Course: Comtemporary Algorithms T.II/2019-20

Lecture 7: Multicore Parallel Techniques I

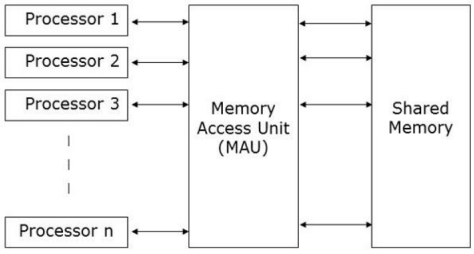
27 January 2020

Lecturer: Dr. Kanat T. *Scribe: Pitipat C. & Nuttapat K.*

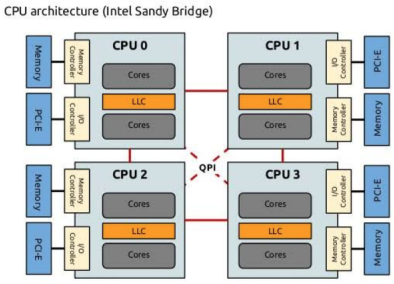
As the number of cores per CPU and the number of transistors on integrated circuit chip increase, parallel programming paradigm is becoming more important since it increase computing performance and resource utilization

1 Computer Architecture POV

Every CPU in a system share the same memory, facilitated by Memory Access Unit (MAU). However, memory access for each CPU core is not uniform because a memory could be located close to the cores or it could be located further away from the cores, which results in higher latency of memory access.



PRAM Abstraction



Recent(?) Intel Architecture

2 Nested Parallelism

Nested parallel computations can be represented by dependency graphs (DAG). Two tasks are parallel if they are not reachable from each other.

- no synchronization among parallel tasks except at joint points
- Good schedulers are known
- Easy to understand, debug, and analyze

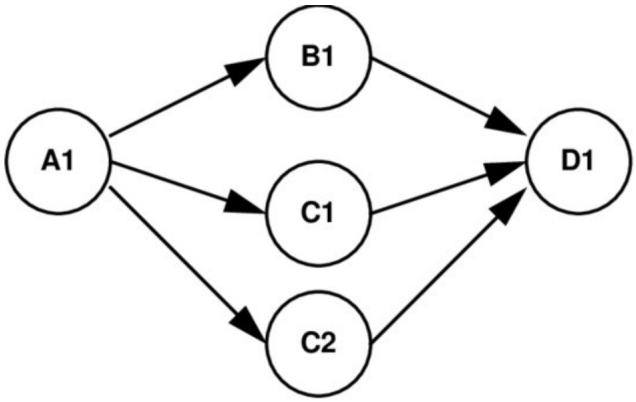


Figure 1: Dependency Graph

2.1 Parallel For-Loop (pfor)

Do these iterations in parallel. Usually accomplished by splitting the task into small pieces and running them simultaneously

```
cilk_for (int i=0; i < n; i++)    // Cilk (C++)
    B[i] = A[i]+1;


#pragma omp for
for (int i=0; i < n; i++)    // OpenMP (annotation on C/C++)
    B[i] = A[i] + 1;
```

Figure 2: Example of Parallel For-Loop in C++

2.2 Fork-Join $(A||B)$

A way of setting up and executing parallel programs by "fork" execution branches off in parallel, then "join" to resume sequential execution. Parallel sections may fork recursively.

```
int fib(int n) {
  if (n < 2) return n;
  int x = cilk_spawn fib(n-1);
  int y = fib(n-2);
  cilk_sync;
  return x + y;
}
```

Figure 3: Example of Fork-Join in C++

2.3 Example in Pseudo-code

```
parfor i in [0:|A|]
  B[i] := f(A[i]);
```

```
sum(A) =
  if (|A| == 1) then return A[0];
  else
    l = sum(A[ : |A|/2]) ||
    r = sum(A[|A|/2 : ]);
    return l + r;
```

3 Work/Depth Cost Model

Work(W)

$W(n) :=$ total number of operations (costs are added across parallel calls)

Span(S) or Depth(D)

$S(n) :=$ depth/critical path of the computation (representing the longest chain of dependencies)

