```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
```

```
  // Parallel: 351.756
  // GNU: 1562.03
  // The sum is: 500000010
// #if OPTION == 1
//  auto begin4 = std::chrono::high_resolution_clock::now();
//  double accum4 = __gnu_parallel::accumulate(data.begin(), data.end(), 0);
//  auto end4 = std::chrono::high_resolution_clock::now();

//  std::chrono::duration<double, std::milli> elapse4{ end4 - begin4 };
//  std::cout << "GNU: " << elapse4.count() << std::endl;
// #endif

  if (accum1 == accum2)
    std::cout << "The result is: " << accum1 << std::endl;
  else
    std::cout << "The result were different, you screwed up..." << std::endl
<< accum1 << " " << accum2 << std::endl;
}
```