Homework 22

Robbie McKinstry, Jack McQuown, Cyrus Ramavarapu

24 October 2016

Problem 4:

The algorithm that can be used in order to get parallel prefix to run in O(log n) time is called Scan. It is an established parallel pattern that is used when the main goal is to compute partial reductions of the entire collection, however in order to Scan to work correctly the operators used must be associative.

The basic idea of scan is to first compute the sums of every two elements in the collection, i.e. (1+2), (3+4), (5+6), $\cdots$ The second step is to then sum these values with each other, so (1+2)+(3+4), (5+6)+(7+8), $\cdots$ The third step is to then bring the first element down into its respective spot in the final collection, and continue the pattern of adding the previous summations [(1+2)+(3+4)]+[(5+6)+(7+8)]. The fourth step is to then take [(1+2)+(3+4)]+(5+6). The final step is to then add [(1+2)]+3, [(3+4)]+5, and [(5+6)]+7.

At each level of Scan there are no data dependencies to worry about, which means that the calculations at each level can be done in parallel. Unfortunately because we are limited to $n/\log n$ processors at each level we will have to do potentially n/p work. With $\log n$ levels and n/p work at each level, the total runtime for this algorithm is $O(n/p + n/\log n)$.

Problem 6:

A:

Given n^2 processors on a CREW PRAM a $\mathcal{O}(()n)$ parallel algorithm can be developed for matrix multiplication. The technique used to create this algorithm is the fabled *stare at the serial code* method. In this case, the serial code for matrix multiplication is as follows:

```
Function: Serial Matrix Mult
Input: Matrix $A_{n\times n}$, Matrix $B_{n\times n}$
Zero Matrix $C_{n\times n}$
for i=0 to n do
    for j=0 to n do
        for k=0 to n do
            C[i,j] += A[i,k]*B[k,j]
Return: C
```

Given n^2 processors, the outer two loops can be parallelized easily since there is no data dependence between them and concurrent reading is allowed. This gives the following algorithm:

```
Function: Serial Matrix Mult
Input: Matrix $A_{n\times n}$, Matrix $B_{n\times n}$
Zero Matrix $C_{n\times n}$
/* PRAGMA: parallel for                                  */
for i=0 to n do
    /* PRAGMA: parallel for                              */
    for j=0 to n do
        for k=0 to n do
            C[i,j] += A[i,k]*B[k,j]
Return: C
```

Since giving a processor to the first two loops exhausts the number of available processors, the processors given to the inner loop must each carry out the n multiplications and summations needed for *C[i,j]*. This leads to a runtime of $\mathcal{O}\big(n\big)$ for this algorithm.

The efficiency of this algorithm can be found by the following equation:

$$E(n,p) = \frac{S(n)}{pT(n,p)}$$

Where *S(n)* is the serial run time for the problem and *T(n,p)* is the parallel run time using p processors. Using the values for the above algorith, *E(n,p)* can be found to be:

$$E(n,p) = \frac{S(n)}{pT(n,p)} = \frac{n^3}{n^2 n} = 1$$

This algorithm is therefore perfectly efficient.

However, if the number of processors is reduced to $n^{1/4}$ the folding principle can be used to derive an upper bound on the new run time of this parallel algorithms. Since the number of processors decreased, the amoung of work each remaining processor must do increases. As a result, the new run will be at most $n^{11/4}$.

B:

Similar to the algorithm given in **A**, an algorithm for matrix multiplication that runs in $\mathcal{O}\big(\log n\big)$ time can be developed using n^3 processors. The primary modification to the algorithm in **A** is that the innermost loop can now be parallelized as a variation of the parallel SUM problem which takes $\mathcal{O}\big(\log n\big)$ through a divide and conquer strategy. The variation to the parallel sum in this case is that initially the processors will multiply as the first operation and then sum. In this case, the algorithm is as follows:

```
Function: Serial Matrix Mult
Input: Matrix $A_{n\times n}$, Matrix $B_{n\times n}$
Zero Matrix $C_{n\times n}$
/* PRAGMA: parallel for                                    */
for i=0 to n do
    /* PRAGMA: parallel for                                */
    for j=0 to n do
        C[i,j] = ParallelDotProduct($row_i(A)$,$col_j(B)$,n)
Return: C
```

The runtime of this algorithm is analyzed in the same way as the algorithm developed in **A**. Since the two outer loops are completely parallelized, the occur in constant time. Most of the work will be done within the ParallelDotProduct function, which, like the parallel SUM, problem takes $\mathcal{O}\big(\log n\big)$ time. As a result, the overall runtime of this algorithm is $\mathcal{O}\big(\log n\big)$.

