**CS4414 Prelim**

**October 14, 2021. 5 problems, 20 pts each**

**1.** Consider a C++ program that reads a highly detailed engineering design for a bridge (in total, it loads 300 files containing about 3GB of data). Then the program carries out a structural analysis to evaluate stress on the various elements of the bridge, after which it creates a graphical visualization for the developer to study.

This program is currently much slower than it should be; you've done a pencil-and-paper estimate of how fast this program should run, and seems to be taking 10x longer than you would expect. Only the visualization runs at a reasonable speed. The computer used is a normal Intel-based desktop with lots of memory, but it lacks any kind of special accelerator hardware, except the built-in display graphics unit (which cannot be used as a GPU).

Upon investigation, you learn that the structural analysis turns employs a mechanical/physical simulation that is applied point by point at 5M distinct points on the bridge. At each point, a calculation is performed that sums up the stress contributions from each of 5000 "load sources" on the bridge (in total, 5M x 5000 stress contributions are evaluated).

**a.** [4 pts] In CS4414 lectures we discussed a number of possible causes for slow performance. List your "top four" ideas for things that could cause the structural stress analysis to run 10x too slowly.

**i.** *3GB is actually kind of a lot of data. The program may be stalling waiting for file I/O to occur.*

**ii.** *The application "should" be highly parallel, given this description. But to benefit from parallelism in C++, the program must conform to a set of rules that the develolper may not have known about.*

***iii.*** *C++ has many features for shifting runtime work to compile time, such as constexpr and by-reference argument passing with const annotations. The developer may not have used them.*

***iv.*** *The program might not have been compiled with –O3. Many C++ performance optimizations only kick in if you compile with –O3.*

***v.*** *(A fifth idea): Modern computers are NUMA systems but to leverage NUMA you do need to code with multiple threads. Maybe the program is single threaded.*

***vi.*** *(A sixth idea): Maybe the program actually is multithreaded but is careless about NUMA overheads such as references to "remote" memory (DRAM far from the core where the thread runs), or uses a lot of high-cost locking or some other costly mechanism.*

***vii.*** *(A seventh idea): Some coding styles are simply not very efficient, such as searching a list or tree when you could have used a hashmap ("std::unordered_map"). The programmer may have just written really bad code!*

***viii.*** *(An eighth idea): Maybe the program uses a huge amount of virtual memory and is paging heavily.*

*For problems 1.b – 1.d, we want to focus on how you would identify the <u>main</u> cause.*

**b.** [4 pts] How could the Linux "top" command help?

*Top is a great way to understand if a program is busy (and on how many cores) or idle (waiting). This will let us understand whether we should focus on slow I/O or paging, versus focus on the computational aspects of the task.*

**c.** [4 pts] How could the Linux "time" command help?
*With the time command, we can understand how much Kernel time ("sys time") is being expended. If a program is doing a lot of system calls, like file reads, the system time will be very high and this can direct our attention to the ways it is talking to the Linux kernel. If the time is mostly user time, this tells us that the costly actions are in the code itself, like in computational loops.*

**d.** [4 pts] How could the Linux "gprof" command help?
*Gprof is probably the most helpful tool of all. It tells us which methods are called, how often, and how much time is spent in each (and in its "children", namely other things it calls). It can also tell us how many times individual lines are executed.*

**e.** [4 pts] While investigating, you notice that the build command turns out to give g++ the argument –g, and not –O3. Explain what –g means, what –O3 means, and how this might contribute to the issue.
*This would cause a big slowdown. –g forces C++ to include debug information, such as which line is currently executing, which can prevent C++ from doing instruction reordering. More broadly, -O3 controls the main optimizations in C++. With –g but no –O3, we are "asking" for a slow version!*

***2.*** Still with reference to the bridge structural analysis from question 1, consider Gene Amdahl's famous formula: ***maximum speedup = 1/(1-p)***

**a.** [4 pts] Explain briefly what this formula means, in English (your wording should include a clear definition of the variable *p* that appears in the formula).
*The parameter p refers to the percentage of the program that can be done in parallel (or even the percentage that is "embarassingly parallel"). The formula basically pretends that the parallel part can all be done "on massively parallel step". This leaves (1-p) still to be executed sequentially, hence the limit to speedup will be given as 1/(1-p), his formula. For example, if p=0.5, the speedup will be at best 2x.*

**b.** [4 pts] To apply Amdahl's law to our bridge analysis program, you would need to understand how to assign a value to *p*. How could *p* be estimated?
*We would need to think of the code in terms of stages: the stage that loads data from the files, the code that performs load-sums on a per-point basis, etc. With threads, file loading can occur in parallel to computing. The sums could perhaps all occur in parallel, each doing one point. We would do a paper-and-pencil estimate of "how much of the program" was covered by things like this.*

