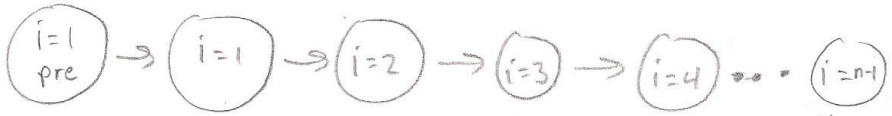
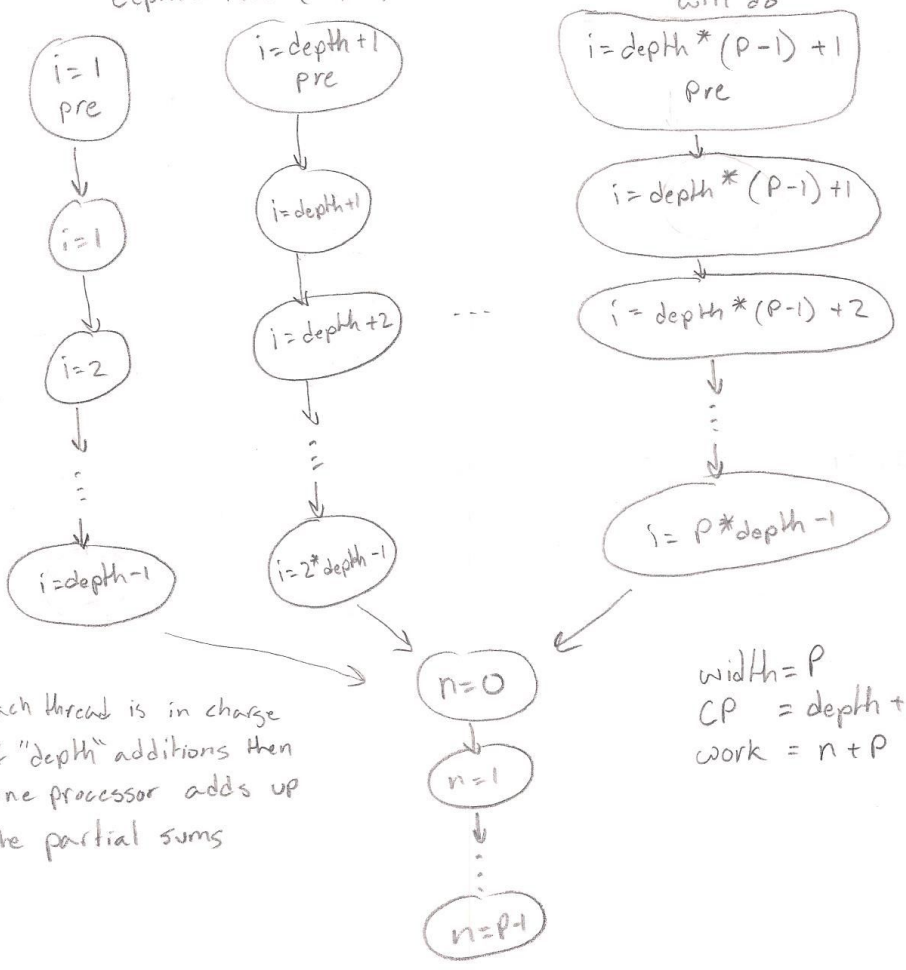
3.1.1)



All tasks are dependent on variable "result"

