ADVANCED APPLICATIONS: LAB 5

I have taken the concept of multithreading from Lab 3 as the base of this lab

Environment

Core Emscripten can be cloned from GitHub sources.



Figure github clone

The latest SDK tools can be install using ‘install latest’ command in the cloned repository.

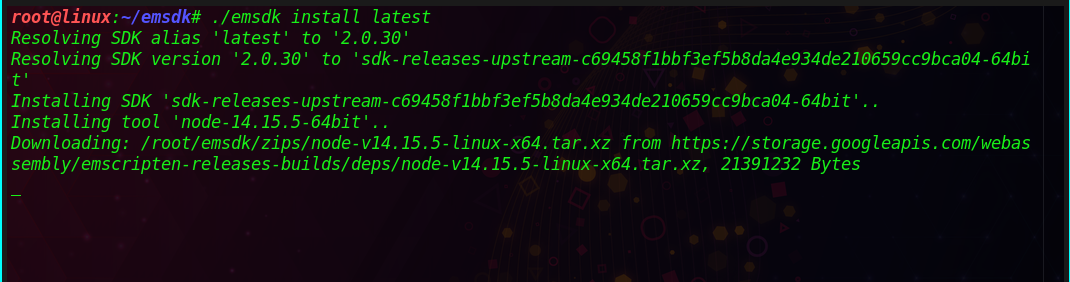


Figure SDK installation

Latest installed SDK can be made available to current user by activating. Instructions are written to ‘.emscripten’ file located at the current user’s directory.

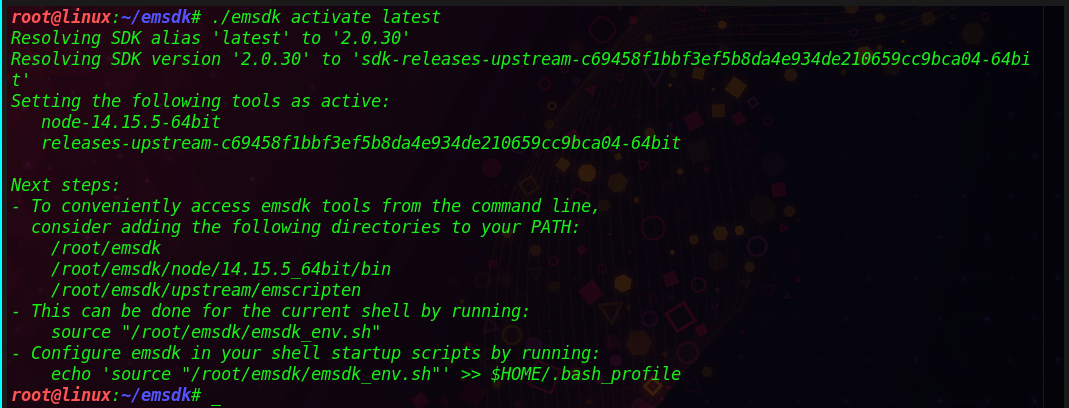


Figure Installed SDK

Active PATH and environmental variables for the current terminal available in the user context are set using source command.



.

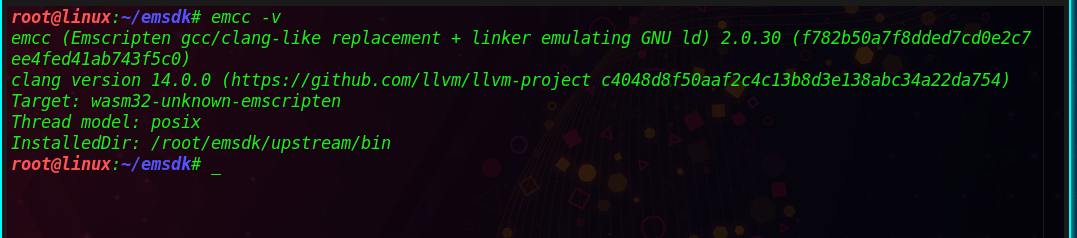
.