Parallelism

$$\mathbb{P} = W/S$$

3.1 Parallel For-Loop

```
pfor i in range(n):
    f(i)
```

Let $W_f(i)$ be work of $f(i)$ and $S_f(i)$ be span of $f(i)$

$$W_{\text{total}} = \sum_{i}^{n} W_f(i)$$
$$S_{\text{total}} = max_i S_f(i)$$

3.2 Parallel Fork-Join

```
x,y = f(...) || g(...)
```

Let W_f be work of f, S_f be span of f, W_g be work of g, and S_g be span of g

$$W_{\text{total}} = W_f + W_g$$
$$S_{\text{total}} = max(S_f, S_g)$$

3.3 Brent's Theorem

Define

$$T_p = \text{time on } p \text{ processors}$$

Let W be total number of operations, P be number of processors, and S be sequential time steps

$$T_p \leq \frac{W - S}{P} + S$$

in which

$$T_p \geq \frac{W - S}{P}$$

and

$$T_p \geq S$$

3.4 Goals

1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is **work efficient**.
2. Parallelism (W/S) should be polynomial. $O(\sqrt{n})$ is probably good enough

4 Example: Sum

Consider this implementation of sum

```
def sum(xs):
    total = 0
    for i in xs:
        total += i
    return total
```

$$W(n) = O(n)$$
$$S(n) = O(n)$$

If we change to use **pfor**, then

```
def sum(xs):
    total = 0
    pfor i in xs:
        total += i
    return total
```

$$W(n) = O(n)$$
$$S(n) = O(1)$$

However, we might encounter race condition, so discard this implementation
Recall the divide an conquer technique. If we halve the problem and recursively solve smaller problems in parallel (using fork-join).

```
def sum_dq(xs):
    if len(xs) <= 1:
        return xs
    n = len(xs)
    l, r = sum_dq(xs[:n/2]) || sum_dq(xs[n/2:]
    return l+r
```

Assume that array slicing runs in $O(1)$ time by using pointers. Hence,

$$W(n) = 2W(n/2) + O(1) = O(n)$$
$$S(n) = S(n/2) + O(1) = O(logn)$$

5 QuickSort

We will now see how can we make our beloved QuickSort parallel. Consider the following code :

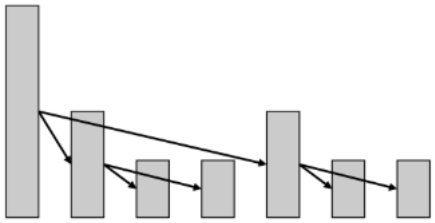
```
def qs(xs: Seq[Int]) -> Seq[Int]:
  if (len(xs) <= 1): return xs
  else:
    p = RNG.choice(xs)
    s0 = [ e for e in xs if e < p ]
    s1 = [ e for e in xs if e == p ]
    s2 = [ e for e in xs if e > p ]
    (r0, r2) = par(qs(s0) || qs(s2))
    return r0 + s1 + r2
```

Lets analyze the time complexity of the parallel quick sort. By partition sequentially and appending in parallel, we then have work of $\boldsymbol{O}(n \log n)$ and span of $\boldsymbol{O}(n)$. Thus the parallelism, p, is $\frac{W}{S} = \boldsymbol{O}(\log n)$ which is not a very good parallelism.

How can we have a better parallelism ?.

5.1 Thought Experiment 1

Assume: Partitioning and concatenation can be done in $\boldsymbol{O}(n)$ work and $\boldsymbol{O}(\log n)$ span but recursive calls are made sequentially.

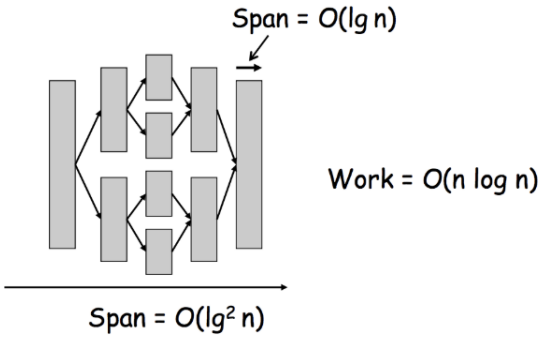


What is the complexity of such assumption ? We have the work of $\boldsymbol{O}(n \log n)$ and span of $\boldsymbol{O}(n)$ therefore we get $\boldsymbol{O}(\log n)$ parallelism which is still not good enough.

