6.895 Theory of Parallel Systems Lecture 2

Cilk, Matrix Multiplication, and Sorting

Lecturer: Charles Leiserson *Scribe: Jason Hickey and Tyeler Quentmeyer*

Lecture Summary

1. *Parallel Processing With Cilk*
   This section provides a brief introduction to the Cilk language and how Cilk schedules and executes parallel processes.

2. *Parallel Matrix Multiplication*
   This section shows how to multiply matrices efficiently in parallel.

3. *Parallel Sorting*
   This section describes a parallel implementation of the merge sort algorithm.

1 Parallel Processing With Cilk

We need a systems background to implement and test our parallel systems theories. This section gives an introduction to the Cilk parallel-programming language. It then gives some background on how Cilk schedules parallel processes and examines the role of race conditions in Cilk.

1.1 Cilk

This section introduces Cilk. Cilk is a version of C that runs in a parallel-processing environment. It uses the same syntax as C with the addition of the keywords `spawn`, `sync`, and `cilk`. For example, the Fibonacci function written in Cilk looks like this:

```
cilk int fib(int n)
{
   if(n < 2) return n;
   else
    {
        int x, y;
        x = spawn fib(n-1);
        y = spawn fib(n-2);
        sync;
        return(x + y);
    }
}
```

Cilk is a ***faithful*** extension of C, in that if the Cilk keywords are elided from a Cilk program, the result is a C program which implements the Cilk semantics.

A function preceded by `cilk` is defined as a Cilk function. For example,

```
cilk int fib(int n);
```

defines a Cilk function called `fib`. Functions defined without the `cilk` keyword are typical C functions. A function call preceded by the `spawn` keyword tells the Cilk compiler that the function call can be made

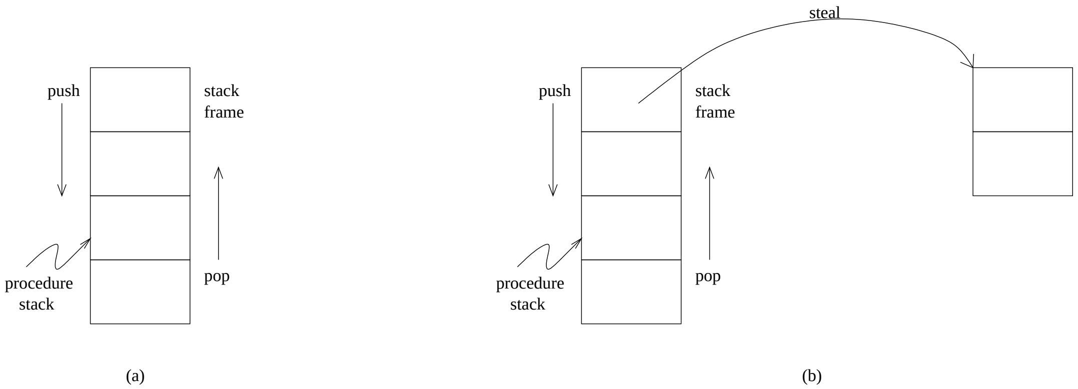

Figure 1: Call stack of an executing process. Boxes represent stack frames.

asynchronously in a concurrent thread. The `sync` keyword forces the current thread to wait for asynchronous function calls made from the current context to complete.

Cilk keywords introduce several idiosyncracies into the C syntax. A Cilk function cannot be called with normal C calling conventions – it must be called with `spawn` and waited for with `sync`. The `spawn` keyword can only be applied to a Cilk function. The `spawn` keyword cannot occur within the context of a C function. Refer to the Cilk manual for more details.

1.2 Parallel-Execution Model

This section examines how Cilk runs processes in parallel. It introduces the concepts of work-sharing and work-stealing and then outlines Cilk's implementation of the work-stealing algorithm.

Cilk processes are scheduled using an online greedy scheduler. The performance bounds of the online scheduler are close to the optimal offline scheduler. We will look at provable bounds on the performance of the online scheduler later in the term.

Cilk schedules processes using the principle of work-stealing rather than work-sharing. Work-sharing is where a thread is scheduled to run in parallel whenever the runtime makes an asynchronous function call. Work-stealing, in contrast, is where a processor looks around for work whenever it becomes idle.

