6.895 Theory of Parallel Systems Lecture 6

Analyzing Space Bounds

Lecturer: Charles Leiserson *Scribe: Jeremy Fineman and Siddhartha Sen*

Lecture Summary

1. *Bounds on space requirements*
   We look at upper and lower bounds on the space requirements of parallel Cilk programs.
2. *Memory Allocators*
   We compare different types of heap storage memory allocators, and we examine the Hoard memory allocator in detail.
3. *Project Ideas*
   We present several project ideas based on the topics in these notes.

1 Bounds on space requirements

In this section, we look at upper and lower bounds on space requirements for Cilk programs. We begin with a general upper bound and then consider a tighter bound for a specific matrix multiplication algorithm. Next, we look at a lower bound on general multithreaded computations. Finally, we mention the Blelloch-Narlikar algorithm, which provides a better upper bound on space for Cilk-like dags.

General upper bound

The Cilk scheduler is a busy-leaves scheduler. As a result, we can show an upper bound on space which is linear to P, the number of processors in the system.

We can view the execution of a parallel program as a cactus stack. Let S_1 be the amount of space required to execute the program on a single processor. Figure 1 shows that S_1 can never exceed the maximum height of the stack. When run on a P-processor system, the Cilk scheduler ensures that there are at most P leaves allocated. From Figure 1, we see that there are at most P stacks that have been allocated by the system, where each stack has a maximum height of S_1. Thus, we derive the general upper bound of $S_P \leq P \cdot S_1$.

Applying the general upper bound on space

We apply the general upper bound for space requirements to the divide-and-conquer matrix multiplication algorithm. Each iteration of this algorithm on an $n \times n$ matrix spawns 8 multiplications of submatrices of size $n/2 \times n/2$. For each of these multiplications, we need a temporary matrix to store the result.

For serial execution, we can describe the total space used with the following recurrence, illustrated by the recursion tree in Figure 2:

$$\begin{aligned} S_1(n) &= S_1\left(\frac{n}{2}\right) + n^2 \\ &= \Theta(n^2)\ . \end{aligned} \tag{1}$$

Therefore, a P-processor execution uses $O(P \cdot S_1) = O(Pn^2)$ space.

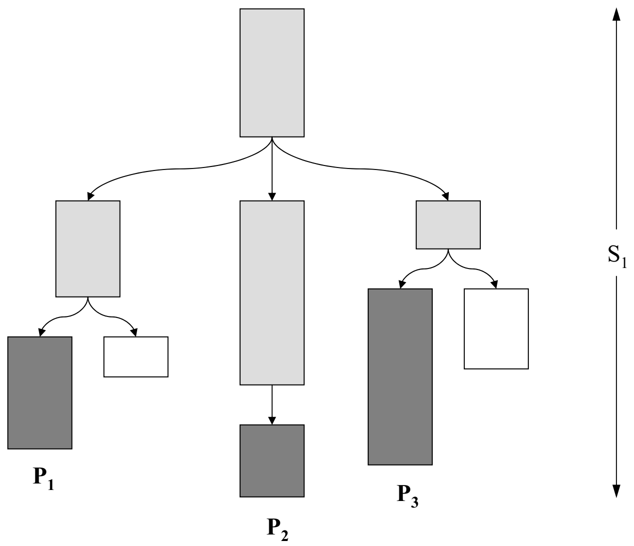

Figure 1: A cactus stack. Cilk uses a busy-leaves scheduler, which means that at most P leaves are scheduled at any given time. The shaded stack frames represent allocated memory, and the darker frames represent leaves currently being executed by the processors. In this example, $P = 3$.

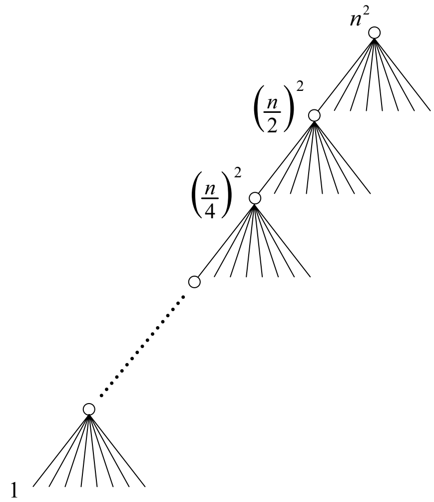

Figure 2: The recursion tree for the divide-and-conquer matrix multiplication algorithm on a single processor. The values shown indicate the space used at each level.

Stronger bound for divide-and-conquer matrix multiplication

We can derive a better bound on space for the divide-and-conquer algorithm. The general upper bound we derived earlier counts the maximum height (S_1) of the stack P times. As a result, many of the frames in the shallower levels are counted more than once; the top level frame, for example, is counted P times, but its space is only allocated once. Since shallower levels require more space than deeper ones, the problem of double-counting is worse towards the top of the tree. This poor accounting results in a much weaker bound than we can achieve.

