CS 529: Advanced Data Structures & Algorithms
Assignment 4: Parallel Algorithms

Hunter Lawrence, Andrew Struthers

April 26, 2024

1. Consider the problem of adding two $n \times n$ matrices. If it takes t_a time for one person to add two numbers, how many people do we need to minimize the total time spent to get the final answer? What will be the minimum amount of time needed to find the answer, if we assume that we have enough people? Justify your answers.

   (a) Given n tasks that take equal time, the most time-efficient way to complete all of them would be for each task be done on its own processor at the same time. Therefore, the number of processors needed to complete n tasks is also n, as each processor executes each tasks simultaneously.

   The addition of two $n \times n$ matrices requires each of the $2n^2$ elements to be added together appropriately, where each addition is a single task. Therefore the addition of two $n \times n$ matrices is equivalent to a set of n^2 tasks. By the above conclusion, the number of people required to execute the addition of two $n \times n$ matrices is n^2. Because each addition takes the same amount of time t_a, and each addition is being done at the same time, the minimal total time required is also t_a.

2. Write a CREW PRAM algorithm for adding all n numbers in a list in $O(\log n)$ time.

   (a) The standard way of adding all n numbers in a list is the naive approach of having a global *sum* variable, then iterating through the list linearly, adding the elements of the list to *sum* one by one. We can improve this to $O(\log n)$ time with the following algorithm:

Function addElements(list)

Input: n-length list of numbers
Output: sum of all elements in *list*

```
1 sum ← 0 as a variable for the sum of all elements
2 set q to some division of n, such as √n
3 create q threads, that are each responsible for summing n/q numbers
4 foreach thread ∈ threads do
5 |   add the result of threadAdd(list, q, i) to sum
6 return sum
```

Function threadAdd(list, q, i)

Input: n-length list of numbers, q elements of *list* to add, i thread ID
Output: sum of elements in $list[q * i, q * i + q]$ (q elements)

```
1 sum ← 0 as a variable for the sum of this thread's elements
2 for i ← q * i, i < q * i + q do
3 |   add list[i] to sum
4 return sum
```

   This will take a list of n elements, and break the list up into smaller subsections. Each process is responsible for linearly adding a chunk of the original list together. This is a common divide and conqure approach, where each process adds small subset of the original list together all at the same time. The linear operation on a subset of the original list takes $O(\log n)$ time, because the linear operation only operates on $\log(n)$ elements. Then, the `addElements` function exclusively writes "`sum += subsum`" $\log n$ times. This is a CREW-PRAM summation in $\log n$ time.

3. Write a PRAM algorithm using n^3 processors to multiply two $n \times n$ matrices. Your algorithm should run in $O(\log n)$ time.

   (a) *Proof.* We seek to prove that two $n \times n$ matrices can be multiplied together in $O(\log n)$ time using n^3 processors. Let A and B be $n \times n$ matrices. We want to calculate C, where $C = A \times B$ is another $n \times n$ matrix, in $O(\log n)$ time with n^3 processors.

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n} \\ A_{1,0} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{n,0} & \cdots & \cdots & A_{n,n} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & B_{0,1} & \cdots & B_{0,n} \\ B_{1,0} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ B_{n,0} & \cdots & \cdots & B_{n,n} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} & \cdots & C_{0,n} \\ C_{1,0} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ C_{n,0} & \cdots & \cdots & C_{n,n} \end{bmatrix}$$

   Let $i, j \in \mathbb{Z}$ be nonnegative integers such that $0 \leq i \leq n$ and $0 \leq j \leq n$. The $(i,j)^{\text{th}}$ element of matrix C can be calculated by the following equation:

$$C_{i,j} = \sum_{k=0}^{n} A_{i,k} \cdot B_{k,j} \tag{1}$$

   This equation shows that the calculation for $C_{i,j}$ is a summation of n elements in a "list". Each element in the list is one element of the resultant vector from multiplying the i^{th} column vector of matrix A by the j^{th} row vector of matrix B. We have shown in Problem 2 above that the *addElements* function is capable of summing an n-element list of numbers with n processors in $O(\log n)$ time. Thus, we can compute the $(i,j)^{\text{th}}$ element of C using n processors in $O(\log n)$ time. Calculating every element in C requires n^2 calculations of $C_{i,j}$ for all $0 \leq i \leq n$, $0 \leq j \leq n$. By having one processor calculate a given $C_{i,j}$ using n threads in $O(\log n)$ time, as shown above, we can create n^2 processors to calculate each $C_{i,j}$ in parallel. Each one of the n^2 processors uses n threads to calculate $C_{i,j}$ in $O(\log n)$ time. Therefore, we have shown that with n^2 processors that each use n threads to perform the matrix multiplication, we can use n^3 processors to multiply two $n \times n$ matrices in $O(\log n)$ time using a PRAM algorithm. □

Function matrixMul(M_1, M_2)

Input: two distinct $n \times n$ matricies of real numbers
Output: The resulting matrix S of multiplying M_1 and M_2

```
1  S ← [n][n]
2  create n^2 threads, that are each assigned a unique location (i, j) in S and a corresponding location in M_1
3
4  // generate solutions in a single, constant, action
5  foreach thread ∈ threads do
6  |  productList ← [n]
7  |  create a new set of n threads, each assigned a unique location x in productList with corresponding location in
   |    M_2(i, x)
8  |
9  |  // generate n products in constant time
10 |  foreach thread ∈ newThreads do
11 |  |  productList[x] ← M_1(i, j) * M_2(i, x)
12 |
13 |  release the threads in newThreads
14 |
15 |  // use addElements with the productList to get each result in O(log n) time
16 |  S(i, j) ← addElements(productList)
17 |
18 return S
```