To better explain how Cilk implements work-stealing, let us first examine the call stack of a vanilla C program running on a single processor. In figure 1, the stack grows downward. Each stack frame contains local variables for a function. When a function makes a function call, a new stack frame is pushed onto the stack (added to the bottom) and when a function returns, it's stack frame is popped from the stack. The call stack maintains synchronization between procedures and functions that are called.

In the work-sharing scheme, when a function is spawned, the scheduler runs the spawned thread in parallel with the current thread. This has the benefit of maximizing parallelism. Unfortunately, the cost of setting up new threads is high and should be avoided.

Work-stealing, on the other hand, only branches execution into parallel threads when a processor is idle. This has the benefit of executing with precisely the amount of parallelism that the hardware can take advantage of. It minimizes the number of new threads that must be setup. Work-stealing is the lazy way to put off work for parallel execution until parallelism actually occurs. It has the benefit of running with the same efficiency as a serial program in a uniprocessor environment.

Another way to view the distinction between work-stealing and work-sharing is in terms of how the scheduler walks the computation graph. Work-sharing branches as soon and as often as possible, walking the computation graph with a breadth-first search. Work-stealing only branches when necessary, walking the graph with a depth-first search.

Cilk's implementation of work-stealing avoids running threads that are likely to share variables by scheduling threads to run from the other end of the call stack. When a processor is idle, it chooses a random processor

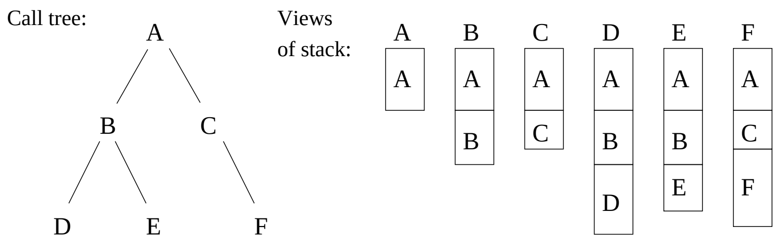

Figure 2: Example of a call stack shown as a cactus stack, and the views of the stack as seen by each procedure. Boxes represent stack frames.

and finds the sleeping stack frame that is closest to the base of that processor's stack and executes it. This way, Cilk always parallelizes code execution at the oldest possible code branch.

1.3 Cactus Stack

Cilk uses a ***cactus stack*** to implement C's rule for sharing of function-local variables. A cactus stack is a parallel stack implemented as a tree. A push maps to following a branch to a child and a pop maps to returning to the parent in a tree. For example, the cactus tree in figure 2 represents the call stack constructed by a call to A in the following code:

```
void A(void)
{
    B();
    C();
}

void B(void)
{
    D();
    E();
}

void C(void)
{
    F();
}

void D(void) {}

void E(void) {}

void F(void) {}
```

Cilk has the same rules for pointers as C. Pointers to local variables can be passed downwards in the call stack. Pointers can be passed upward only if they reference data stored on the heap (allocated with `malloc`). In other words, a stack frame can only see data stored in the current and in previous stack frames. Functions cannot return references to local variables.

The complete call tree is shown in figure 2. Each procedure sees a different view of the call stack based on how it is called. For example, B sees a call stack of A followed by B, D sees a call stack of A followed by B followed by D and so on. When procedures are run in parallel by Cilk, the running threads operate on their view of the call stack. The stack maintained by each process is a reference to the actual call stack, not a copy of it. Cilk maintains coherence among call stacks that contain the same frames using methods that we will discuss later.

1.4 Race Conditions

The single most prominent reason that parallel computing is not widely deployed today is because of race conditions. Identifying and debugging race conditions in parallel code is hard. Once a race condition has been found, no methodology currents exists to write a regression test to ensure that the bug is not reintroduced during future development. For these reasons, people do not write and deploy parallel code unless they absolutely must. This section examines an example race condition in Cilk.

Consider the following code:

```
cilk int foo(void)
{
    int x = 0;
    spawn bar(&x);
    spawn bar(&x);
    sync;
    return x;
}

cilk void bar(int *p)
{
    *p += 1;
}
```

If this were a serial code, we would expect that foo returns 2. What value is returned by foo in the parallel case? Assume the increment performed by bar is implemented with assembly that looks like this:

```
read x
add
write x
```

Then, the parallel execution looks like the following:

```
bar 1:
read x (1)
add
write x (2)

bar 2:
read x (3)
add
write x (4)
```

where bar 1 and bar 2 run concurrently. On a single processor, the steps are executed (1) (2) (3) (4) and foo returns 2 as expected. In the parallel case, however, the execution could occur in the order (1) (3) (2) (4), in which case foo would return 1. The simple code exhibits a race condition. Cilk has a tool called the Nondeterminator which can be used to help check for race conditions.