**c.** [4 pts] List two techniques that C++ programmers use to make a program more parallel.
***i.*** *We try to write code that C++ can transform into parallel logic, by ensuring that for loops have obvious bounds, that data is densely packed, etc.*
***ii.*** *We often uses multiple threads, to leverage the multiple NUMA cores in our computer.*

**d.** [4 pts] Define the term SIMD. In what ways do modern computers support SIMD computing?
*SIMD: Single instruction multiple data, meaning "this one machine instruction will actually operate on a vector of data items." Computers often support MMX instructions, which use a SIMD model.*

**e.** [4 pts] Suppose that after replacing –g with –O3 in the compilation line, the program is still not getting any benefit from SIMD instructions. What issues of *coding style* can prevent use of SIMD instructions?
*C++ needs simple loop structures with predictable loop termination conditions, multiples of 2 when possible, data packed densely in arrays. Any function calls should be inlineable, constexpr used as much as feasible, and there shouldn't be any if statements in the code we are trying to parallelize.*

**3.** File systems.

**a.** [5 pts] When we delete a Linux file, what is done to the file name, the inode, and the blocks containing the data?

*The file name is still in the directory but the inode number will be replaced by 0 or -1. The inode itself will have its link count decremented and will be moved to a free list if there are no more links to it. The blocks will be on the file system block free list.*

**b.** [2 pts] We can access a disk in two ways: as a file system, or as a raw block storage device. Briefly define these two terms.

*The file system abstraction lets us treat the disk as a part of the Linux file system, with directories, files, links, etc. As a raw block device, the disk just holds some huge number of blocks (size might be 512 bytes or larger). This just treats the disk as a storage unit.*

**c.** [5 pts] Describe a way to recover a deleted file by accessing a disk in raw "block" mode.

*If a file was recently deleted we may be able to find the inode still on the free list, the name still in the directory, and the blocks still on the free blocks list. We could then reconstruct the entire file.*

**d.** A *symbolic link* is a feature available from the Linux file system.

**i.** [2 pts] Briefly explain what a symbolic can be used to do.

*A symbolic link is just a file, B, containing some pathname, A. If Linux is told to access B, when it discovers that B is a symbolic link, it will just continue its search using this other pathname, A.*

**ii.** [2 pts] Suppose that A is a file, and that B is a symbolic link to that file. Someone edits the file using B to access it. Would a person who later looks at the file using A be able to see the edits? Explain.

*The actual file is really the same – in the example above, A is the file. B is just a second way to get to it. So if we edit the file, A gets modified and anyone accessing it via name B sees the edits.*

**iii.** [2 pts] If A is deleted, could B still be used to access the file? Explain.

*No, because when B is used as a file name, Linux silently switches to pathname A, and then will discover that there is no such file.*

**iv.** [2 pts] If B is deleted, could A still be used to access the file? Explain.

*Yes, because A was never touched (in fact A doesn't even know that B is a second pathname for accessing it). So A was unaffected by deleting B.*

**4.** C++ question: General understanding

**a.** [4 pts]

**i.** [2 pts] Why do we use namespaces in C++? Give just one main reason.

*To avoid accidentally using the same name for two totally unrelated purposes.*

**ii.** [2 pts] What's the point of writing **#pragma once** at the top of the C++ header files?

*So that if an include file (xxx.hpp) is included more than once, no compilation errors will occur.*

**b.** [4 pts]

**i.** [2 pts] Write the main difference between static class variables and regular (non-static) class variables.

*There is a single shared instance of a static variable, across all object instances. With a regular class variable, each instance has a private copy.*

**ii.** [2 pts] Give one example scenario where you would use a static class variable.

*In our traffic simulator, we might have a total_driving_time variable in the Car class, to sum up the total time spent driving over all the cars. As each car object reaches the destination, it adds its own driving time to the static (shared) total.*

**c.** [4 pts] This next question is based on a data structure you used in homework one, and that was discussed in recitation. Consider the following code:

```
#include <queue>

class Student {
    int score;

public:
    Student(int score) : score(score) {}
};

int main() {
    std::priority_queue<Student> students;
    Student x(97);
    students.push(x);
}
```

Now answer the following questions:

**i.** [2 pts] This will not compile. What will the error be?
*A priority queue can implement a queue of Student objects, but needs a way to do <=> comparisons. For example, it could do ordering by student score. The compilation error will be a complaint that the default comparator doesn't work for Student objects.*

**ii.** [2 pts] How will you fix this? Explain in words. No need to write code.
*We would supply a comparator function when we construct the priority queue.*

**d.** [4 pts] Assume that the following lines of code appear in the body of a method somewhere in your program:

```
int x = 3;
int& y = *new int(7);
int *p = &x;
int *q = &y;
```

Answer True or False for the following. No explanations required.