Figure emsdk installation complete

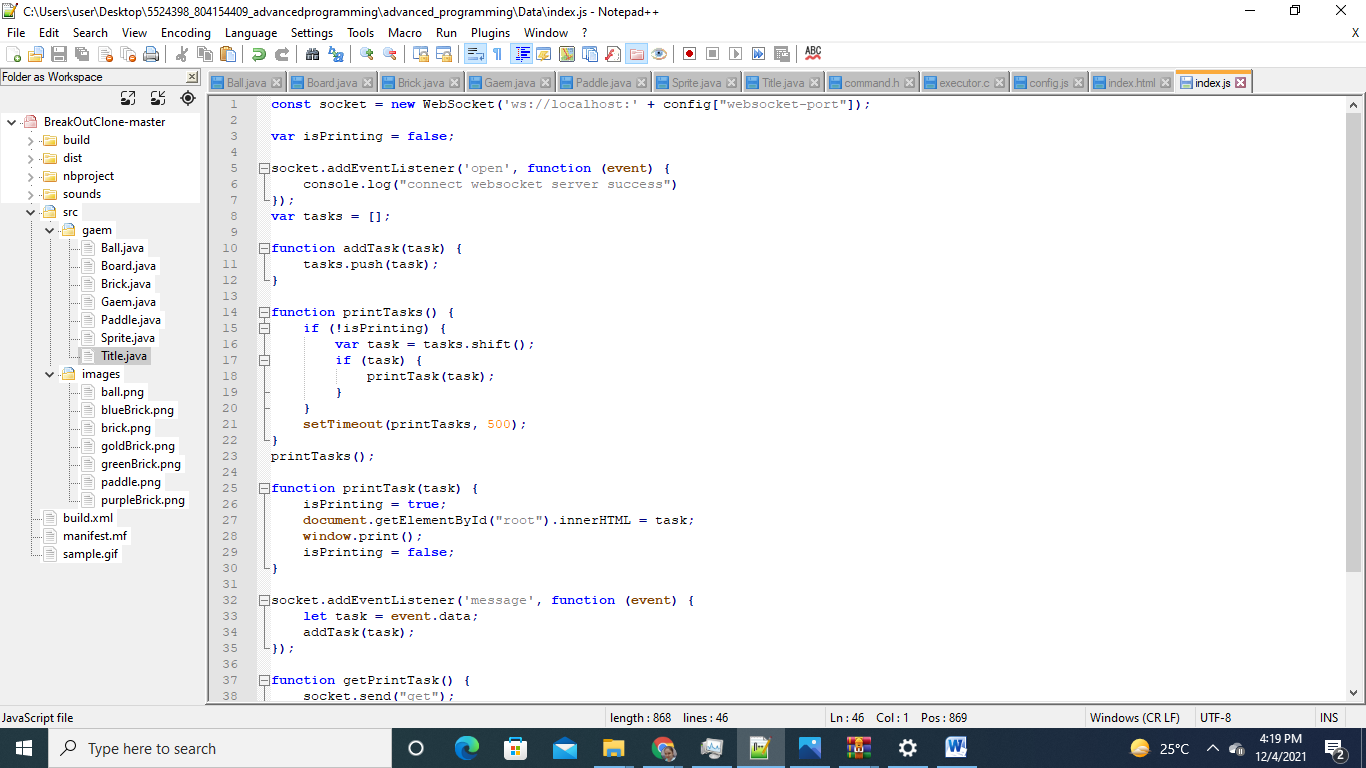
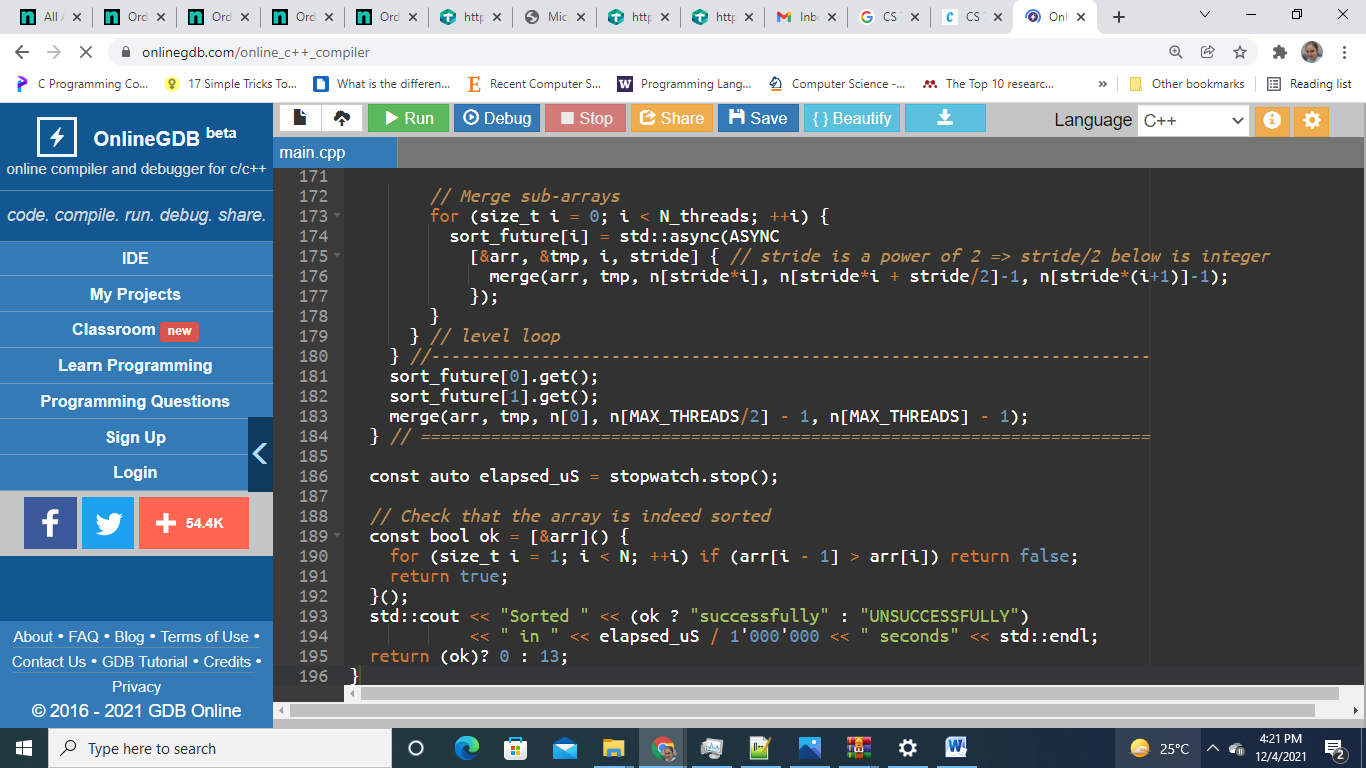


