Parallelism Primitives

Shared Memory Primitives

- **Create thread**
  - Ask operating system to create a new “thread”
  - Threads starts at specified function (via function pointer)
- Memory regions
  - Shared: globals and heap
  - Per-thread: stack
- Primitive memory operations
  - Loads & stores
  - Word-sized operations are atomic (if “aligned”)
- **Various “atomic” memory instructions**
  - Swap, compare-and-swap, test-and-set, atomic add, etc.
  - Perform a load/store pair that is guaranteed not to interrupted

Thread Creation

- Varies from operating system to operating system
  - POSIX threads (P-threads) is a widely supported standard (C/ C++)
  - Lots of other stuff in P-threads we're **not** using
    - Why? really design for single-core OS-style concurrency

- pthread_create(id, attr, start_func, arg)
  - "id" is pointer to a "pthread_t" object
  - We're going to ignore "attr"
  - "start_func" is a function pointer
  - "arg" is a void *, can pass pointer to anything (ah, C…)

Compare and Swap (CAS)

- Atomic Compare and Swap (CAS)
  - Basic and universal atomic operations
  - Can be used to create all the other variants of atomic operations
  - Supported as instruction on both x86 and SPARC

- Compare_and_swap(address, test_value, new_value):
  - Load [Address] -> old_value
  - if (old_value == test_value):
    - Store [Address] <- new_value
  - Return old_value

- Can be included in C/C++ code using “inline assembly”
  - Becomes a utility function

Inline Assembly for Atomic Operations

- x86 inline assembly for CAS
  - From Intel's TBB source code `machine/linux_intel64.h`

```
static inline int64 compare_and_swap(volatile void *ptr,
                                     int64 test_value,
                                     int64 new_value)
{
  int64 result;
  __asm__
  __volatile__("lock\ncmpxchgq %2,%1"
               : "=a"(result), "=m"(*(int64 *)ptr)
               : "q"(new_value), "0"(test_value), "m"(*(int64 *)ptr)
               : "memory");
  return result;
}
```

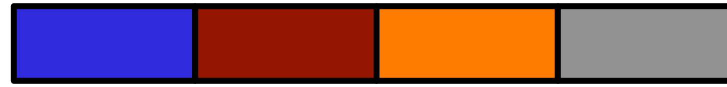
- Black magic
  - Use of **`volatile`** keyword disable compiler memory optimizations

Thread Local Storage (TLS)

- Sometimes having a non-shared global variable is useful
  - A per-thread copy of a variable
- Manual approach:
  - Definition: `int accumulator[NUM_THREADS];`
  - Use: `accumulator[thread_id]++;`
  - Limited to `NUM_THREADS`, need to pass `thread_id` around false sharing
- Compiler supported:
  - Definition: `__thread int accumulator = 0;`
  - Use: `accumulator++;`
  - Implemented as a per-thread "globals" space
    - Accessed efficiently via `%gs` segment register on x86-64
  - More info: http://people.redhat.com/drepper/tls.pdf

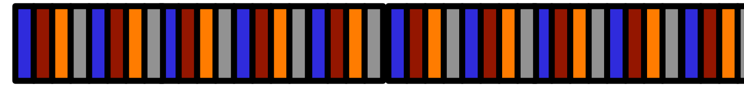
Simple Parallel Work Decomposition

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!

Dynamic Work Distribution

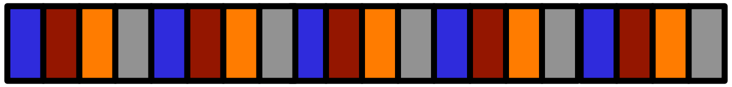


- Sequential code
  - for (int i=**0**; i<**SIZE**; i++):
    - calculate(i, …, …, …)