The efficiency of this algorithm is also calculated in an identical manner. Substituting the appropriate values for this algorithm into the above equation results in the following efficiency calculation:

$$E(n,p) = \frac{S(n)}{pT(n,p)} = \frac{n^3}{n^3 \log n} = \frac{1}{\log n}$$

If the number of processors is reduced to $n^{1/4}$, the folding principle can be used to derive an upper bound on the new runtime. Since the number of processors has decreased, the remaining processors will now have to do more work. The factor by which the processors decreased is $n^{11/4}$. As a result the runtime of the algorithm on these constrained resources will be at most $n^{11/4} \log n$

C:

If only $n^3/\log n$ processors are available, the algorithm developed in **B** can still be used to perform parallel matrix multiplication in $\mathcal{O}\big(\log n\big)$ time. This is because, n^2 processors will still be used to parallelize the outermost loops for matrix multiplication. The remaining $n/\log n$ processors will then be given to the *ParallelDotProduct* problem. Like the *ParallelSum* problem, the run time of the problem stays the same despite the reduction of processors because the recursive, divide and conquer tree becomes shallower; however, the amount of work done at the leaves increases.

Given these factors, the algorithm for this problem is identical to the one given in **B**, but is repeated here for clarity.

```
Function: Serial Matrix Mult
Input: Matrix $A_{n\times n}$, Matrix $B_{n\times n}$
Zero Matrix $C_{n\times n}$
/* PRAGMA: parallel for                                    */
for $i$=0 to $n$ do
    /* PRAGMA: parallel for                                */
    for $j$=0 to $n$ do
        $C[i,j]$ = ParallelDotProduct($row_i(A)$,$col_j(B)$,$n$)
Return: $C$
```

Substituting the appropriate values into the efficiency calculation for this algorithm reveals an improvement in the efficiency as compared to the conditions given in **B**.

$$E(n,p) = \frac{S(n)}{pT(n,p)} = \frac{n^3 \log n}{n^3 \log n} = 1$$

Under these conditions, the algorithm is perfectly efficient.

The folding principle can be once again applied to this problem to place an upper bound on the runtime of this algorithm under the more restrictive conditions of having only $n^{1/4}$ processors. Similar to problem **B**, the runtime will be no worse than $n^{11/4}$ because the number of processors decreased by a factor of $n^{11/4}/\log n$.

Problem 7:

Let c be the vector of coefficients, and c_i to be the coefficient for n^i in $p(x)$. Now, to calculate $p(k)$ for a specific k, schedule in parallel each core to calculate the value of each term of the polynomial, which is $c_i * k^i$ for the ith term. With $\mathcal{O}\big(\frac{n}{lgn}\big)$ cores, this occurs in $\mathcal{O}\big(lgn\big)$ operations since there are n terms. Then, sum the terms using parallel addition like we have in the past, having each core add two terms together before adding the sum of that term and the next in the following step, until only one term remains. As we've demonstrated in the past, operations that are associative, like this addition, occur in $\mathcal{O}\big(lgn\big)$ time because each addition happens in constant time, but there are $\mathcal{O}\big(lgn\big)$ additions. As such, the time complexity is the sum of two $\mathcal{O}\big(lgn\big)$ operations, which makes the resultant time complexity of the algorithm $\mathcal{O}\big(lgn\big)$, when there are $\mathcal{O}\big(nlgn\big)$ cores.

Problem 8:

First, initialize an array of size n, with the zeroth and first indices set to 1. Next repeat the following: Populate the next $2i$ positions in the array in parallel where i is the number of repeats performed thus far. Consequently, the number of array iterations is $\mathcal{O}\big(lgn\big)$.

The crux of the problem is how to populate each of the individual array entries. At a given time, a core will have access to an array with i populated entires, and it will be tasked with populating some entry k, where $i < k \leq 2i$. The key is to compose the kth entry as a selection of smaller entires until k is only composed of known entries. For example, all entires up to $F(11)$ are known, $F(13)$ may be calculated as $F(12) + F(11)$, and $F(12)$ can be calculated as $F(11) + F(10)$. Thus, $F(13) = 2F(11) + F(10)$. As such, for each unknown $F(N)$, you add one to the number of times $F(N - 1$ must be summed and the number of times $F(N-2)$ must be summed, and you repeat this until you have only known terms and their coefficients. An optimization necessary for logarithmic time complexity is to be able to calculate what the coefficients are by knowing the distance from the kth term and the ith term.

The final entry in the array is the nth Fib number.