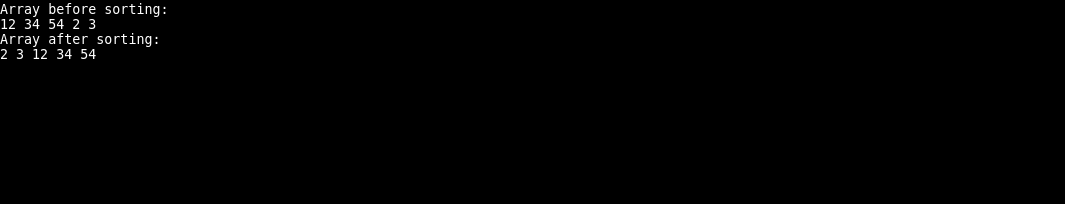
Figure 5 codes with environment variables

Code base transpiling

A sort algorithm developed in c++ is transpiled in this case to native code.

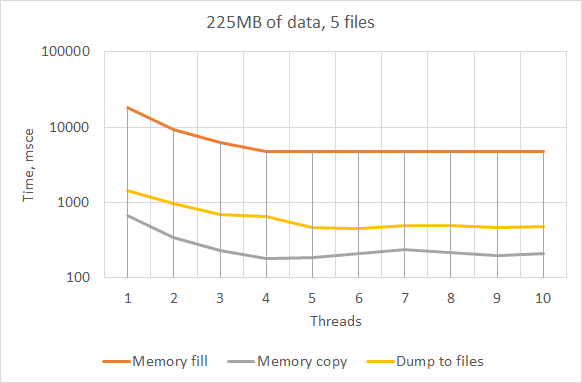
.