2 Matrix Multiplication and Merge Sort

In this section we explore multithreaded algorithms for matrix multiplication and array sorting. We also analyze the work and critical path-lengths. From these measures, we can compute the parallelism of the algorithms.

2.1 Matrix Multiplication

To multiply two $n \times n$ matrices in parallel, we use a recursive algorithm. This algorithm uses the following formulation, where matrix A multiplies matrix B to produce a matrix C:

$$\begin{aligned}\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.\end{aligned}$$

This formulation expresses an $n \times n$ matrix multiplication as 8 multiplications and 4 additions of $(n/2) \times (n/2)$ submatrices. The multithreaded algorithm MULT performs the above computation when n is a power of 2. MULT uses the subroutine ADD to add two $n \times n$ matrices.

```
MULT(C, A, B, n)
   if n = 1
      then C[1,1] ← A[1,1] · B[1,1]
      else allocate a temporary matrix T[1..n, 1..n]
           partition A, B, C and T into (n/2) × (n/2) submatrices
           spawn MULT(C11, A11, B11, n/2)
           spawn MULT(C12, A11, B12, n/2)
           spawn MULT(C21, A21, B11, n/2)
           spawn MULT(C22, A21, B12, n/2)
           spawn MULT(T11, A12, B21, n/2)
           spawn MULT(T12, A12, B22, n/2)
           spawn MULT(T21, A22, B21, n/2)
           spawn MULT(T22, A22, B22, n/2)
           sync
           spawn ADD(C, T, n)
           sync
```

```
ADD(C, T, n)
   if n = 1
      then C[1,1] ← C[1,1] + T[1,1]
      else partition C and T into (n/2) × (n/2) submatrices
           spawn ADD(C11, T11, n/2)
           spawn ADD(C12, T12, n/2)
           spawn ADD(C21, T21, n/2)
           spawn ADD(C22, T22, n/2)
           sync
```

The analysis of the algorithms in this section requires the use of the Master Theorem. We state the Master Theorem here for convenience.

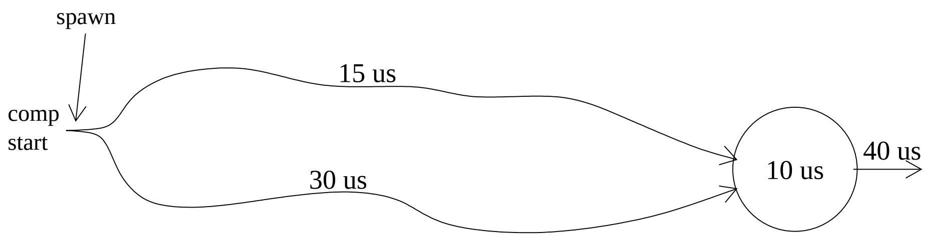

Figure 3: Critical path. The squiggles represent two different code paths. The circle is another code path.

Theorem 1 (Master Theorem) *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.*
2. *If $f(n) = \Theta(n^{log_b a} lg^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{log_b a} lg^{k+1} n)$.*
3. *If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

We begin by analyzing the work for Mult. The work is the running time of the algorithm on one processor, which we compute by solving the recurrence relation for the serial equivalent of the algorithm. We note that the matrix partitioning in Mult and Add takes O(1) time, as it requires only a constant number of indexing operations. For the subroutine Add, the work at the top level (denoted $A_1(n)$) then consists of the work of 4 problems of size $n/2$ plus a constant factor, which is expressed by the recurrence

$$\begin{aligned} A_1(n) &= 4A_1(n/2) + \Theta(1) && (1) \\ &= \Theta(n^2). && (2) \end{aligned}$$

We solve this recurrence by invoking case 1 of the Master Theorem. Similarly, the recurrence for the work of Mult (denoted $M_1(n)$):

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) && (3) \\ &= \Theta(n^3). && (4) \end{aligned}$$

We also solve this recurrence with case 1 of the Master Theorem. The work is the same as for the traditional triply-nested-loop serial algorithm.

