Efficient Parallel C++

As indicated by the title, this lab course consists of two aspects of C++ programming. First, efficiency (here mostly fast execution), and second, parallel execution (mostly also for acceleration).

In this initial piece of information we want to give you a short overview over some common mistakes, and some techniques that we consider important for efficient programming in C++. All these techniques can be separated into two categories. General purpose tips/errors that you can avoid easily, and problems that are more problem-specific. These often need problem-specific solutions like reducing branch mispredictions within an inner loop, data restructuring for cache purposes, or specialized data-structures. Sometimes we even change the algorithm design to allow for better optimizations.

These 2×2 categories of mistakes/optimizations form the following grid:

	general	specialized
sequential		
parallel		

1 Some really basic tips

know where to optimize This is one of the most important guidelines when optimizing code. The more often one section of the code is executed (i.e., inner loops), the more impact optimizations have on the overall performance. Accelerating a function by one microsecond might be valuable when said function is called millions of times. On a similar note, optimizing a function that only uses 3% of your overall running time can never save you more than that 3%.

bad optimizations Always measure all running time changes caused by one of your optimizations. Some optimizations (even those mentioned here) might have negative impact. Only regular measurements show if you are moving into the right direction.

premature optimization is the root of all evil ... but oversimplification is even worse! Code that was never meant to be optimized is terrible to optimize! Therefore, you should not try to write optimal code all the time, but always be aware of pitfalls and make sure you are not cutting off any ways to efficient code (see also `http://ubiquity.acm.org/article.cfm?id=1513451`).

2 Efficient Programming

2.1 General: Performance Bugs

know the cost of your operations Not all operations are created equally. For us it is important to know which operations are faster than others. Slow operations include calling functions (that have not been inlined), complex numeric functions (like trigonometric functions, square roots ...) but also integer division/modulo computation.

memory accesses One special type of operation that we want to mention specifically is memory access. As you all hopefully know there is a hierarchy of memory tiers (register, cache, main memory, and hard disks), where each level gets progressively slower. Especially the jump from cache to main memory is very important for programs, since this is where one memory access switches from around 3 cycles to around 200 cycles.

This makes a single memory access significantly more costly than many other supposedly slow operations like computing a hash function. Luckily, memory accesses can sometimes be grouped together (and pipelined), but this is only the case if the second memory access does not depend on the first (independent reads).

Another reason why the memory hierarchy is important to keep in mind is that we cannot specifically control the contents of our caches. Instead, all data that is accessed will be stored in a cache when it is accessed. This works in small memory sections called cache lines; one cache line will always be pulled as a whole. Cache lines that are accessed repeatedly will usually be cached and thus are usually faster to access.

indirections (too many pointers) In C++ objects/data can be stored either on the heap (cre-

ated with `new` or `malloc`) or on the stack (local variables or static/global variables). We also have several options of "naming/addressing" those elements. There is the local element which can be changed immediately, the reference to an element, and a pointer.

Whenever we access an element through a pointer (or reference for that matter) there is a waiting period, especially since the computation is often dependent on the read data. Furthermore, each of these indirections has the potential for a cache miss and thus the enormous waiting period connected to a main memory access (see above).

Accessing local elements might often be faster than accessing elements stored at the end of a reference or pointer. The reason for this is that offsets relative to the current stack frame (necessary to access the members of the object) can usually be computed at compile time, and the stack is almost guaranteed to be in the cache.

unnecessary copying (language specific) An element is copied whenever it is passed from one context to another (and no reference or pointer is used). Sometimes copying is exactly what you want (mostly for basic types like `int`, `double` or derived types like `std::pair<int, int>`). Other times the copy can take a long time, for example for larger objects that might even have member objects (that are also copied), or even data-structures. To identify and prevent unnecessary copying you may declare the copy constructor of these larger objects as deleted, thus preventing any code that tries to copy the object from compiling.

polymorphic inheritance (language specific) Other lectures will probably tell you about the maintainability and code de-duplication benefits that inheritance offers. When programming for algorithm engineering this is often not as necessary. Different classes are often only used to try out different approaches and often do not interact with each other. For us the problems of inheritance outweigh the benefits.