5.2 Thought Experiment 2

Assume: Partitioning and concatenation can be done in $\boldsymbol{O}(n)$ work and $\boldsymbol{O}(logn)$ depth **and** recursive calls are made in parallel.

Since for each concatenation, we have a span of $\boldsymbol{O}(\log n)$ hence overall span is of $\boldsymbol{O}(\log^2 n)$. The work of such assumption is $\boldsymbol{O}(n \log n)$ therefore we have the parallelism of $\boldsymbol{O}(n/\log n)$ which we consider a good parallel algorithm.



6 Parallel Techniques

Recall the following operations on collections :

1. map: applies a function f to every element of the collection
2. filter : keep the element if element's predicate is true
3. reduce: pairwise combines elements in a tree until you have 1 using a (n associative) binary operator
4. etc

Associative Binary Operator

$$\star : \boldsymbol{U} \times \boldsymbol{U} \mapsto \boldsymbol{U}$$

eg. plus, multiply operator.

The operator is said to be associative if $\forall a, b, c \in \boldsymbol{U}, (a \star b) \star c = a \star (b \star c)$

6.1 Map

Consider the following parallel map algorithm:

```
def par_map(xs: List[int], f: Callable) -> List[int]:
  out = []
  pfor i in range(n):
    out[i] = f(xs[i])
  return out
```

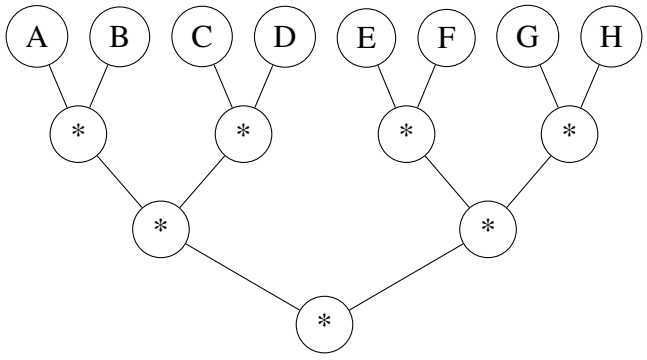
Hence we have $\boldsymbol{O}(n)$ work and $\boldsymbol{O}(1)$ span since we can apply f to each element in parallel. Thus we have parallelism of $\boldsymbol{O}(n)$

6.2 Reduce

Define a reduce using associative binary operator above as follow:

$$reduce(\star, xs) \mapsto x_1 \star x_2 \cdots \star x_n$$

thus we can do $\star$ on all pairs first and then keep going down



(1)

Thus the number of elements each round are reduced by half. Hence we need to do $\boldsymbol{O}(\log n)$ round which make total work done be $\boldsymbol{O}(n)$ and $\boldsymbol{O}(\log n)$ span

6.3 Filter

A filter takes input of collection A and a predicate p and return the collection containing $a \in A$ such that $p(a)$ is true in the same order as in A. Consider the following example:

Given a list $A = [2, 1, 4, 0, 3, 1, 5, 7]$ with predicate $p(a) : a < 4$. We will perform the following step:

Step 1: Compute the array of flag F, $F[i] = P(A[i])$
Hence we have $[1, 1, 0, 1, 1, 1, 0, 0]$

Step 2: Apply plusScan to the flag array F and maps to the indexes array, I.
Hence we have $[0, 1, 2, 2, 3, 4, 5, 5]$

Step 3: Now we allocate result array, R where $R[i] = X[I[i]]$.
Thus we have the final result $[2, 1, 0, 3, 1]$

Thus we have a total work for filter of $\boldsymbol{O}(n)$ and span of $\boldsymbol{O}(\log n)$ from plusScan

6.4 Flatten

The flatten takes a nested sequence A as input and return a flatten sequence R that contains each sequences in A concatenate together.
For example, flatten ([[3,2], [2,3,4],[5],[1,2,7,9]]) = [3,2,2,3,4,5,1,2,7,9]