The critical-path length is the maximum path length through a computation, as illustrated by figure 3. For Add, all subproblems have the same critical-path length, and all are executed in parallel. The critical-path length (denoted $A_\infty(n)$) is a constant plus the critical-path length of one subproblem, and is represented by the recurrence (sovled by case 2 of the Master Theorem):

$$\begin{aligned} A_\infty &= A_\infty(n/2) + \Theta(1) && (5) \\ &= \Theta(\lg n). && (6) \end{aligned}$$

Using this result, the critical-path length for Mult (denoted $M_\infty(n)$) is

$$\begin{aligned} M_\infty &= M_\infty(n/2) + \Theta(\lg n) && (7) \\ &= \Theta(\lg^2 n), && (8) \end{aligned}$$

by case 2 of the Master Theorem. From the work and critical-path length, we compute the parallelism:

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n). \tag{9}$$

As an example, if $n = 1000$, the parallelism $\approx 10^7$. In practice, multiprocessor systems don't have more that $\approx$ 64,000 processors, so the algorithm has more than adequate parallelism.

In fact, it is possible to trade parallelism for an algorithm that runs faster in practice. Mult may run slower than an in-place algorithm because of the hierarchical structure of memory. We introduce a new algorithm, Mult-Add, that trades parallelism in exchange for eliminating the need for the temporary matrix T.

```
MULT-ADD(C, A, B, n)
if n = 1
   then C[1,1] ← C[1,1] + A[1,1] · B[1,1]
   else  partition A, B, and C into (n/2) × (n/2) submatrices
            spawn MULT(C11, A11, B11, n/2)
            spawn MULT(C12, A11, B12, n/2)
            spawn MULT(C21, A21, B11, n/2)
            spawn MULT(C22, A21, B12, n/2)
            sync
            spawn MULT(C11, A12, B21, n/2)
            spawn MULT(C12, A12, B22n/2)
            spawn MULT(C21, A22, B21, n/2)
            spawn MULT(C22, A22, B22, n/2)
            sync
```

The work for Mult-Add (denoted $M'_1(n)$) is the same as the work for Mult, $M'_1(n) = \Theta(n^3)$. Since the algorithm now executes four recursive calls in parallel followed in series by another four recursive calls in parallel, the critical-path length (denoted $M'_\infty(n)$) is

$$M'_\infty(n) = 2M'_\infty(n/2) + \Theta(1) \tag{10}$$
$$= \Theta(n) \tag{11}$$

by case 1 of the Master Theorem. The parallelism is now

$$M'_1(n)/M'_\infty = \Theta(n^2). \tag{12}$$

When $n = 1000$, the parallelism $\approx 10^6$, which is still quite high.

The naive algorithm (M'') that computes n^2 dot-products in parallel yields the following theoretical results:

$$M''_1(n) = \Theta(n^3) \tag{13}$$
$$M''_\infty = \Theta(\lg n) \tag{14}$$
$$=> Parallelism = \Theta(n^3/\lg n). \tag{15}$$

Although it does not use temporary storage, it is slower in practice due to less memory locality.

2.2 Sorting

In this section, we consider a parallel algorithm for sorting an array. We start by parallelizing the code for Merge-Sort while using the traditional linear time algorithm Merge to merge the two sorted subarrays.

```
MERGE-SORT(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
            spawn MERGE-SORT(A, p, q)
            spawn MERGE-SORT(A, q + 1, r)
            sync
            MERGE(A, p, q, r)
```

Since the running time of MERGE is $\Theta(n)$, the work (denoted $T_1(n)$) for MERGE-SORT is

$$T_1(n) = 2T_1(n/2) + \Theta(n) \tag{16}$$
$$= \Theta(n \lg n) \tag{17}$$

by case 2 of the Master Theorem. The critical-path length (denoted $T_\infty(n)$) is the critical-path length of one of the two recursive spawns plus that of MERGE:

$$T_\infty(n) = T_\infty(n/2) + \Theta(n) \tag{18}$$
$$= \Theta(n) \tag{19}$$

by case 3 of the Master Theorem. The parallelism, $T_1(n)/T_\infty = \Theta(\lg n)$, is not scalable. The bottleneck is the linear-time MERGE. We achieve better parallelism by designing a parallel version of MERGE.

```
P-MERGE(A[1..l], B[1..m], C[1..n])
if m > l
   then spawn P-MERGE(B[1 . . m], A[1 . . l], C[1 . . n])
elseif n = 1
   then C[1] ← A[1]
elseif l = 1
   then if A[1] ≤ B[1]
            then  C[1] ← A[1]; C[2] ← B[1]
            else  C[1] ← B[1]; C[2] ← A[1]
   else  find j such that B[j] ≤ A[l/2] ≤ B[j + 1] using binary search
         spawn P-MERGE(A[1..(l/2)], B[1..j], C[1..(l/2 + j)])
         spawn P-MERGE(A[(l/2 + 1)..l], B[(j + 1)..m], C[(l/2 + j + 1)..n])
   sync
```