The `virtual` keyword is essential for polymorphic inheritance (the most specialized implementation of the function is called). But its implementation works with a vtable. Each object has a pointer to its vtable, which stores the specialized virtual functions. Because of this system, looking up a virtual function introduces "avoidable" indirections.

We prefer arrays and unbounded arrays (`std::vector`) as our main data-structures, but storing derived objects in base-class arrays is impossible since not all types have the same size.

Note that inheritance can still be used for code de-duplication, but it should be used conscious of its shortcomings. There are some techniques (e.g., CRTP, curiously recurring template pattern) and keywords (`final`) that can alleviate some of the problems (we do not explain these here – check the internet if you are interested.

some things are free (language specific) Modern compilers are able to do a lot of things for you. You should be aware of the things you can use them for.

Short functions that are called very often can be *inlined* to eliminate expensive function calls. Optimizations with *compile time constants*, when you can prove to the compiler that something is known at compile time (usually by making it a template parameter or `constexpr`), it can be used for optimizations. For example, by replacing a division by 2 with a right shift of the number (`i >> 1`).

This compiler magic only goes so far; just because you know something can be optimized will not make the compiler do so. Sadly, if you want to check if something gets optimized things become tricky, sometimes looking into the assembly is all we can do. If the code in question can be boiled down to a couple of lines, then `https://godbolt.org` is a good address to see what your compiler would do to those lines.

As always, not every optimization is right for every situation. The common example for this is loop unrolling. Sometimes, it will pay off and other times it might even make your code slower.

initializing memory (language specific) Whenever you are working with a larger amount of data, you should be aware that initializing arrays or vectors is usually not free. It is notable that POD types (Plain Old Data, see `http://en.cppreference.com/w/cpp/concept/PODType`) such as `int`s and simple `struct`s have different behaviour than complex types. When you are declaring a new POD object, it will not be ini-

tialized to any specific value (i.e., initialization is "free"). Non-POD objects will generally be initialized whenever they are declared (locally on the stack, or using `new` on the heap). This is especially important to know when designing data types (should they be POD?) and when constructing large arrays, vectors or similar containers.

When you don't want an array of non-PODs to be initialized, then there are two possibilities: Use `malloc` (to get heap memory) and cast the returned pointer to the appropriate type, or use an `std::vector` + `reserve()` to get the appropriate memory, then you fill this memory with `emplace_back()`.

The destruction of vectors (of non-PODs) can also cost a lot of time, since the destructor of each element needs to be called. This can impact performance in cases where vectors are constructed and destructed frequently.

2.2 Problem-specific: Optimization Possibilities

cache effects (control cache line usage) We have already mentioned the importance of caches to reduce the impact of slow main memory accesses. But since programmers do not have any direct control over caches, we have to find some ways to manipulate the cache content. The best way to store data is often dependent on the algorithm and its specific data access pattern. For the most part, we use two ways to improve the cache performance of our applications.

The first is *cache line usage*. As previously mentioned, data is always read into the cache in bulks called cache lines (64 bytes). We can save a lot of time if we can use all of these bytes instead of just a fraction. Therefore, it is important to control what elements lie close to each other in the memory. If we use arrays to store objects, then adjacent objects are stored in the same cache line (if they fit). This can often be used to iterate over an array very efficiently.

Another way to increase cache line usage is to reduce unnecessary data. If you only ever need one member when iterating over all elements, e.g., iterating over an edge list and only reading the target node, then you can benefit a lot by using an array that only stores these target nodes. Sometimes, an array of elements should be replaced by distinct arrays for each member.

The second way to improve cache performance in your applications is to have regular access patterns. This allows the prefetcher to guess necessary cache lines, and pre-read them. The prefetcher is a heuristical element of the cache mechanic. You should not count on it, but it usually works if you either read through one array with predetermined offsets or if you repeatedly read from two arrays with the same offset.

branch mispredictions Whenever the processor reaches a conditional jump during its execution, it will execute one of the branches speculatively. This makes the execution faster whenever this choice was right, and slower whenever it was wrong. This is another aspect of program execution that is really hard to influence. The best way to reduce the number of branch mispredictions is to reduce the number of conditional jumps, especially if those jumps occur within the main loop of your algorithm.