- 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

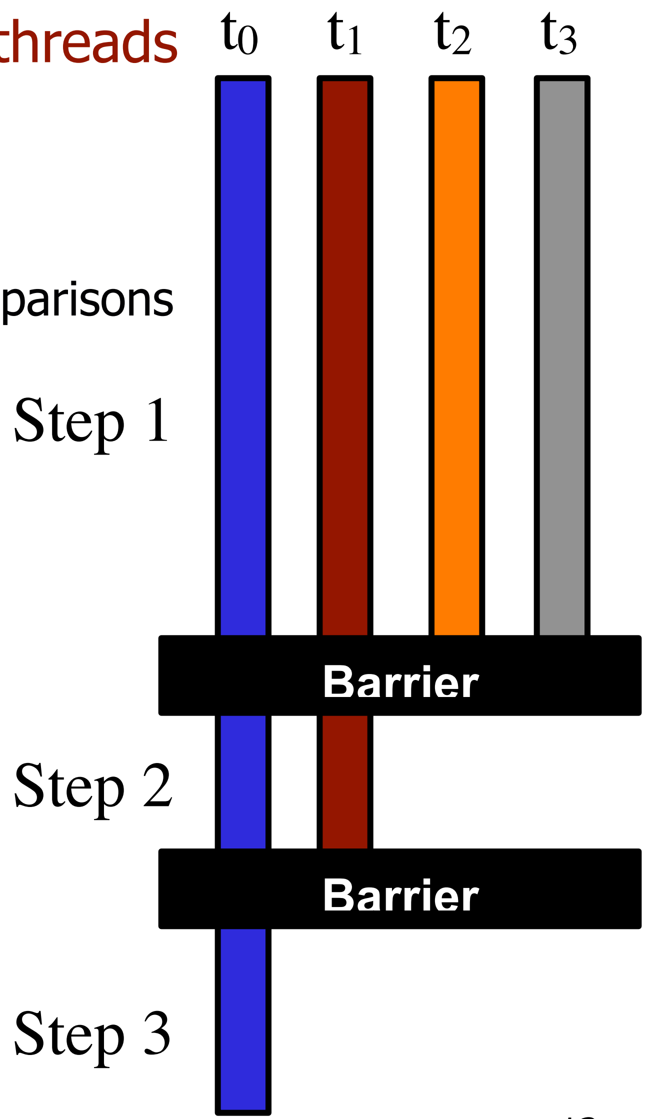
Coarse-Grain Dynamic Work Distribution

- 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

Barriers

Example: Barrier-Based Merge Sort

- Merge-sort 4096 elements with four threads
- Step #1:
  - Sort each 1/4th of array
  - (N/4)*log(N/4) = 1024*10 = 10240 comparisons
- Step #2:
  - Two N/2 merges
  - 2048 comparisons
- Step #3:
  - Final merge
  - 4096 comparisons
- Total: 3x speed up four threads
  - Parallel: 16384 comparisons
  - Sequential: ~50k comparisons

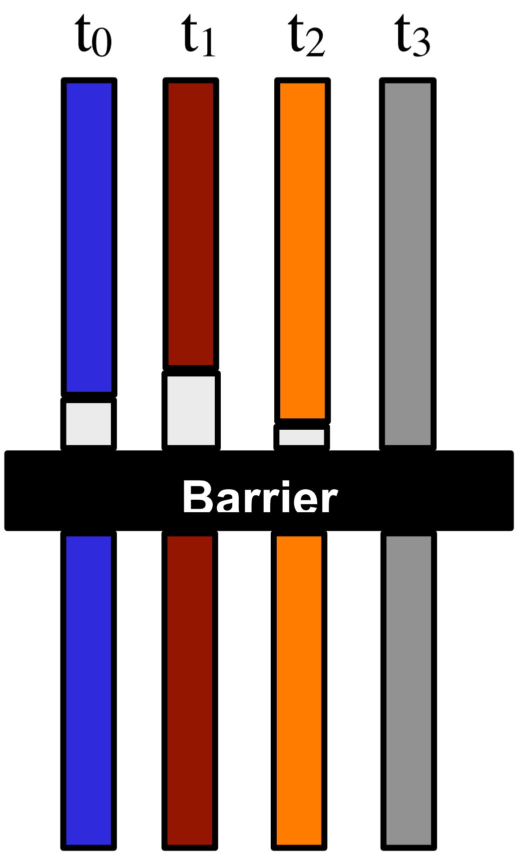


Global Synchronization Barrier

- At a barrier
  - All threads wait until all other threads have reached it
- Strawman implementation (**wrong!**)

```
global (shared) count : integer := P

procedure central_barrier
  if fetch_and_decrement(&count) == 1
    count := P
  else
    repeat until count == P
```



- What is wrong with the above code?