P-MERGE puts the elements of arrays A and B into array C in sequential order, where $n = l + m$. The algorithm finds the median of the larger array and uses it to partition the smaller array. Then, it recursively merges the lower portions and the upper portions of the arrays. The operation of the algorithm is illustrated in figure 4.

We begin by analyzing the critical-path length of P-MERGE. The critical-path length is equal to the maximum critical-path length of the two spawned subproblems plus the work of the binary search. The binary search completes in $\Theta(\lg m)$ time, which is $\Theta(\lg n)$ in the worst case. For the subproblems, half of A is merged with all of B in the worst case. Since $l \geq n/2$, at least $n/4$ elements are merged in the smaller subproblem. That leaves at most $3n/4$ elements to be merged in the larger subproblem. Therefore, the critical-path is

$$\begin{aligned} T_\infty(n) &\leq T(3/4n) + O(\lg n) \\ &= O(\lg^2 n) \end{aligned}$$

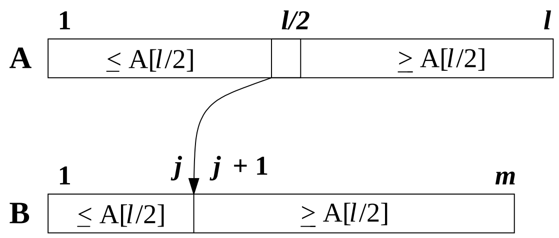

Figure 4: Find where middle element of A goes into B. The boxes represent arrays.

by case 2 of the Master Theorem. To analyze the work of P-MERGE, we set up a recurrence by using the observation that each subproblem operates on αn elements, where $1/4 \leq \alpha \leq 3/4$. Thus, the work satisfies the recurrence

$$T_1(n) \quad = T(\alpha n) + T((1-\alpha)n) + O(\lg n).$$

We shall show that $T_1(n) = \Theta(n)$ by using the substitution method. We take $T(n) \leq an - b\lg n$ as our inductive assumption, for constants $a, b > 0$. We have

$$\begin{aligned} T_(n) &\leq a\alpha n - b\lg(\alpha n) + a(1-\alpha)n - b\lg((1-\alpha)n) + \Theta(\lg n) \\ &= an - b(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\ &= an - b(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\ &= an - b\lg n - (b(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\ &\leq an - b\lg n, \end{aligned}$$

since we can choose b large enough so that $b(\lg n + \lg(\alpha(1-\alpha)))$ dominates $\Theta(\lg n)$. We can also pick a large enough to satisfy the base conditions. Thus, $T_(n) = \Theta(n)$, which is the same as the work for the ordinary MERGE. Reanalyzing the MERGE-SORT algorithm, with P-MERGE replacing MERGE, we find that the work remains the same, but the critical-path length is now

$$\begin{aligned} T_\infty(n) &= T_\infty(n/2) + \Theta(\lg^2 n) && (20) \\ &= \Theta(\lg^3 n) && (21) \end{aligned}$$

by case 2 of the Master Theorem. The parallelism is now $\Theta(n\lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$. By using a more clever algorithm, a parallelism of $\Omega(n/\lg n)$ can be achieved.

While it is important to analyze the theoritical bounds of algorithms, it is also necessary that the algorithms perform well in practice. One short-coming of MERGE-SORT is that it is not in-place. An in-place parallel version of QUICK-SORT exists, which performs better than MERGE-SORT in practice.

Additionally, while we desire a large parallelism, it is good practice to design algorithms that scale down as well as up. We want the performance of our parallel algorithms when running on one processor to compare well with the performance of the serial version of the algorithm. The best sorting algorithm to date requires only 20% more work than the serial equivalent. Coming up with a dynamic multithreaded algorithm which works well in practice is a good research project.