The bounds of loops are often the cause for branch mispredictions, usually the heuristic will say that the loop gets executed one more time. This might not actually be the desired behaviour for short loops that do not have a lot of repetitions.

complex data-structures (somewhat language specific) This point is the culmination of many of the previous tips. **Try not to use anything but arrays, vectors, priority queues, and hash tables.** And of the later two, do not use the STL implementations, but implementations tailored to your algorithm. There are some rare exceptions to this rule, but you should be aware of the architecture, and the performance impact of any data-structure.

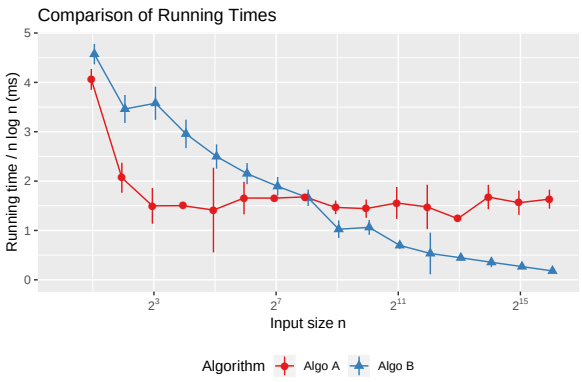
You should think about the number of cache misses per access; the number of independent and dependent reads; and other performance criteria (e.g., amortized time vs. worst case time). Also think about initialization cost and costs for resetting temporary data structures. All these criteria are especially important if you start implementing your own data-structures.

3 Visualizing Measurements

Measuring the impact of performance optimizations is important, and plotting measurement results allows you and others to quickly understand what that impact is. However, for this to work, you need to make sure you present relevant and complete metrics in an intuitive way. Here are some guidelines you should follow:

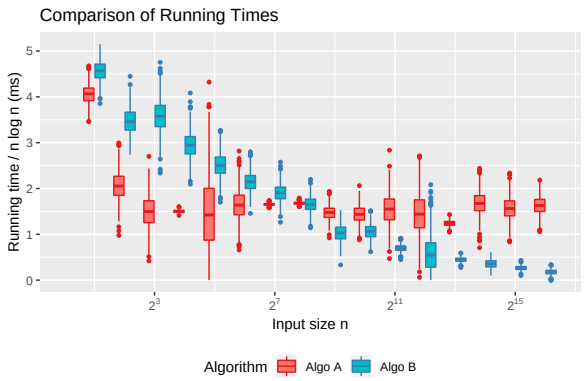
- Have a short, clear title (e.g., "Running time of Algorithm A").
- Have a legend (unless you only have one line/kind of data in your plot).
- Label the axes.
- The y-axis (i.e., your measurement) should start at 0, unless there is a good reason not to. Otherwise, you exaggerate relative differences – if that is what you want to do, it's often better to show the full plot and a second, "zoomed-in" plot.
- Think about scales. Sometimes, you might want to use log-scale on one or both axes, other times you don't. If your x-axis is the input size, it's often a good idea to scale the y-axis by the expected (theoretical) running time (or whatever your measurement might be). For example, when plotting a sorting algorithm, you might want to scale your y-axis by $n \log n$ – that way, your lines will be mostly parallel to the x-axis and it's easier to see differences.
- Don't rely purely on colour to differentiate data points. Colours depend heavily on the medium (e.g., LCD vs. beamer vs. paper – especially on black-and-white printers!), and colour blindness is a thing. Instead, use different shapes, line types, or textures in addition to different colours.
- Don't plot just the average. Variance in the data is important (unless it's not, but you can only judge that if you looked at it). There are many ways to show the variance, the most common ones are plotting mean and standard deviation, or using boxplots.
- Don't try to show too much information in one plot. If a plot is too busy, it's hard to filter out the important parts quickly, which is what plots are all about. Instead, try to split your data into multiple plots, each with a different focus – it's okay to show data multiple times in different contexts. However, it can be good to keep an "overview" plot, even if not much can be learned from it, just to contextualize the more detailed plots; see also the "zoomed-in" plots mentioned above.

