Activity: Redesigning for parallelism

1 Reduce

Consider the reduce function:

```
template<typename T, typename op>
T reduce (T* array, size_t n) {
  T result = array[0];
  for (int i=1; i<n; ++i)
    result = op (result, array[i]);
  return result;
}
```

Do not be scared by the syntax! In C++, templates allow you to replace types and values in a piece of code by a type or a value known at compilation time. This is similar to generics in Java.

So if you define `T` as `int` and `op` as `sum`, it boils down to computing the sum of the array. You could use `op` as `max` and compute the maximum value of the array.

1.1 int, sum

Consider first the `int, sum` case which computes the sum of an array of integers.

We have already shown that there is no parallelism in the parallel task graph extracted from the algorithm as it is written.

Question: Assuming you have P processors, rewrite the code to introduce one local variable per processor to store partial computation so as to achieve more parallelism. What is the width, critical path and work ?

Question: What does a schedule look like on P processors?

1.2 Variants

(Note that by correct we mean that the parallel version would produce the exact same result as the sequential version in all cases.)

Question: Would that parallel version be correct for `int, max`? Why?

Question: Would that parallel version be correct for `string, concat`? Why?

Question: Would that parallel version be correct for `float, sum`? Why?

Question: Would that parallel version be correct for `float, max`? Why?

```cpp
/*
1) Reduce
1.1)
*/
template<typename T, typename op>
T reduce (T* array, size_t n){
    float frac = n/P;
    int num_in_each_process = std::ceil(frac);
    T results[P];
    for(int i=0;i<P;i++){
        int start_index = i * snum_in_each_process;
        results[i] = array[start_index];
        for(int j=1;j<num_in_each_process;j++){
            if(start_index + j < n){
                results[i] = op(results[i], array[start_index + j]);
            }
        }
    }
    T result = results[0];
    for(int i=1;i<P;i++){
        result = op(result,results[i]);
    }
}
/*
width: P
CP:    3 + n/P + P
work:  3 + n + P
1.2)
int, max:       yes, because each block would have a local maximum, then the global would be found from those values
string, concat: yes, because each block is still working in sequential order within it's contained process. When 'conquering', they are also still in order
float, sum:     yes, because of the communitive proproty of addition: (A+B)+C == A+(B+C)
float, max:     yes, because each block would have a local maximum, then the global would be found from those values
```

2 Prefix sum

Prefixsum is an algorithm that has many uses in parallel computing. The algorithm computes $pr[i] = \sum_{j<i} arr[j], \forall 0 \leq i \leq n$ and is often written sequentially:

```
void prefixsum (int* arr, int n, int* pr) {
  pr[0] = 0;
  for (int i=0; i<n; ++i)
    pr[i+1] = pr[i] + arr[i];
}
```

We have seen that this writing of prefix sum generates a parallel task graph that is one long chain.
Question: Rewrite this algorithm to make it parallel on P processors. (Hint: What goes wrong if you were blindly parallelising the code by cutting the work in say 3 chunks? You may have to add some work without changing the complexity in Big-Oh notation. A single pass on the array is not enough.)
Question: What is the work, width, and critical path of the algorithm you created?

```
2)Prefix Sum
*/
void prefixsum (int* arr , int n, int* pr) {
    float frac = n/P;
    int num_in_each_process = std::ceil(frac);
    for(int i=0;i<P;i++){
        int start_index = i * num_in_each_process;
        pr[start_index] = 0;
        for(int j=0;j<num_in_each_process;j++){
            if(start_index + j + 1 < n){
                pr[start_index + j + 1] = pr[start_index + j] + arr[start_index + j];
            }
        }
    }
    for(int i=1;i<P;i++){
        int start_index = i * num_in_each_process;
        int diff = pr[start_index - 1];
        for(int j=0;j<num_in_each_process;j++){
            if(start_index + j + 1 < n){
                pr[start_index + j] += diff;
            }
        }
    }
}
/*
width: P
CP:    2 + n/P + Pn
work:  3 + 2n
```

3 Merge Sort

There is limited parallelism in merge sort because the merge operation is naturally sequential.
Question: Rewrite merge to make it suitable to execute with P processors. (Hint: it is one of these cases where you may have to increase complexity slightly.)
Question: What are the work and critical path of the merge algorithm you created?
Question: If you used that parallel merge in merge sort, what would the work and critical path of merge sort become?

```
3)Merge Sort
*/
T[] merge(T* a, T* b){
    float frac = n/P;
    int num_in_each_process = std::ceil(frac);
    T merged[P];
    for(int i=0;i<P;i++){
        int start_index = i * num_in_each_process;
        T[] a1 = a[start_index..start_index + num_in_each_process];
        T[] b1 = b[start_index..start_index + num_in_each_process];
        merged[i] = doMerge(a1,b1);
    }
    for(int i=0;i<P/2;i++){
        T retHold[std::ceil(merged/2)];
        for(int j=0;j<retHold.size();j++){
            T m1 = merged.at(j);
            T m2 = merged.at(merged.size()-j-1);
            if(m1 != m2){
                retHold[j] = doMerge(m1,m2);
            }
        }
        merged = retHold;
    }
    return merged;
}

T[] doMerge(T* a, T* b){
    int i = 0;
    int j = 0;
    int k = 0;
    T ret[2n];
    while(i < a.size() && j <b.size()){
        if(b[j] < a[i]){
            ret[k] = b[j];
            j++;
        }else{
            ret[k] = a[i];
            i++;
        }
        k++;
    }
    for(i;i<a.size();i++){
        ret[k] = a[i];
        k++;
    }
    for(j;j<b.size();j++){
        ret[k] = b[j];
        k++;
    }
}
/*
work: nPlog(nP)
CP:   nlog(nP)
*/
```