1 Types of Parallelism

Parallelism, performing two or more actions simultaneously, exists at many levels within modern computer systems. All of the following can be considered to be a type of parallelism.

- Multiple logic gates
- Processor pipelining
- Vector instructions
- Hyper-threading
- Multicore processors and multiprocessor systems
- Distributed or cloud computing

This class will primarily develop the theory of multicore and multiprocessor processing with shared memory.

As a prerequisite to this class, students should have experience analyzing sequential algorithms (e.g. CSE241). We build on this experience by examining two key differences between the sequential and parallel analyses: the computational model and how we will evaluate parallel algorithms. Lastly, we demonstrate how to parallelize both matrix multiplication and merge sort.

2 Computational Model

Any algorithm must be analyzed in the context of the machine on which it executes. We can conceptualize this machine as a set of assumptions, which we then call a computational model. For example, the Random Access Machine (RAM) is the most commonly used model to analyze sequential algorithms, and the primary assumption is that all memory takes the same length of time to access. Note that this assumption is not true for modern processors, but it is close enough for a general analysis.

We will begin this class by studying the Parallel-RAM model (PRAM, pronounced P-RAM). In this model a machine may have an arbitrary number of processors, even infinite. Like the RAM model, every processor can access every memory location in the same constant amount of time as any other processor and memory location. PRAM is a very simple model of parallel computing, yet powerful and applicable. However, it does lack features like cache that can dramatically influence parallel performance in the real world.

There are three flavors of PRAM that determine how memory is shared between processors.
• EREW: exclusive read, exclusive write
• CREW: concurrent read, exclusive write
• CRCW: concurrent read, concurrent write

Exclusivity means that a particular memory location can only be accessed by a single processor per execution cycle, while concurrent has no limitation. The CREW flavor is the one we will work with most commonly, where any number of processors may read from a memory location concurrently, but only one processor may write to a memory location at a time.

The CRCW model can be thought of as weird because there is no intuitive way to resolve multiple processors writing to the same memory location at the same time. As such, any CRCW model must specify some behavior for when this occurs. Such resolution mechanisms could be, for example, to assign processors different priorities, to randomly accept some write, or to perform a bitwise AND between multiple writes.

3 Measuring Parallelism

The goal of analyzing any algorithm is to determine how “good” it is. In a sequential analysis we try to put an upper-bound on the total execution time of an algorithm by bounding the total amount of work that is to be done. This is insufficient, however, for parallel analysis; in general, a parallel algorithm might have more total work to perform, but have a shorter total running time due to parallelism.

Thus, rather than merely analyzing the total work performed we are also interested in two other quantities:

**Definition 1.** The work of an algorithm, denoted \( T_1 \), is the total running time on one processor.

**Definition 2.** The span of an algorithm, denoted \( T_\infty \), is the total running time on an infinite number of processors.

Note that the span may also be called the critical path length, critical length, or depth. As they are defined above, the span represents the longest sequential chain of instructions through a parallel algorithm, while the work represents all instructions without exception. Individually these are useful indicators, but separately they don’t give us any idea about how well an algorithm will actually perform on a real system.
MATMULT(A,B,C,n)
1 for i = 1 to n
2 do
3 for j = 1 to n
4 do
5 for k = 1 to n
6 do
7 \[ C[i, j] += A[i, k]B[k, j] \]

Figure 3: Sequential Matrix Multiplication Algorithm

Definition 3. The parallelism of an algorithm is defined to be \( \frac{T_1}{T_\infty} \).

We will use the quantity \( \frac{T_1}{T_\infty} \) as the primary indicator of algorithm “goodness”, but not the only. It is possible to arbitrarily increase the amount of work an algorithm must perform which will in turn result in an arbitrarily large measure of parallelism, but such an algorithm is certainly not a “good” algorithm. Rather, to make parallel algorithms more efficient we will primarily focus on shortening the critical path length. At times it is possible to shorten the critical path length at the expense of introducing more work, which is a valid way to yield more parallelism, but care must be taken that the resulting algorithm will actually perform better than the existing solution.

