Massachusetts Institute of Technology — Lecture 9
6.892: Theory of Parallel Systems — March 6
Professor Charles E. Leiserson — Scribe: Michael B.Taylor

Lecture 9: Cilk, Matrix Multiplication and other Ilk

1 Reasoning about Parallel Interaction

A *thread* is a maximal sequence of statements not containing the parallel control statements: **return**, **sync**, and **spawn** . All threads of a procedure instance are executed atomically with respect to each other. There are no guarantees about threads from different procedure instances. This formulation allows us to reason about the parallel interactions of a Cilk programs.

1.1 The Cilk Inlet Construct

The **inlet** construct is used to synchronize access to common resources within a Cilk procedure instance. In the example below, the function "product" is calculating the product of the numbers a[i]..a[j]. It does this in a divide-and-conquer fashion. The inlet (mult) is used to combine the results of the first and second recursive calls. It does this by serializing access to the variable p. Only one of the spawned threads can multiply its result with p at a time. The inlet is a thread itself. The manual states that Cilk's implicit guarantee is that "the threads of a procedure instance, including its inlets, operate atomically with respect to one another." It is interesting to note that +=, -=, etc generate *implicit* inlets. The Cilk 5.0 Reference manual contains a more detailed discussion of inlets (pp 17-18.)

```
cilk int product(int *a, int i, int j)
{
  int p = 1;

  inlet void mult(int x)
  {
    p *= x;
    return;
  }

  if (i == j) return(a[i]);
  mult(spawn product(a, i, (i+j)/2);
  mult(spawn product(a, i, (i+j)/2 + 1,(i+j));
  return(p);
}
```

2 The Cilk Abort Construct

The Cilk **abort** construct is useful when a procedure realizes that work it has spawned off is unnecessary. This is a common event in programs that execute *speculatively* (heuristic search, for example) but also in day-to-day programs. For example, a chess program may have spawned off many parallel tasks to search parts of the game tree. When it finds a favorable position in the tree, it may want to tell the other tasks to stop looking. Our "product" program can also make use of the **abort** construct. If it discovers a zero in the array it knows that result of the computation will be zero. We now look at a second version of the product procedure which makes use of **abort**. The major change is on line 7. The **abort** command is a bit tricky to use. It aborts only extant children, and not future children. This is the reason for line 12. Line 12 ensures that once p becomes zero, no more children are spawned. Note, that in all cases, your programs should execute if any instance of abort were replaced with a no-op. Abort also makes no guarantees about the return values of aborted procedures. The full gory details of **abort** are described on page 18 of the manual. There have been some proposals for extensions: The *superreturn* construct would allow a value to be returned for all instances of a procedure. The *on_abort* construct would specify a cleanup routine for an aborted procedure.

```
1  cilk int product(int *a, int i, int j)
2  {
3    int p = 1;

4    inlet void mult(int x)
5    {
6      p *= x;
7      if (p == 0) abort;
8      return;
9    }

10   if (i == j) return(a[i]);
11   mult(spawn product(a, i, (i+j)/2);
12   if (p == 0) return(0);
13   mult(spawn product(a, i, (i+j)/2 + 1,(i+j));
14   return(p);
15 }
```

3 Matrix Multiplication (Re)*visited...

3.1 Simple Recursive Algorithm

For matrices C,A,B, we can recursively formulate matrix multiplication as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$ and so on. This breaks each matrix multiply operation into 8 multiplies and 4 additions of matrices of side $n/2$. The Cilk psuedo code will look something like:

```
cilk void MatrixMultiply(Matrix *C, Matrix *A, Matrix *B,int n)
{
  if n is sufficiently small
     do C = AB in an serial manner
  else
    alloca temporary matrix T
    spawn MatrixMultiply(C11,A11, B11, n/2);
    spawn MatrixMultiply(C12,A11, B12, n/2);
    spawn MatrixMultiply(C21,A21, B11, n/2);
    spawn MatrixMultiply(C22,A21, B12, n/2);
    spawn MatrixMultiply(T11,A12, B21, n/2);
    spawn MatrixMultiply(T12,A12, B22, n/2);
    spawn MatrixMultiply(T21,A22, B21, n/2);
    spawn MatrixMultiply(T22,A22, B22, n/2);
    sync;
    spawn MatrixAdd(C,T, n);
    sync;
    return;
}
```

```
cilk void MatrixAdd(Matrix *C, Matrix *T,  int n)
{
  if n is sufficiently small
    do C = C + T in an serial manner
  else
    spawn MatrixAdd(C11,T11, n/2);
    spawn MatrixAdd(C12,T12, n/2);
    spawn MatrixAdd(C21,T21, n/2);
    spawn MatrixAdd(C22,T22, n/2);
    sync;
    return;
}
```

The work for this would be:

$$\begin{aligned}
A_1(n) &= 4A_1\left(\frac{n}{2}\right) + \Theta(1) \\
&= \Theta(n^2) \\
M_1(n) &= 8M_1\left(\frac{n}{2}\right) + A_1(n) \\
&= \Theta(n^3)
\end{aligned}$$

which is the same as the serial algorithm.