Sense-Reversing Barrier

- Correct barrier implementation:

```
global (shared) count : integer := P
global (shared) sense : Boolean := true
local (private) local_sense : Boolean := true

procedure central_barrier
  // each processor toggles its own sense
  local_sense := !local_sense
  if fetch_and_decrement(&count) == 1
    count := P
    // last processor toggles global sense
    sense := local_sense
  else
    repeat until sense == local_sense
```

- Single counter makes this a “centralized” barrier

Other Barrier Implementations

- Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - O(n) if threads arrive simultaneously, slow for lots of processors
- Combining Tree Barrier
  - Build a $\log_k(n)$ height tree of counters (one per cache block)
  - Each thread coordinates with **k** other threads (by thread id)
  - Last of the **k** processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - O(log n) in best case
- Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release

Barrier Performance (from 1991)

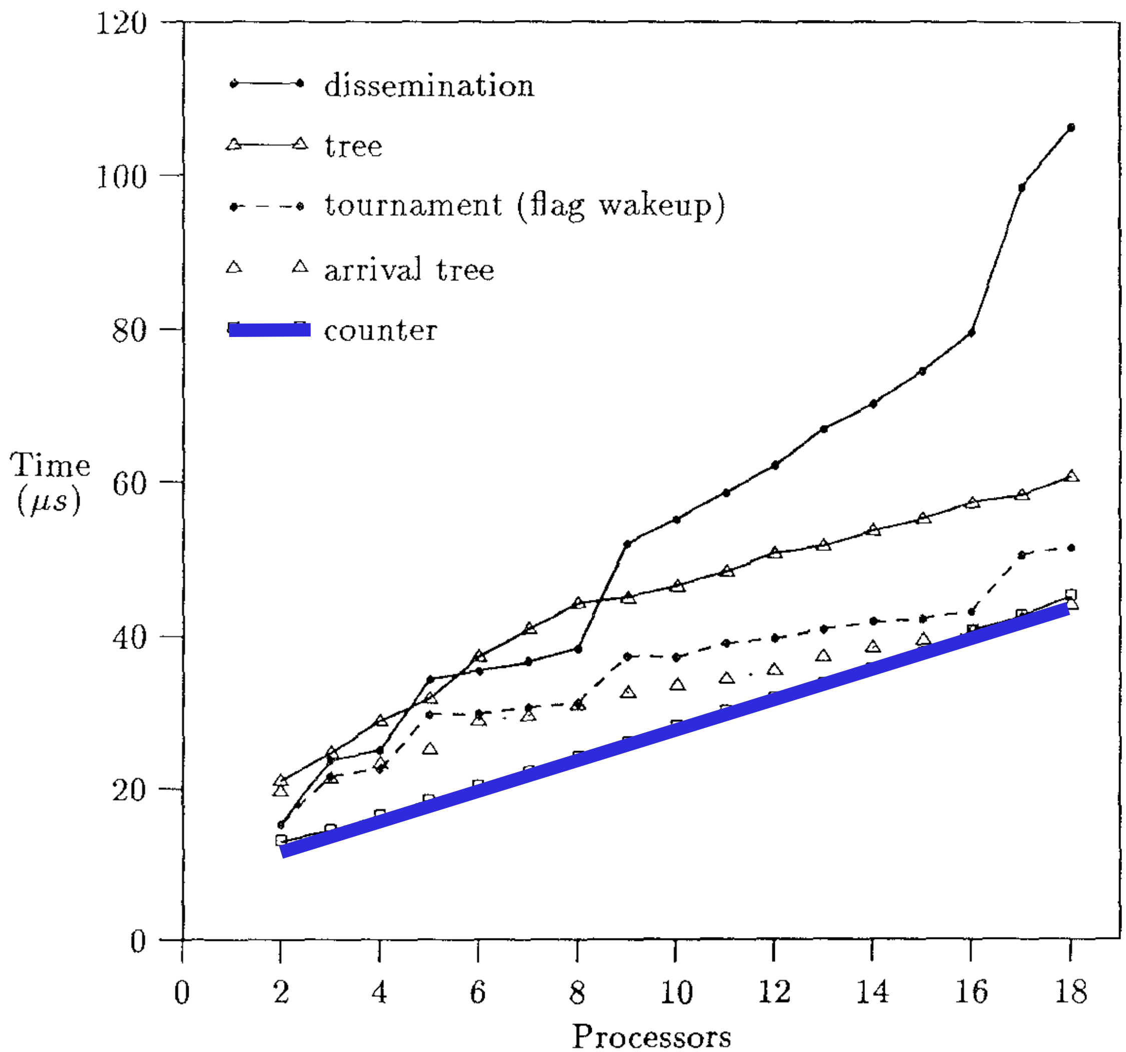


Fig. 21. Performance of barriers on the Symmetry

From Mellor-Crummey & Scott, ACM TOCS, 1991

Locks