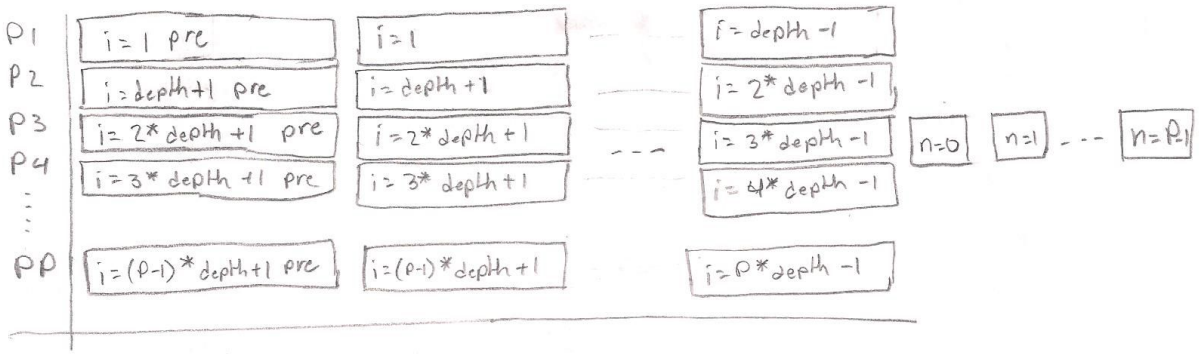
width = 1
CP = n
work = n

depth = floor(n/p) ← this is the number of additions each processor will do



each thread is in charge of "depth" additions then one processor adds up the partial sums

width = p
CP = depth + p
work = n + p



Each thread is incharge of "depth" number of sums. Then one thread (P3) is in charge of adding up the partial sums. The pre task foreach thread is the initial set of the result variable

3.2) reduce<int, max> - Yes. It would find the max element in each section of the array then compare the max for each section against each other to find the whole max

reduce<std::string, concact> - yes, the algo would concact each section then concat each section together, it would maintain proper order

reduce<float, sum> - yes, code is templated, and int is similar to float & would work the same way

