Lecture 15
Parallel Storage Allocation

Charles E. Leiserson
October 30, 2012
BASIC ALLOCATORS AND THEIR PROPERTIES
1. Allocator Speed

**Definition.** Allocator *speed* is the number of allocations and deallocations per second that the allocator can sustain.

**Q.** Is it more important to maximize allocator speed for large blocks or small blocks?

**A.** Small blocks!

**Q.** Why?

**A.** Typically, a user program writes all the bytes of an allocated block. A large block takes so much time to write that the allocator time has little effect on the overall runtime. In contrast, if a program allocates many small blocks, the allocator time can represent a significant overhead.
**Fragmentation**

**Definition.** The user footprint is the maximum over time of the number $U$ of bytes in use by the user program (allocated but not freed). The allocator footprint is the maximum over time of the number $A$ of bytes of virtual memory provided to the allocator by the operating system. ($A$ typically grows monotonically.) The fragmentation is $F = A/U$.

**Theorem** (proved previously). The fragmentation for binned free lists is $O(\lg U)$. ■
Fragmentation Taxonomy

- **Space overhead**: space used by the allocator for bookkeeping.
- **Internal fragmentation**: waste due to allocating larger blocks than the user requests.
- **External fragmentation**: waste due to the inability to use storage because it is not contiguous.
- **Blowup**: for a parallel allocator, the additional waste beyond what a serial allocator would require.
Scalability

As the number of threads (processors) grows, the time to perform an allocation or deallocation should not increase.

- The most common reason for loss of scalability is lock contention.

Q. Is lock contention more of a problem for large blocks or for small blocks?

A. Small blocks!

Q. Why?

A. Typically, a user program writes all the bytes of an allocated block, making it hard for a thread allocating large blocks to issue allocation requests at a high rate. In contrast, if a program allocates many small blocks in parallel, contention can be a significant issue.
Global Heap

- All threads (processors) share a single heap.
- Accesses are mediated by a mutex (or lock-free synchronization).

😊 Blowup $= 1$.
😊 Slow — acquiring a lock is like an L2-cache access.
😊 Contention inhibits scalability.
Local Heaps

- Each thread allocates out of its own heap.
- No locking is necessary.

😊 Fast — no synchronization.
😊 Suffers from memory drift: blocks allocated by one thread are freed on another ⇒ unbounded blowup.
Local Heaps with Ownership

- Each object is labeled with its owner.
- Freed objects are returned to the owner’s heap.
- Fast allocation and freeing of local objects.
- Freeing remote objects requires synchronization.
- Blowup $\leq P$.
- Resilience to false sharing.
False Sharing

The compiler happens to place $x$ and $y$ in the same cache block.

Write $x$
False Sharing

The compiler happens to place $x$ and $y$ in the same cache block.
False Sharing

The compiler happens to place \( x \) and \( y \) in the same cache block.

Write \( x \)
False Sharing

The compiler happens to place x and y in the same cache block.

© 2012 Charles E. Leiserson and I-Ting Angelina Lee
How False Sharing Can Occur

A program can induce false sharing by passing an object it owns to another thread.
- The programmer can mitigate this problem by aligning the object on a cache-line boundary and padding out the object to the size of a cache line, but this solution can be wasteful of space.

An allocator can induce false sharing in two ways:
- Actively, when the allocator satisfies memory requests from different threads using the same cache block.
- Passively, when the program passes objects lying on the same cache line to different threads, and the allocator reuses the objects’ storage after the objects are freed to satisfy requests from those threads.
The Hoard Allocator

- $P$ local heaps.
- 1 global heap.
- Memory is organized into large superblocks of size $S$.
- Only superblocks are moved between the local heaps and the global heap.

😊 Fast.
😊 Scalable.
😊 Bounded blowup.
😊 Resilience to false sharing.
Assume without loss of generality that all blocks are the same size (fixed-size allocation).

```c
x = malloc() on thread i
if (there exists a free object in heap i) {
    x = an object from the fullest nonfull superblock in i’s heap;
} else {
    if (the global heap is empty) {
        B = a new superblock from the OS;
    } else {
        B = a superblock in the global heap;
    }
    set the owner of B to i;
    x = a free object in B;
} return x;
```
Hoard Deallocation

Let $u_i$ be the in-use storage in heap $i$, and let $a_i$ be the storage owned by heap $i$. Hoard maintains the following invariant for all heaps $i$:

$$u_i \geq \min(a_i - 2S, a_i/2),$$

where $S$ is the superblock size.

`free(x)`, where $x$ is owned by thread $i$:

```c
put x back in heap i;
if (u_i < min(a_i - 2S, a_i/2)) {
    move a superblock that is at least \(1/2\) empty from heap $i$ to the global heap;
}
```
Hoard’s Blowup

**Theorem.** Let $U$ be the user footprint for a program, and let $A$ be Hoard’s allocator footprint. We have

$$A \leq U + 2SP,$$

and hence the blowup is

$$A/U = 1 + O(SP/U).$$
CACTUS STACKS
An execution of a serial C/C++ program can be viewed as a **serial walk** of an **invocation tree**.
Rule for pointers: A parent can pass pointers to its stack variables down to its children, but not the other way around.
A cactus stack supports multiple views in parallel.
A heap–based cactus stack allocates frames off the heap.
**Theorem.** Let $S_1$ be the stack space required by a serial execution of a Cilk program. The stack space of a $P$–worker execution using a heap–based cactus stack is at most $S_P \leq P S_1$.

**Proof.** Cilk’s work–stealing algorithm maintains the busy–leaves property: Every active leaf frame has a worker executing it. ■
Example: D&C Matrix Multiplication

```cpp
template <typename T> 
void MMult(T *C, T *A, T *B, int n, int size) {
    T *D = new T[n*n];
    //base case & partition matrices
    cilk_spawn MMult(C11, A11, B11, n/2, size);
    cilk_spawn MMult(C12, A11, B12, n/2, size);
    cilk_spawn MMult(C22, A21, B12, n/2, size);
    cilk_spawn MMult(C21, A21, B11, n/2, size);
    cilk_spawn MMult(D11, A12, B21, n/2, n);
    cilk_spawn MMult(D12, A12, B22, n/2, n);
    cilk_spawn MMult(D22, A22, B22, n/2, n);
    MMult(D21, A22, B21, n/2, n);
    cilk_sync;
    MAdd(C, D, n, size); // C += D;
    delete[] D;
}
```

Notice that allocations of the temporary matrix D obey a stack discipline.
Analysis of D&C Matrix Mult.

**Work:** \( M_1(n) = \Theta(n^3) \)

**Span:** \( M_\infty(n) = \Theta(\lg^2 n) \)

**Space:** \( S_1(n) = S_1(n/2) + \Theta(n^2) \)

By the busy-leaves property, we have

\[ S_p(n) = O(Pn^2). \]

We can actually prove a stronger bound.
Worst-Case Recursion Tree

We have $8^k = P$, which implies that $k = \log_8 P = (\lg P)/3$. The cost per level grows geometrically from the root to level $k$ and then decreases geometrically from level $k$ to the leaves. Thus, the space is $\Theta(P(n/2^{(\lg P)/3})^2) = \Theta(P^{1/3}n^2)$.
Heap-Based Linkage

**Problem:** Parallel functions fail to interoperate with legacy and third-party serial binaries.

Cilk Plus uses a less space-efficient strategy that preserves interoperability by using a pool of linear stacks.