Consider the worst case for the divide-and-conquer algorithm. Given that space requirements drop drastically at lower levels of the tree, the worst case is fully parallel at the top until there are P leaves, then serial thereafter. The recursion tree in /figrefmmultworstcasespace illustrates this transition. We can solve the recursion tree by considering the space used at each level. The top level requires n^2 space. The next level requires $8(n/2)^2$ space. If we continue this patten, the level with P leaves requires $8^{\log_8 P}(n/2^{\log_8 P})^2 = P(n/P^{1/3})^2$ space, since the depth of this level is $\log_8 P$. The serial execution following from each of the P leaves requires only a constant times $(n/P^{1/3})^2$ space. Summing the space for all the levels yields $O(n^2 P^{1/3})$ total space.

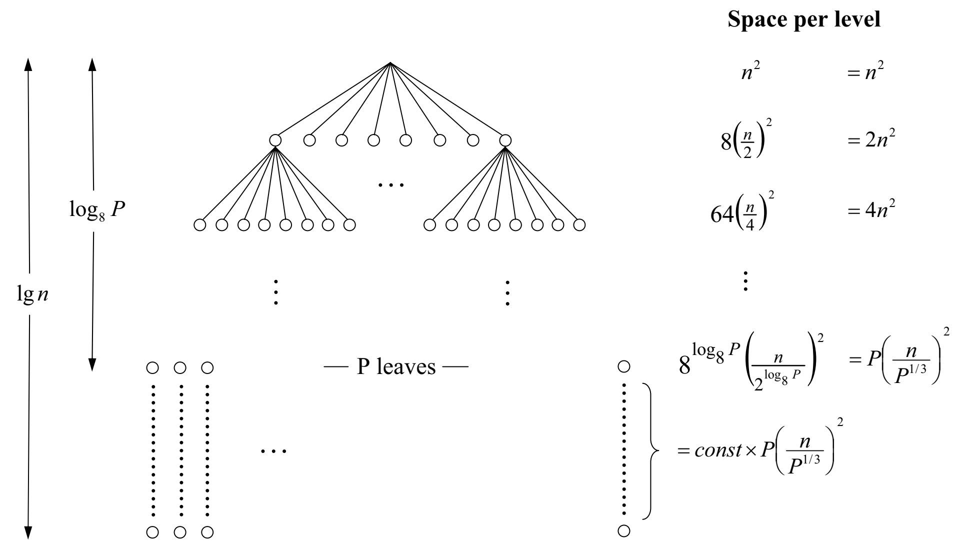

Figure 3: The worst case for the divide-and-conquer matrix multiplication algorithm running on P processors. The execution is fully parallel until we reach P leaves, after which the execution continues serially on each processor (since Cilk uses a busy-leaves scheduler).

Lower bound

We now consider a lower bound on space for computation dags. We know that Cilk produces series-parallel execution dags. Earlier, we showed that a busy-leaves scheduler results in an expansion of space that is linear to the number of processors—that is, $S_P \leq P \cdot S_1$. Both of these properties are necessary to get good space bounds; without these properties, we can find a multithreaded computation such that $T_P \leq T_1/2$ implies $S_P \approx \sqrt{T_1}$. In other words, space grows with work.

Definition 1 (Depth-first computation dag) *In a depth-first computation dag, a left to right depth-first traversal visits all threads on which a given thread depends before that thread. In other words, there exists a serial execution for the dag. The dag does not have to be series-parallel.*

Figure 4 shows an example of a depth-first computation dag. The edge from node K to D cannot exist in the graph because it violates the dependency condition above; namely, that a thread (node D) is only visited after all threads on which it depends (nodes K and C) are visited first. Clearly, a left to right depth-first traversal cannot visit both C and K before visiting D.

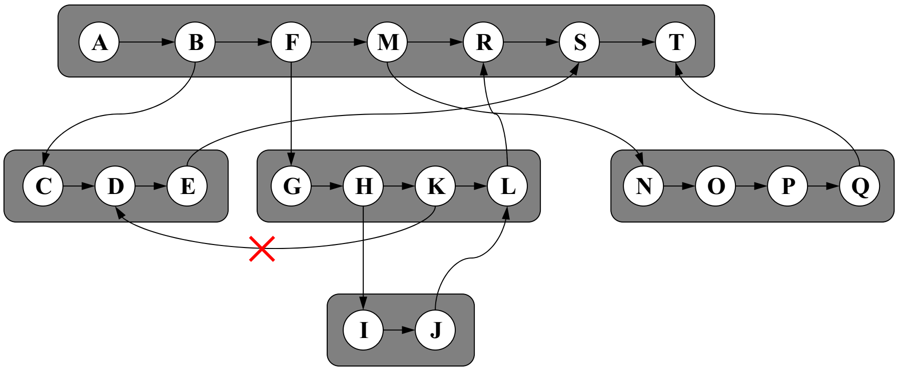

Figure 4: A depth-first computation dag. The nodes represent threads of execution, and the arrows represent the continuation, spawn, and return edges. An alphabetical listing of the nodes illustrates a left to right depth-first traversal of the threads.