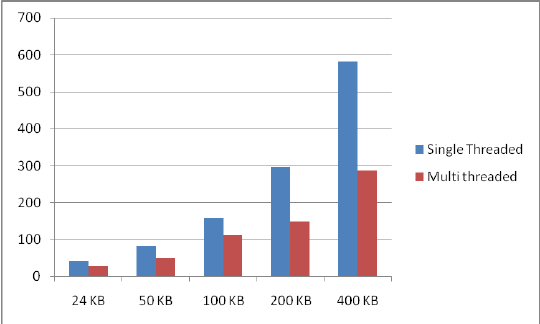
Results can be viewed in web browser



Analysis of performance

Results





Appendix

#include <array>

#include <cassert>

#include <chrono>

#include <cstring>

#include <iostream>

#include <future>

#include <string>

// To build single-threaded code, change SINGLE\_THREADED to true:

constexpr bool SINGLE\_THREADED = false;

static const size\_t N = 10'000'000; // size of the array to sort

using value\_t = float;

using array = std::array<value\_t, N>;

// The following are only for multi-threaded code

//#define FORCE\_ASYNC

#undef FORCE\_ASYNC

#ifdef FORCE\_ASYNC

#define ASYNC std::launch::async,

#else

#define ASYNC

#endif

constexpr size\_t N\_LEVELS = 2; // 1 level -> 2 threads, 2 -> 4 threads, 3 -> 8 threads

constexpr size\_t MAX\_THREADS = 1 << N\_LEVELS;

static\_assert(N > MAX\_THREADS, "N is too small!");

constexpr std::array<size\_t, MAX\_THREADS + 1> ComputeArrIndices() noexcept {

std::array<size\_t, MAX\_THREADS + 1> n{};

for (size\_t i = 0; i < MAX\_THREADS; ++i) {

n[i] = i \* N / MAX\_THREADS;

}

n[MAX\_THREADS] = N;

return n;

}

namespace util {

template<typename TimeT = std::chrono::microseconds,

typename ClockT = std::chrono::high\_resolution\_clock,

typename DurationT = double>

class Stopwatch {

public:

Stopwatch() noexcept { start(); }

void start() noexcept { \_start = \_end = ClockT::now(); }

DurationT stop() noexcept { \_end = ClockT::now(); return elapsed(); }

DurationT elapsed() const { return static\_cast<DurationT>(std::chrono::duration\_cast<TimeT>(\_end-\_start).count()); }

private:

std::chrono::time\_point<ClockT> \_start, \_end;

};

} // namespace util

// Merges two sections of arr[]: arr[l..m] and arr[m+1..r]

void merge(array& arr, array& tmp, size\_t l, size\_t m, size\_t r) {

assert(l <= m && m <= r);

const size\_t n1 = m - l + 1; // on the left

const size\_t n2 = r - m; // on the right

// Copy data to left and right temp arrays (replacing loops with memcpy makes

// no discernible difference--optimizing compilers must be doing this).

for (size\_t i = 0; i < n1; ++i) {

tmp[l + i] = arr[l + i];

}

for (size\_t i = 0; i < n2; ++i) {

tmp[m + 1 + i] = arr[m + 1 + i];

}

// Merge the temp arrays back into arr[l..r]

size\_t i = 0, j = 0, k = l;

while (i < n1 && j < n2) {

arr[k++] = (tmp[l + i] <= tmp[m + 1 + j])? tmp[l + i++] : tmp[m + 1 + j++];

}

#if 0

// Copy the remaining elements of the left array, if any

while (i < n1) {

arr[k++] = tmp[l + i++];

}

// Copy the remaining elements of the right array, if any

while (j < n2) {

arr[k++] = tmp[m + 1 + j++];

}

#else

if (i < n1) {

std::memcpy(&arr[k], &tmp[l + i], (n1-i)\*sizeof(value\_t));

}

if (j < n2) {

std::memcpy(&arr[k], &tmp[m + 1 + j], (n2-j)\*sizeof(value\_t));

}

#endif

}

void mergeSort(array& arr, array& tmp, size\_t l, size\_t r) {

if (l < r) {

const size\_t m = l + (r - l) / 2; // = (l + r)/2

mergeSort(arr, tmp, l, m);

mergeSort(arr, tmp, m + 1, r);

merge(arr, tmp, l, m, r);

}

}

int main()