reduce<float, max> - yes, similar to reduce<int, max>

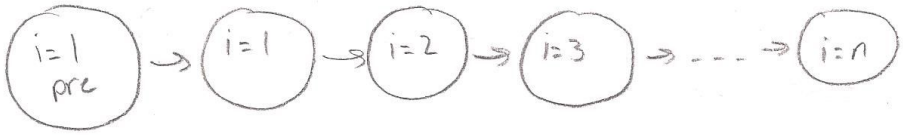
```cpp

// PLEASE COMPILE WITH g++ reduce_par.cpp -lpthread

#include <memory>
#include <iostream>
#include <vector>
#include <thread>
#include <cassert>
#include <string>
// #include <numeric>
// #include <execution>
#include <parallel/numeric>

template <typename T>
T sum(T a, T b) {
  return a + b;
}

template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}

// template <typename T, typename = std::enable_if_t<std::is_same_v<T, std::string>>>
template <typename T, typename = std::enable_if_t<std::is_same<T, std::string>::value>>
constexpr auto concact = sum<T>;

// Code from the assignment. The last parameter is needed
// because F is a type, even though it can be modified to only
// have 2 parameters, I belive this is easier on the head
template <typename T, typename F>
T reduce_sin(T* array, size_t n, F op) {
  T result = array[0];
  for (int i=1; i<n; ++i)
    result = op(result, array[i]);
  return result;
}

// I do not want to deal with promises or futures, so Im doing this.
/// Partially reduces the array from [start, end) and places it in
/// result.
/// @note The *result variable does not have a race condidtion as it
///       is only being written by one thread.
template <typename T, typename F>
void partial_reduce(T* start, T* end, T* result, F op) {
  *result = reduce_sin(start, std::distance(start, end), op);
}

/// Parallel reduce
template <typename T, typename F>
T reduce_par (T* array, size_t n, size_t p, F op) {
  // create nessicary structures
  std::unique_ptr<T[]> data{ new T[p] };
  std::vector<std::thread> threads{  };

  // calculate parallelism
  size_t depth = n / p;
  T* start = array;
```

```
  T* end = array + depth;

  // the actual parallel code
  for (size_t i = 0; i < p - 1; ++i) {
    // The <T, F> is needed because partial_reduce is a templated function so we need
    // to thread the <T, F> version of the function
    threads.emplace_back(partial_reduce<T, F>, start, end, &data[i], op);
    start = end;
    end += depth;
  }

  // the last one (this thread will do it so we dont spin on waiting for them to join)
  partial_reduce(start, array + n, &data[p - 1], op);

  // wait for them to finish
  for (auto& t : threads)
    t.join();

  // final sum
  return reduce_sin(data.get(), p, op);
}


// options:
// 1: reduce<int, sum>
// 2: reduce<int, max>
// 3: reduce<std::string, concact>
// 4: reduce<float, sum>
// 5: reduce<float, max>
#define OPTION 1
#define MAX 500'000'007
// @warning, the only one that doesnt really work very well is option 4. Adding up
// more than 1mil values gives you two different answers and I'm mostly certains its
// because of FP arithmetic


#if OPTION == 1
  #define NUMERIC
  using   NUM_TYPE = int;
  template <typename T> constexpr auto op_t = sum<T>;
#elif OPTION == 2
  #define NUMERIC
  using   NUM_TYPE = int;
  template <typename T> constexpr auto op_t = max<T>;
#elif OPTION == 3
  template <typename T> constexpr auto op_t = concact<T>;
#elif OPTION == 4
  #define NUMERIC
  using   NUM_TYPE = float;
  template <typename T> constexpr auto op_t = sum<T>;
#elif OPTION == 5
  #define NUMERIC
  using   NUM_TYPE = float;
  template <typename T> constexpr auto op_t = max<T>;
#else
  #error "[E] Valid option not chosen"
```

```
115 #endif
116
117
118 int main() {
119
120 #ifdef NUMERIC
121
122   constexpr unsigned long long LG_NUM = MAX;
123   std::vector<NUM_TYPE> data{};
124   data.push_back(3); // this is a "valid" dummy data for max
125
126   for (unsigned long long i = 1; i < LG_NUM + 1; ++i) {
127     data.push_back(1); // sums of ones
128   }
129
130 #else
131
132   char temp[2] = { '\0', '\0' };
133   std::vector<std::string> data{};
134   for (int j = 0; j < 3; ++j) {
135     for (unsigned long long i = 0; i < 26; ++i) {
136       // 97-122
137       temp[0] = i + 97;
138       data.emplace_back(temp);
139     }
140   }
141
142 #endif
143
144   auto op = op_t<decltype(data)::value_type>;
145
146   auto begin1  = std::chrono::high_resolution_clock::now(); //// AHHHHHHH
147   auto accum1  = reduce_sin(data.data(), data.size(), op);
148   auto end1    = std::chrono::high_resolution_clock::now(); //// AHHHHHHH
149
150   std::chrono::duration<double, std::milli> elapse1{ end1 - begin1 };
151   std::cout << "Serial: " << elapse1.count() << std::endl;
152
153   auto begin2  = std::chrono::high_resolution_clock::now(); //// AHHHHHHH
154   auto accum2  = reduce_par(data.data(), data.size(), 8, op);
155   auto end2    = std::chrono::high_resolution_clock::now(); //// AHHHHHHH
156
157   std::chrono::duration<double, std::milli> elapse2{ end2 - begin2 };
158   std::cout << "Parallel: " << elapse2.count() << std::endl;
159
160   // Turns out that neither gnu g++ or clang++ supports the parallelism TS,
kms
161   // https://youtu.be/Mcjrc2uxbKI?t=571 MSVC ahead of the game *slow claps*
162   // nvm: https://godbolt.org/z/TCGaze The internet video lies
163   // auto begin3 = std::chrono::high_resolution_clock::now();
164   // double result = std::reduce(std::execution::par, sum.begin(),
sum.end());
165   // auto end3 = std::chrono::high_resolution_clock::now();
166
167   // std::chrono::duration<double, std::milli> elapse3{ end3 - begin3 };
168   // std::cout << "STL: " << elapse3.count();
169
170   // hehe https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html
171   // after seeing the results, another lie. Why does the docs lie?????
172   // Serial: 1512.65
```

```
173   // Parallel: 351.756
174   // GNU: 1562.03
175   // The sum is: 500000010
176 // #if OPTION == 1
177 //  auto begin4 = std::chrono::high_resolution_clock::now();
178 //  double accum4 = __gnu_parallel::accumulate(data.begin(), data.end(), 0);
179 //  auto end4 = std::chrono::high_resolution_clock::now();
180 
181 //  std::chrono::duration<double, std::milli> elapse4{ end4 - begin4 };
182 //  std::cout << "GNU: " << elapse4.count() << std::endl;
183 // #endif
184 
185   if (accum1 == accum2)
186     std::cout << "The result is: " << accum1 << std::endl;
187   else
188     std::cout << "The result were different, you screwed up..." << std::endl
    << accum1 << " " << accum2 << std::endl;
189 }
```

4)



all tasks are dependent on the i-1 iteration. This is because to calculate pr[i], you need pr[i-1]

width = 1
cp = n+1
work = n+1

Parallel version

Similar to the reduce function, the array will be split into smaller chunks and prefix sum will be computed then the last sum of each chunck is added into the next chunk