Theorem 2 *For $S_1 \geq 4$ and $T_1 \geq 16S_1^2$, there exists a depth-first computation dag with work T_1, critical path length $T_\infty \leq 8\sqrt{T_1}$, and stack depth S_1 such that $\forall\rho$ and $1 \leq \rho \leq 1/8T_1/T_\infty$, if $T_P \leq T_1/\rho$, then $S_P \geq 1/4(\rho-1)\sqrt{T_1} + S_1$.*

Proof Idea In the dag in Figure 5, there are dependencies from the last threads of each of the C_i to the first threads of C_{i+1}. To get speedup, one must continually spawn new procedures, but they stall immediately, consuming space in the form of activation records. For details of this proof, see [2]. □

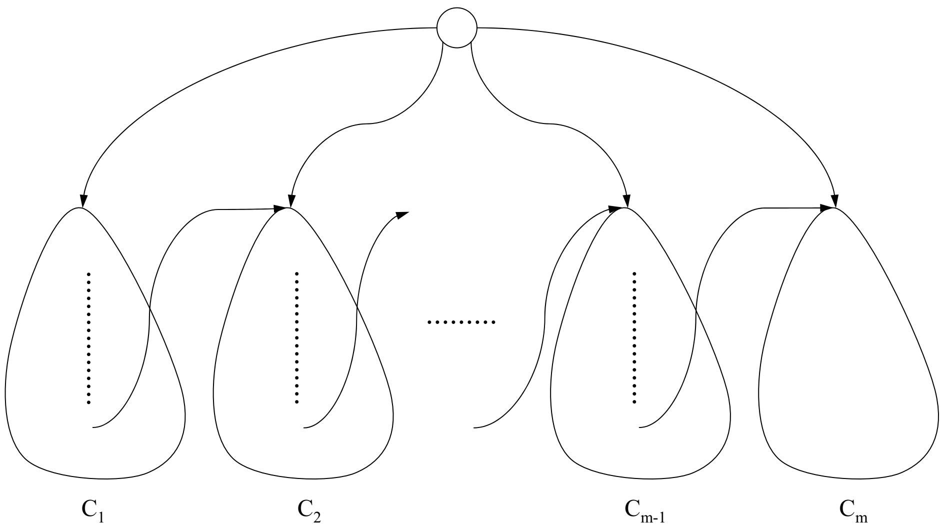

Figure 5: Depth-first computation dag for the proof of Theorem 2.

Blelloch-Narlikar algorithm

Blelloch and Narlikar prove a bound of $S_P \leq S_1 + O(P \cdot T_\infty)$ for Cilk-like dags. The main idea of their algorithm is to execute the P leftmost leaves of the computation dag in depth-first order.

Unfortunately, this approach has poor locality because if a new thread spawns on the left, then we have to rearrange processors to work on that thread. We can improve locality by executing any P of the nP leftmost leaves, where n is some tunable parameter.

2 Memory Allocators

In this section, we look at different types of heap storage memory allocators for parallel systems and examine the strengths and weaknesses of each. The main idea in designing a memory allocator is to ask the operating system for space as infrequently as possible. Once space is allocated by the OS, we can keep reusing it. In a serial allocator, memory is allocated to a process in chunks that are powers of 2 in size.

We use four measures to compare the different memory allocators:

- ***Overhead***, T_1/T_{serial}, is the increase in time that results from running a serial program in a parallel memory allocation environment.
- ***Scalability***, T_1/PT_P, reflects the additional time incurred by the parallel memory allocator (on P processors).
- ***False sharing*** occurs when two processors share the same cache line. If both processors write a different variable in the same line, there is no actual contention, but the memory system may behave as if a true conflict exists.
- ***Fragmentation*** is the ratio of memory allocated to memory used. When a thread performs a `malloc` followed by a `free`, we consider the memory to still be allocated. Our goal is to minimize the ratio A/U, where A is the maximum amount of space allocated so far, and U is the maximum amount of space used so far.