Let's look at some examples. The first is a comparison of the running times of two algorithms:



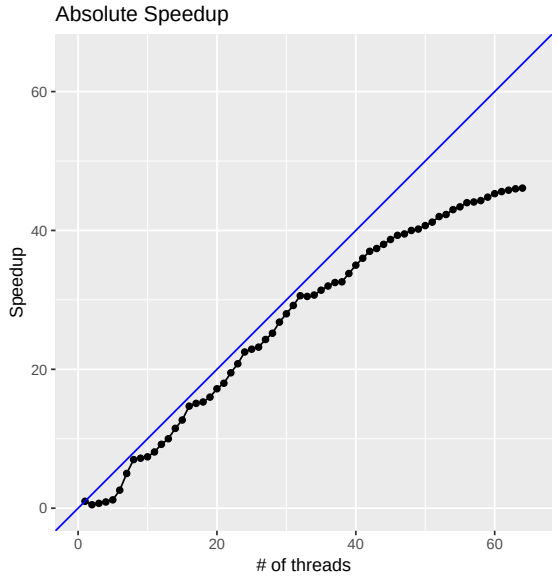
The x-axis is in log-scale, which isn't uncommon for input sizes, where you want to show the behaviour for very small and very large inputs. The y-axis is linear, but the values are actually scaled by $n \log n$. This lets us easily see that algorithm A seems to follow that trend (the results are roughly parallel to the x-axis), but algorithm B doesn't. We can also see that both algorithms have a certain overhead (they're much slower for small input sizes).

The second example is the same data, but in a boxplot:



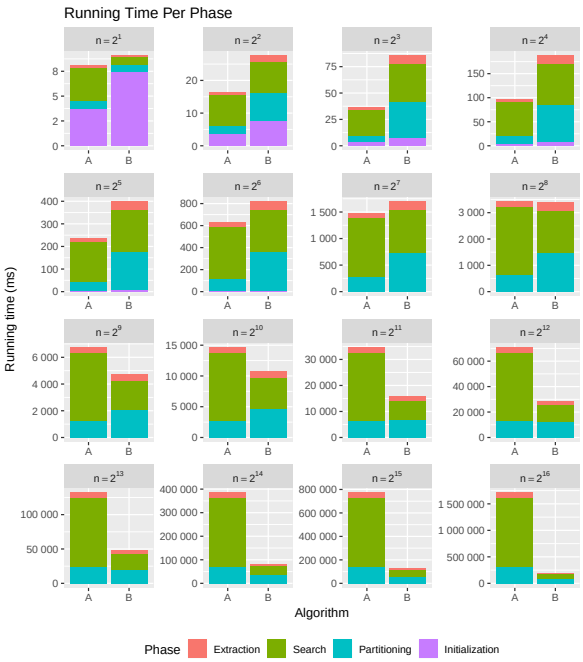
This looks much busier than the first plot, and in this case, it doesn't add any interesting information: There don't seem to be any skewed distributions, so all the boxes are symmetrical. That fact can be mentioned in the text, but is probably not worth the cognitive overhead of a boxplot.

The next example is a speedup plot:



These are common for parallel algorithms, as you want to know how effective your parallelization is and what kind of overhead you incur. For evaluation, you usually don't need to measure every possible number of threads, as that makes your experiments take much more time – but it's nice to have for the final implementation.

The final example is a bit more complex. It shows a breakdown of the running times per phase of the algorithm:



This plot is probably too small to be shown in this single column, but it serves to demonstrate how a lot of information can be shown succinctly: We can see that algorithm B has a higher initialization cost than algorithm A, but that quickly becomes irrelevant as the input size increases. Instead, the partitioning becomes a large chunk of the total running time for B, but not for A. As the input reaches large sizes, the better scaling of B shows, especially in the search phase.