4 Matrix Multiplication

We will apply these ideas by attempting to parallelize the classical matrix multiplication algorithm under the PRAM with CREW model. We formally state the problem as: given two \( n \) by \( n \) matrices \( A \) and \( B \), how can we efficiently compute \( AB = C \)?

Recall the matrix multiplication formula, the \([i, j]^{th}\) cell of \( C \) may be computed with

\[
C[i, j] = \sum_{k=1}^{n} A[i, k]B[k, j]
\]  

(1)

We can then translate this to a sequential program as in figure 3.

The equation above is a closed-form solution for each individual cell in the resultant matrix \( C \). Every part of the solution is independent from every other part of the solution, so we can compute the value of every cell in parallel. To accomplish this we can use a parallel-for loop, which specifies that different iterations of the loop are allowed to execute in parallel. This is shown in figure 4. Note that we cannot parallelize the innermost loop because of our decision to use the CREW PRAM model- the exclusive write condition implies that we cannot perform more than one update of a resultant cell per unit time.

The work for both MATMULT and PARAMATMULT is clearly \( O(n^3) \). Recall that the span is the longest sequential chain of instructions in the algorithm: in this algorithm every cell takes exactly \( O(n) \) to compute and every cell can be computed in parallel, so the span must be \( O(n) \). Thus, we can compute: \( T_1 = O(n^3) \), \( T_\infty = O(n) \), so the parallelism \( \frac{T_1}{T_\infty} \) is \( O(n^2) \).

How do we interpret this parallelism? For a given problem of size \( n \) we could theoretically achieve a parallel speedup of \( n^2 \) times over a sequential execution.
PARAMATMULT(A,B,C,n)
1 para-for i = 1 to n
2 do
3 para-for j = 1 to n
4 do
5 for k = 1 to n
6 do
7 \[ C[i,j] = A[i,k]B[k,j] \]

Figure 4: Parallel Matrix Multiplication Algorithm

There are, however, practical considerations: if we were multiplying two 100x100 matrices we could use parallelism to perform that computation up to 10,000 times faster, but in order to do so we would also need 10,000 processors. Also, there are many technical issues that effect parallelism in real computer systems which we do not account for in the PRAM model. Thus, we rarely achieve the theoretical speedup predicted by this analysis in real life.

5 Merge Sort

We now turn our attention to a problem that requires a more sophisticated technical analysis. Formulated loosely, the goal of any sorting method is to permute an array \( A \) of numbers so that the value of each element is greater than or equal to the previous element. Merge sort accomplishes this through recursively splitting \( A \) into two evenly sized pieces and sorting those subproblems independently. When both subproblems are sorted they are then recombined (or merged) back into a sorted array.

For example, consider the array of integers \( B = 7-9-3-5-6-1-2-0 \). The action of merge sort upon \( B \) is illustrated in figure 5, and sequential pseudocode is given in figure 6. The most straightforward way to parallelize this algorithm is to perform each recursive call to \( \text{MERGESORT} \) concurrently, as in figure 7.

The work, which we will call \( T_{MS1} \), of the parallel algorithm in figure 7 is given by the recurrence \( T_{MS1}(n) = 2T_{MS1}(\frac{n}{2}) + T_{M1}(n) \) where \( T_{M1}(n) \) is the work of the \( MERGE \) method. Simple inspection shows that \( T_{M1}(n) = O(n) \), and then the Master theorem gives us \( T_{MS1}(n) = O(n \log n) \). To calculate the critical path length \( T_{MS\infty} \) we observe that the parallel block in figure 7 has critical path length \( T_{MS\infty}(\frac{n}{2}) \) because both calls to \( \text{MERGESORT} \) execute in parallel. Additionally, it is again easy to see by inspection that \( T_{M\infty}(n) = O(n) \) for the \( MERGE \) procedure. Hence, we derive the recurrence \( T_{MS\infty}(n) = T_{MS\infty}(\frac{n}{2}) + T_{M\infty}(n) \), which is again solvable by the Master method to be \( O(n) \). This results in a total parallelism of only \( O(\log n) \).