{

std::cout << "Sorting "<< N <<" floats on "<< (SINGLE\_THREADED? "single thread" : std::to\_string(MAX\_THREADS)+" threads") << std::endl;

// Allocate arrays on the heap (to avoid stack overflow)

auto pArr = std::make\_unique<std::array<value\_t, N>>();

auto pTmp = std::make\_unique<std::array<value\_t, N>>();

auto& arr = \*(pArr.get());

auto& tmp = \*(pTmp.get());

// Random initialize the array

for (size\_t i = 0; i < N; ++i) {

arr[i] = rand() / static\_cast<value\_t>(RAND\_MAX);

}

tmp.fill(value\_t{}); // not needed, really

util::Stopwatch<> stopwatch;

if constexpr (SINGLE\_THREADED) { // ==========================================

mergeSort(arr, tmp, 0, N - 1);

}

else // multiple threads =====================================================

{

// Build indices of sub-arrays for the highest level.

// For lower levels, it'll be traversed with `stride` depending on the level.

static const std::array<size\_t, MAX\_THREADS + 1> n = ComputeArrIndices();

static std::array<std::future<void>, MAX\_THREADS> sort\_future;

if constexpr (N\_LEVELS == 1) //---------------------------------------------

{

sort\_future[0] = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[0], n[1]-1); });

sort\_future[1] = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[1], n[2]-1); });

}

else if constexpr (N\_LEVELS == 2) //----------------------------------------

{

// Sort four sub-arrays

static auto sort\_future0 = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[0], n[1]-1); });

static auto sort\_future1 = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[1], n[2]-1); });

static auto sort\_future2 = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[2], n[3]-1); });

static auto sort\_future3 = std::async(ASYNC [&arr, &tmp] { mergeSort(arr, tmp, n[3], n[4]-1); });

// Merge two sub-arrays

sort\_future[0] = std::async(ASYNC [&arr, &tmp] {

sort\_future0.get(); sort\_future1.get(); merge(arr, tmp, n[0], n[1]-1, n[2]-1);

});

sort\_future[1] = std::async(ASYNC [&arr, &tmp] {

sort\_future2.get(); sort\_future3.get(); merge(arr, tmp, n[2], n[3]-1, n[4]-1);

});

}

else if constexpr (N\_LEVELS > 2) // 8 or more threads ----------------------

{

// Sort sub-arrays

for (size\_t i = 0; i < MAX\_THREADS; ++i) {

sort\_future[i] = std::async(ASYNC [&arr, &tmp, i] { mergeSort(arr, tmp, n[i], n[i + 1]-1); });

}

// Process each level by spawning worker threads to merge sub-arrays

for (size\_t level = N\_LEVELS; level > 0; --level) {

constexpr size\_t one = 1;

const size\_t N\_threads = one << level; // # of threads for this level

const size\_t stride = one << (N\_LEVELS - level); // stride for indices in `n`

// On 1st pass, wait for mergeSort() futures; later--on merge() futures from previous pass

const size\_t N\_to\_wait = (level == N\_LEVELS)? MAX\_THREADS : 2 \* N\_threads;

for (size\_t i = 0; i < N\_to\_wait; ++i) {

sort\_future[i].get();

}

// Merge sub-arrays

for (size\_t i = 0; i < N\_threads; ++i) {

sort\_future[i] = std::async(ASYNC

[&arr, &tmp, i, stride] { // stride is a power of 2 => stride/2 below is integer

merge(arr, tmp, n[stride\*i], n[stride\*i + stride/2]-1, n[stride\*(i+1)]-1);

});

}

} // level loop

} //------------------------------------------------------------------------

sort\_future[0].get();

sort\_future[1].get();

merge(arr, tmp, n[0], n[MAX\_THREADS/2] - 1, n[MAX\_THREADS] - 1);

} // =========================================================================

const auto elapsed\_uS = stopwatch.stop();

// Check that the array is indeed sorted

const bool ok = [&arr]() {

for (size\_t i = 1; i < N; ++i) if (arr[i - 1] > arr[i]) return false;

return true;

}();

std::cout << "Sorted " << (ok ? "successfully" : "UNSUCCESSFULLY")

<< " in " << elapsed\_uS / 1'000'000 << " seconds" << std::endl;

return (ok)? 0 : 13;

}