**i.** [1 pt] **&x** is a memory address on the stack
*True*

**ii.** [1 pt] **&y** is a memory address on the stack
*False (y is an alias – a reference – for an int that was allocated in the heap)*

**iii.** [1 pt] **&p** is a memory address on the stack
*True*

**iv.** [1 pt] **&q** is a memory address on the stack
*True*

**e.** [4 pts]

**i.** [2 pts] Define *memory leak* and explain very briefly how the C++ "new" operator can sometimes lead to memory leaks.
*A memory leak occurs if we allocate memory with malloc or new, fail to the delete that memory, but no longer have any references to the allocated memory region.*

**ii.** [2 pts] C++ offers several protected wrappers for pointers that can avoid memory leaks. Two of those are **std::shared_ptr<T>** and **std::unique_ptr<T>.** How do they differ, and when might **std::shared_ptr<T>** be more appropriate?
*A unique_ptr is the "only reference" to some object. When it goes out of scope, the object is deleted. A shared_ptr can be copied, and each copy becomes an additional reference to the same underlying object instance. When all references have gone out of scope, the object will be deleted, but until then they all point to the single shared instance of the object.*

**5.** C++ question: Exploring **std::vector<T>**

**a.** [4 pts]

**i.** [2 pts] Why does a vector sometimes need to call **malloc?**
*With std::vector we can make the vector longer just by adding new elements. But this means that sometimes the vector must grow by calling malloc and then copying itself into a larger memory region. That copy could be costly!*

**ii.** [2 pts] How does a vector avoid calling **malloc** every time a new element is inserted?
*The vector has a field called capacity which is increased exponentially (for example, doubled) every time size equals capacity and a new element is being inserted. So, for the duration that size trails capacity while it is increasing by 1 each time a new element is inserted, there is no reallocation happening.*

**b.** [4 pts] Consider the following two vectors:

```
std::vector<int> v1 = {1, 2, 3, 4};
std::vector<int> v2 = {7, 16};
```

Answer True or False for the following and explain your reasoning in one line.

**i.** [2 pts] **v1.size() == v2.size()**
*False. Size() gives the number of elements. The sizes are 4 and 2.*

**ii.** [2 pts] **sizeof(v1) == sizeof(v2)**
*True (this is gives the size of the std::vector object, not the size of the data it holds).*

**c.** [4 pts] A vector provides O(1) random access. That is, given an arbitrary index **i** within the range of the vector, the element at that index can be retrieved in constant time. The vector achieves this by making sure that the elements are stored contiguously in memory and then finding the address of the element at index **i** by a formula that uses the address of the first element and the value of **i**.

Consider a vector of integer vectors i.e., **std::vector<std::vector<int>> vec;** with a fixed size of 5, and suppose that the internal vectors i.e., **vec[0], vec[1], ..., vec[4]** are automatically being resized because of insertion operations. Would this cause the outer vector to be resized? Explain.
*No. The outer vector held 5 elements the whole time, and that number 5 didn't change. Each element was an inner vector – and in this case, it was the inner vectors that changed size.*

**d.** [4 pts]

**i.** [2 pts] Explain the difference between **std::vector<T>::push_back(const T& value)** and **std::vector<T>::emplace_back(Args&&… args)**.
*Push_back and emplace_back are both used to add an element to the end of the vector. Push_back takes an already constructed value object and creates a copy of it inside the vector. Emplace_back*

*creates a new element directly inside the vector where it needs to be. For this purpose, it takes the constructor arguments required to construct the value object. Thus, it avoids copying.*

**ii.** [2 pts] The algorithm library in C++ (**#include <algorithm>**) provides many useful container functions. We will consider one such function, **std::sort(RandomIt first, RandomIt last, Compare comp)**, which sorts the elements in the range **first** to **last** using the given comparator function that compares two elements a and b. Our range will be the entire vector for this problem where the vector and its comparator function are defined as:

```
std::vector<int> vec = {60, 55, 39, 48, 50, 44, 57};
bool compare (int a, int b) {
     return std::abs(a - 50) < std::abs(b - 50);
}
```

Note: **std::abs (int n)** returns the absolute value of an integer. For example, **std::abs(-1) = 1**.

Write the vector contents after the call to sort:

```
std::sort(vec.begin(), vec.end(), compare);
```

*50, 48, 55, 44, 57, 60, 39*

**e.** [4 pts] The following program has a bug: the developer expected it to output 0 1 2 3 (each on its own line), but in fact it has no output at all! On checking, you find that nums.get_vec().size() is always 0 after calling push_back.

```
#include <iostream>
#include <vector>

class numbers {
    std::vector<int> num;

public:
    std::vector<int> get_vec() {
        return num;
    }
};

int main() {
    numbers nums;
    for(int i = 0; i < 4; ++i) {
        nums.get_vec().push_back(i);
    }
    for(int n : nums.get_vec()) {
        std::cout << n << std::endl;
    }
}
```

Explain the bug and describe a simple way to fix it (no need to show us the code). It is fine if your proposed fix requires some change to the given code.

*The program is making copies by passing the num vector back by value in the get_vec method. The copy is modified, but then discarded. To fix the issue, it suffices to return the vector by reference.*