In general, we would like to achieve at least a polynomial factor (e.g. \( O(n) \) or more) of parallelism. Recall that in order to improve parallelism we first want to look at shortening the critical path length of the whole algorithm. In the recurrence for \( T_{MS\infty} \) we see that the \( MERGE \) process dominates with \( O(n) \) time, so we first look there. Currently \( MERGE \) is an entirely sequential process, so any parallelism will yield an improvement in critical path length.

We first formalize the problem. Consider two sorted arrays \( A_1 \) and \( A_2 \) of potentially different sizes, but assume that \( A_1 \) is larger if they are. How can we efficiently merge these two arrays? First, let \( x \) be the middle element of
Recursively divide
Recursively divide
Recursively divide
Merge sorted elements together
Merge sorted elements together
Merge sorted elements together

Figure 5: Merge sorting an array with a base case size of one.

MERGESORT(A,n)
1 if n = 1
2 then
3 return A
4 else
5 MERGESORT(A[1...\(\frac{n}{2}\)], \(\frac{n}{2}\))
6 MERGESORT(A[\(\frac{n}{2}+1\)...n], n)
7 MERGE(A_1, A_2)

Figure 6: Sequential Merge Sort
MERGESORT(A,n)
1     if n = 1
2          then
3             return A
4     else
5         In Parallel
6             MERGESORT(A[1.. \frac{n}{2}], \frac{n}{2})
7             MERGESORT(A[\frac{n}{2} + 1.. n], n)
8         End Parallel
9         MERGE(A_1, A_2)

Figure 7: Parallel Merge Sort

A_1. Perform a binary search for x in A_2. Now, everything to the left of x in both A_1 and A_2 is less than x, and everything to the right in both arrays is greater than x. Thus, we can merge the left segment of A_1 with the left segment of A_2 without considering the right segments, and we can similarly merge the right segments without considering the left segments. Thus, both parts can be merged in parallel, and then the results combined back together in constant time. This process is depicted in figure 8.

Now we must analyze this new approach to ensure that we have actually decreased the critical path length. Let a and c be the number of elements in A_1 and A_2 which are less than the middle element x, respectively. Let b and d be the number of elements in A_1 and A_2 which are greater than the middle element x.

The critical path length is, again, harder to determine. Each parallel merge can be done concurrently, but we don’t know explicitly how many elements are merged in each call, so we must state the recurrence as

\[ T_{\infty}^{PM}(n) = \text{MAX}(T_{\infty}^{PM}(a + c), T_{\infty}^{PM}(b + d)) + T_{\infty}^{BS}(n) \]  

(2)

By construction, \( a + b + c + d = n \), \( a + b = \frac{n}{2} \), and \( a = b \). Thus, \( a = b = \frac{n}{4} \). Therefore, we can conclude both \( a + c < \frac{3n}{4} \) and \( b + d < \frac{3n}{4} \).

Now, we can restate the critical path recurrence with this upper bound: \( T_{\infty}^{PM}(n) = T_{\infty}^{PM}(\frac{3n}{4}) + T_{\infty}^{BS}(n) \). The critical path through the binary search is \( O(\lg n) \), so using the Master method we can solve to get \( T_{\infty}^{MS}(n) = O(\lg^2 n) \). Thus, the critical path for the merge sort with the parallel merge is then

\[ T_{\infty}^{MS}(n) = T_{\infty}^{MS}(\frac{n}{2}) + T_{\infty}^{PM}(n) \]  

(3)

\[ T_{\infty}^{MS}(n) = O(\lg^3 n) \]  

(4)

Hence, the overall parallelism is:

\[ \frac{T_{\infty}^{MS}}{T_{\infty}^{MS}} = \frac{O(n \lg n)}{O(\lg^3 n)} = O\left(\frac{n}{\lg^2 n}\right) \]  

(5)

So we have successfully achieved a polynomial-factor parallelism.
Step 1: Find X in both arrays

Everything in A and C
is less than X

A and C

B and D

Everything in B and D
is more than X

Step 2: Merge A with C and B with D in parallel

Because both arrays started about sorted, the result of step 2 is now merged and sorted. Because the merge is parallel, the critical path length is $O(3n/4)$.

Figure 8: How to merge two sorted arrays in parallel.