Pass 1

ar	pr
0	0
1	0
2	1
3	3
4	6
5	10
	15

thread 1
offset = 15

ar	pr
6	0
7	6
8	13
9	21
10	30
11	40
	51

thread 2
offset = 51

ar	pr
12	0
13	12
14	25
15	39
16	54
17	70
	87

thread 3
offset = 87

do partial prefix sum for each chunck of the data in this case 3 chunks. The offset is the largest partial sum of that chunk. This is step 1

Pass 2

error = 0	
pr	fin
0	0
0	0
1	1
3	3
6	6
10	10
15	15

error = 15	
pr	fin
0	15
6	21
13	28
21	36
30	45
40	55
51	66

error = 51 + 15 = 66	
pr	fin
0	66
12	78
25	91
39	105
54	120
70	136
87	153

Now add all the offsets from the previous chunks to each member of the partial sum the sum of the offsets is the error. This sumation is independent and can be threaded similar to step 1. This is step 2 and final step

5) mergesort super highlevel pseudocode

```
mergesort (arr)
    if len(arr) ≤ 1:
        return arr

    left = left half of arr
    right = right half of arr

    left = mergesort (left)
    right = mergesort (right)

    return merge (left, right)
```

an array of size 1 is sorted

split into 2 parts

sort both left + right parts recursively

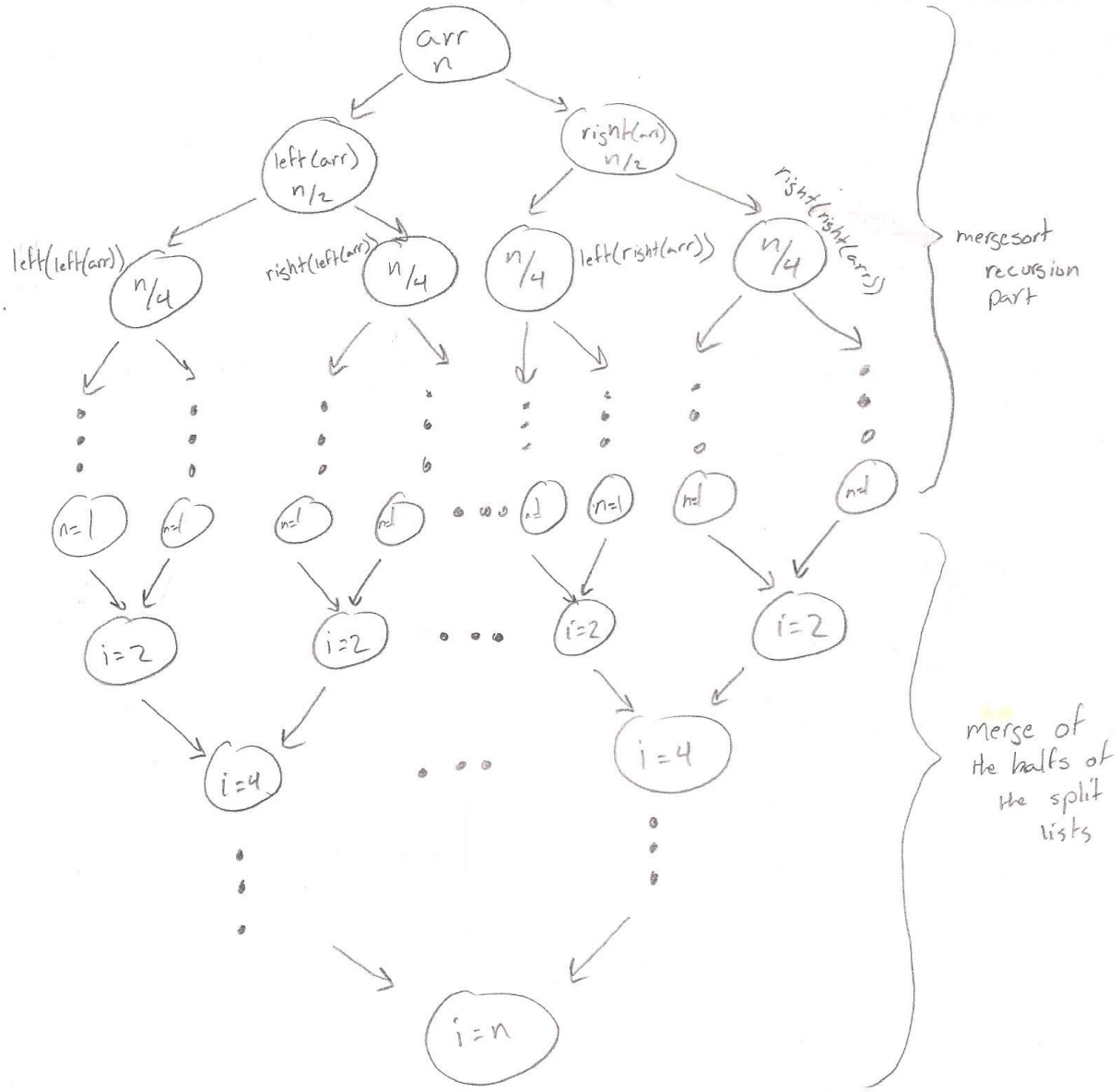
merge the two parts in order

```
merge (left, right)
    result = empty array

    while left is not empty
          and right is not empty:
        if left[0] ≤ right[0]:
            result += left[0]
            rem left[0]

        else:
            result += right[0]
            rem right[0]

    if left is not empty
        result += left

    if right is not empty
        result += right

    return result
```