The critical path would be:

$$\begin{aligned}
A_\infty(n) &= A_\infty\left(\frac{n}{2}\right) + \Theta(1) \\
&= \Theta(\lg\ n) \\
M_\infty(n) &= M_\infty\left(\frac{n}{2}\right) + A_\infty(n) \\
&= \Theta(\lg^2\ n)
\end{aligned}$$

The average parallelism is

$$\frac{M_1(n)}{M_\infty(n)} = \Theta\left(\frac{n^3}{\lg^2\ n}\right)$$

This is quite good. In fact, it may be in our interests to trade away some of the parallelism for say, space consumption.

3.2 Simple Recursive Algorithm, No Temporary Space

In the example below, we are able to eliminate usage of temporary space by dividing the multiply operation into two phases: a multiply phase and a multiply-add phase. The results of the second set of matrix multiplies will get added directly to the results of the first set. This trades critical path and parallelism off for space.

```
cilk void MatrixMultiply(Matrix *C, Matrix *A, Matrix *B,int n)
{
  if n is sufficiently small
    do C = AB in an serial manner
  else
    spawn MatrixMultiply(C11,A11, B11, n/2);
    spawn MatrixMultiply(C12,A11, B12, n/2);
    spawn MatrixMultiply(C21,A21, B11, n/2);
```

```
    spawn MatrixMultiply(C22,A21, B12, n/2);
    sync;
    spawn MatrixMultiplyAdd(C11,A12, B21, n/2);
    spawn MatrixMultiplyAdd(C12,A12, B22, n/2);
    spawn MatrixMultiplyAdd(C21,A22, B21, n/2);
    spawn MatrixMultiplyAdd(C22,A22, B22, n/2);
    sync;
    return;
}
```

The work will be the same. However, the critical path will be:

$$M_\infty(n) = 2M_\infty\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)$$

The average parallelism is

$$\frac{M_1(n)}{M_\infty(n)} = \Theta\left(n^2\right)$$

This is still mucho parallelism. One processor per element of a matrix is more than processors than one would typically have. In practice, this algorithm works better than the first, since saving space often saves time in hierarchical memory systems.

3.3 Naive Algorithm

It is possible to get an even better critical path and average parallelism. In this algorithm, we spawn off one thread for each element of the matrix. Each thread will compute the single dot-product which forms the values of the particular matrix operation. This involves summing N products, which can be done in a tree-like fashion, with depth $\Theta(lg\ n)$.

The work, critical path and average parallelism for this would be:

$$\begin{aligned}
M_1(n) &= M_1\left(\frac{n}{2}\right) + A_1(n) \\
&= \Theta(n^3) \\
M_\infty(n) &= M_\infty\left(\frac{n}{2}\right) + \Theta(1) \\
&= \Theta(\lg\ n) \\
\frac{M_1(n)}{M_\infty(n)} &= \Theta\left(\frac{n^3}{\lg\ n}\right)
\end{aligned}$$

Unfortunately, this is slow for locality reasons.

3.4 Algorithmic Mutts

It is possible to combine the our two "top-dog" algorithms (the first and second ones) to get something that has properties of both – when possible, the algorithm can choose between a better critical path and better space usage. If we are running on a fixed number of processors, there may be opportunities where space can be reused if we know that the computations that were using that space have completed. We can do this in Cilk by using the SYNCHED define. SYNCHED is 0 in the "slow" compiled version of your Cilk code, and 1 in the "fast" compiled version. This is equivalent to saying, if SYNCHED is 1, there are no spawned children of a procedure instance, otherwise SYNCHED is 0 and there do exist spawned children. This would yield code that looks something like:

```
cilk void MatrixMultiply(Matrix *C, Matrix *A, Matrix *B,int n)
{
  if n is sufficiently small
    do C = AB in an serial manner
  else
    spawn MatrixMultiply(C_11,A_11, B_11, n/2);
    spawn MatrixMultiply(C_12,A_11, B_12, n/2);
    spawn MatrixMultiply(C_21,A_21, B_11, n/2);
    spawn MatrixMultiply(C_22,A_21, B_12, n/2);
    if (SYNCHED)
      spawn MatrixMultiplyAdd(C_11,A_12, B_21, n/2);
      spawn MatrixMultiplyAdd(C_12,A_12, B_22, n/2);
      spawn MatrixMultiplyAdd(C_21,A_22, B_21, n/2);
      spawn MatrixMultiplyAdd(C_22,A_22, B_22, n/2);
    else
      alloca temporary matrix T
      spawn MatrixMultiply(T_11,A_12, B_21, n/2);
      spawn MatrixMultiply(T_12,A_12, B_22, n/2);
      spawn MatrixMultiply(T_21,A_22, B_21, n/2);
      spawn MatrixMultiply(T_22,A_22, B_22, n/2);
      sync;
      spawn MatrixAdd(C,T, n);
      sync;
  return;
}
```

This will result in better space usage. The recurrence for this becomes quite complicated. In reality, we would actually implement an algorithm like Strassen's method which has better work characteristics. This comes at the expense of space, however. The key thing is that the pseudovariable `SYNCHED` is the constant `TRUE` in the fast clone, and hence the fast clone looks identical to the no-temp multiplication algorithm.