Global heap

In a global heap allocator, all processors share a single heap with a global lock. The overhead of allocating memory in this scheme is the global lock. Scalability is poor due to lock contention. Furthermore, false sharing is high for small objects as they are more likely to occur on the same cache lines. Fragmentation, on the other hand, is very low; it is comparable to that of a serial processor.

Local heap

A local heap allocator has a separate heap for each thread. Since there are no locks, local heaps do not introduce any overhead, and scalability is excellent. False sharing is minimal, but it can still occur when one thread passes memory to another. Fragmentation, on the other hand, can grow without bound.

In general, the way fragmentation grows is through ***blow-up*** or ***memory-drift***. Suppose one thread allocates a variable x_1, but a different thread frees it. If this behavior repeats, the second thread will eventually have a huge free list, while the first will continue to allocate memory from the OS.

Local heap with ownership

Similar to a local heap allocator, each thread in this scheme has a separate heap, but data is always returned to the thread that allocated it when it is freed. This scheme has additional overhead due to the synchronization between heaps. The scalability and false sharing are comparable to those of the local heap allocator. Fragmentation, however, approaches P rather than growing without bound.

The Hoard allocator

We now consider the Hoard allocator, designed by Berger et. al. [1]. The Hoard uses P local heaps and one global heap. Memory is allocated in page-sized ***superblocks***; we let S be the superblock size. Without loss of generality, we can treat all allocated objects as being the same size.

The Hoard `malloc` algorithm looks like the following:

`malloc`:
  if there exists a free object in local heap
    then return it
  else if there exists a superblock in global heap
    then move it to local heap
    return an object
  else
    allocate a superblock from the OS
    move superblock to local heap
    return an object .

When the fraction of free memory in the local heap is too big, the Hoard removes a mostly empty superblock from the local heap and returns it to the global heap. Let u_i be the space used in the i-th heap, and let a_i be the storage allocated to that heap. Let f and K be tunable parameters, usually set to about $1/4$ and 4 respectively. If $u_i < (1-f)a_i$ and $u_i < a_i - KS$, then we move an f-empty superblock to the global heap.

The Hoard allocator maintains the following invariant:

Invariant 3 $u_i \geq a_i - KS$ *or* $u_i \geq (1-f)a_i$.

The Hoard maintains Invariant 3 because moving a superblock reduces a_i by S and u_i by at most $(1-f)S$.

The Hoard always allocates from the fullest superblock. It maintains a constant overhead by using a binning strategy. In other words, it keeps a table that maps each superblock to the number of objects in that superblock. Whenever a local heap releases a superblock, it can lookup the emptiest superblock from the table in constant time.

Theorem 4 *Recall that A is the maximum memory allocated up to now, and U is the maximum memory in use up to now. Then $A = O(U + P)$.*

Proof Consider a time t when the last allocation from the OS occurred. At that time, the global heap must have been empty, or we would have passed a superblock from the global heap rather than allocating it from the OS. Thus, we have $A = \sum_{i=1}^{P} a_i(t)$ and $a_0(t) = 0$, where a_0 is the storage allocated in the global heap.

All heaps must satisfy either $u_i \geq a_i - KS$ or $u_i \geq (1-f)a_i$, from Invariant 3. We can rearrange these inequalities to read $u_i + KS \geq a_i$ and $u_i/(1-f) \geq a_i$ respectively. Without loss of generality, we assume that the heaps that satisfy the first term of the invariant are numbered $1 \ldots R$, while those that satisfy the second term are numbered $R+1 \ldots P$. If any heap satisfies both terms of the invariant, we place it in either group. Therefore, we have the following:

$$
\begin{aligned}
A &= \sum_{i=1}^{P} a_i(t) \\
&= \sum_{i=1}^{R} a_i(t) + \sum_{i=R+1}^{P} a_i(t) \\
&\le \sum_{i=1}^{R} (u_i(t) + KS) + \sum_{i=R+1}^{P} \left(\frac{u_i(t)}{1-f}\right) \\
&\le PKS + \frac{U}{1-f} \\
&= O(U+P) \ . \qquad (2)
\end{aligned}
$$

$\square$

From Equation (2), the fragmentation or blow-up is at most $A/U = O(1 + P/U)$. This bound reduces to $O(1)$ if $P = O(U)$, which is usually a fair assumption.

3 Project Ideas

The following is a list of project ideas based on some of the concepts covered in this lecture:

- We can introduce some *slop* to the Blelloch-Narlikar algorithm to increase locality. For example, we can look at the $2P$ leftmost leaves rather than just the P leftmost leaves. This topic is open for further exploration.
- The Hoard allocator has only one global heap. Perhaps we can get a better bound by introducing more heaps or a tree of heaps.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambrdige, MA, November 2000.

[2] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K.H. Randall. An analysis of dag-consistent distributed shared memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.