_start = thread_id * segment_size
    - my_end = my_start + segment_size
    - for (int i=**my_start**; i<**my_end**; i++)
      - calculate(i, …, …, …)
- Hey, its a parallel program!

# Static Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - void each_thread(int thread_id):
    - segment_size = SIZE / number_of_threads
    - assert(SIZE % number_of_threads == 0)
    - my_start = thread_id * segment_size
    - my_end = my_start + segment_size
    - for (int i=**my_start**; i<**my_end**; i++)
      - calculate(i, …, …, …)

# Static Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - void each_thread(int thread_id):
    - segment_size = SIZE / number_of_threads
    - assert(SIZE % number_of_threads == 0)
    - my_start = thread_id * segment_size
    - my_end = my_start + segment_size
    - for (int i=**my_start**; i<**my_end**; i++)
      - calculate(i, …, …, …)
- Hey, its a parallel program!

# Dynamic Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - int counter = 0  **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= SIZE)
        - return
      - calculate(i, …, …, …)
- Dynamic load balancing, but high overhead

# Dynamic Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - int counter = 0 **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= SIZE)
        - return
      - calculate(i, …, …, …)
- Dynamic load balancing, but high overhead

# Dynamic Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - int counter = 0  **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= SIZE)
        - return
      - calculate(i, …, …, …)

# Dynamic Work Distribution

- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)
- Parallel code, for each thread:
  - int counter = 0 **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= SIZE)
        - return
      - calculate(i, …, …, …)
- Dynamic load balancing, but high overhead

# Coarse-Grain Dynamic Work Distribution

# Coarse-Grain Dynamic Work Distribution

# Coarse-Grain Dynamic Work Distribution

- Parallel code, for each thread:
  - int num_segments = SIZE / GRAIN_SIZE
  - int counter = 0 **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= num_segments)
        - return
      - my_start = i * GRAIN_SIZE
      - my_end = my_start + GRAIN_SIZE
      - for (int j=**my_start**; j<**my_end**; j++)
        - calculate(j, …, …, …)

# Coarse-Grain Dynamic Work Distribution

- Parallel code, for each thread:
  - int num_segments = SIZE / GRAIN_SIZE
  - int counter = 0 **// global variable**
  - void each_thread(int thread_id):
    - while (true)
      - int i = **atomic_add**(&counter, 1)
      - if (i >= num_segments)
        - return
      - my_start = i * GRAIN_SIZE
      - my_end = my_start + GRAIN_SIZE
      - for (int j=**my_start**; j<**my_end**; j++)
        - calculate(j, …, …, …)
- Dynamic load balancing with lower (adjustable) overhead