create empty result array

go through both left + right sides and place the saller of the first element into the result array

place the rest of the elements from either list into the result array

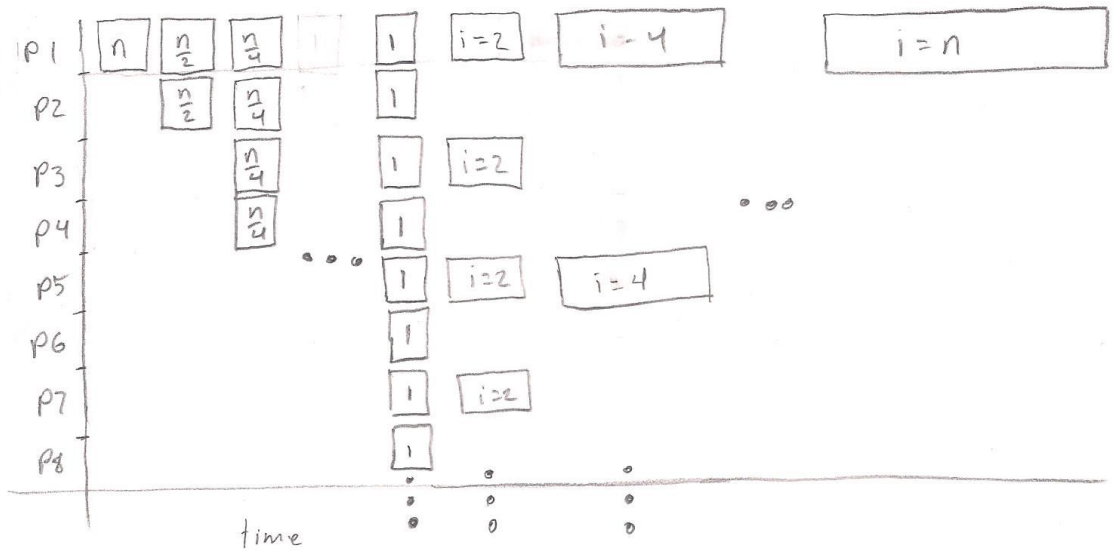


The first half of the dependency tree is the recursive part of mergesort() function it splits the array into 2 subarrays. Then these subarrays are sorted and merged in the second half of the dependency graph. The merge part of the dependency graph takes more time to compute

work = $3n-2$

CP = $2\log_2(n)+1$

width = 1



Diamond shape? If we dont have any penalty for spawning threads, then each recursive call would spawn 2 threads: one for each half of its current thread. There is very little "work" happening other than the creation of the threads. That is why the bars here are thin. when the merger of two subarrays happen, one thread will handle it but because it has to go through 2 sub arrays, the time taken to sort will double as the recursion stack collapses to the end condition 'i=n'

To make the code more parallel, I would remove some of the initial recursion so there is little dependency at the biginning of the run. For example with a P of 4, the fist 2 "layers" of the shantt chart / dependency graph will be gone. Then the processor mergesorts the 4 halves of the array. At the end a thread sorts the 2 halves that are left.