Lecture 13
Parallel Storage Allocation
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?

**B.** Small blocks!

**S.** Why?

**D.** 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.
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?

**B.** Small blocks!

**S.** Why?

**D.** 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.

Write $y$
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 [BMBW2000]
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.
Hoard Allocation

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;
};
```
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

$$\frac{A}{U} = 1 + O\left(\frac{SP}{U}\right).$$
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.

invocation tree  
views of stack
Heap–Based Cactus Stack

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

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

By the busy-leaves property, we have \( S_p(n) = O(P n^2) \).

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 $\Theta(P(n/2^{(\lg P)/3})^2) = \Theta(P^{1/3} n^2)$. 

Branch fully (8-way) until we get to a level $k$ with $P$ nodes and then branch serially from there on.
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.
The Cactus–Stack Problem

- **Series–parallel reciprocity**
  Parallel functions should be able to call or spawn legacy (or third-party) serial functions, and legacy serial functions should be able to call parallel functions.

- **Good performance**
  The scheduler should guarantee near–perfect linear speedup on applications with sufficient parallelism.

- **Bounded stack space**
  The cactus stack should not consume much more space per worker than does a linear stack in a serial execution.

Simultaneously.
A parallel function can either **call** or **spawn** a legacy serial function, since it “knows” the stack protocol.
Serial–Parallel Reciprocity

At some cost in performance, a wrapper can enable a legacy serial function to call a parallel function without “knowing” the heap protocol, but problems remain.
Quadratic Stack Growth [Robison08]

Assume one linear stack per worker

Depth = d

Repeat d times

\[ \ldots \]
Quadratic Stack Growth [Robison08]

Assume one linear stack per worker

Depth = d

Repeat d times

The blue worker repeatedly blocks, then steals, using \( \Theta(d^2) \) stack space.
## Cactus–Stack Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>SP Reciprocity</th>
<th>Time Bound</th>
<th>Space Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Recompile Everything</td>
<td>😞</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2. One Stack Per Worker</td>
<td>✔</td>
<td>✔</td>
<td>😞</td>
</tr>
<tr>
<td>3. Limited–Depth Stacks</td>
<td>✔</td>
<td>😞</td>
<td>✔</td>
</tr>
<tr>
<td>4. Depth–Restricted Stealing</td>
<td>✔</td>
<td>😞</td>
<td>✔</td>
</tr>
<tr>
<td>5. New Stack When Needed</td>
<td>✔</td>
<td>✔</td>
<td>😞</td>
</tr>
<tr>
<td>6. Recycle Ancestor Stacks</td>
<td>✔</td>
<td>✔</td>
<td>😞</td>
</tr>
<tr>
<td>7. TLMM Cactus Stacks</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Thread-local memory mapped (TLMM) region: A virtual-address range in which each worker can map physical memory independently.

Idea: Align the stacks for each worker in the TLMM region.
Use a standard linear stack in virtual memory.

**Unreasonable simplification:** Assume that we can map with arbitrary granularity.
Upon a steal, map the physical memory corresponding to the **stolen prefix**, i.e. the stolen frame and its ancestors to the same virtual addresses in the thief as in the victim.

**Unreasonable simplification:** Assume that we can map with arbitrary granularity.
Subsequent spawns and calls grow downward in the thief’s independently mapped TLMM region.

Unreasonable simplification: Assume that we can map with arbitrary granularity.
Subsequent spawns and calls grow downward in the thief’s independently mapped TLMM region.

Unreasonable simplification: Assume that we can map with arbitrary granularity.
Basic Cilk–M Idea

Upon a steal, map the physical memory corresponding to the **stolen prefix**, i.e. the stolen frame and its ancestors to the same virtual addresses in the thief as in the victim.

**Unreasonable simplification:** Assume that we can map with arbitrary granularity.
Basic Cilk-M Idea

Subsequent spawns and calls grow downward in the thief’s independently mapped TLMM region.

Unreasonable simplification: Assume that we can map with arbitrary granularity.
Handling Page Granularity

Upon a steal, map the physical pages corresponding to the stolen prefix.

Page size

© 2013 Charles E. Leiserson, Saman P. Amarasinghe and I-Ting Angelina Lee
Handling Page Granularity

Advancing the stack pointer avoids overwriting other frames on the page, at the cost of *fragmentation.*
The thief then resumes the stolen frame and executes normally in its own TLMM region.
Handling Page Granularity

The thief then resumes the stolen frame and executes normally in its own TLMM region.
Handling Page Granularity

Once again, the stack pointer must be advanced, which causes additional fragmentation.
Handling Page Granularity

Once again, the stack pointer must be advanced, which causes additional *fragmentation*. 

Once again, the stack pointer must be advanced, which causes additional *fragmentation*.
Cilk–M Guarantees

- **Time bound:** $T_p = T_1 / P + c T_\infty$, where $c = O(S_1 + D)$.

- **Space bound:** $S_p / P \leq S_1 + D$, where $S_1$ is measured in pages.
  - Cilk–M employs a reclaiming heuristic to mitigate the effects of fragmentation.

- **SP reciprocity:**
  - Backward compatible linkage.
  - No need to distinguish function types.
  - Parallelism or not is dictated only by how a function is invoked (*spawn* vs. *call*).