4.1

Multiprocessing – parallelizes a program over multiple compute units (ex CPU cores) in order to exploit redundant resources such as registers and arithmetic logic units of different cpu cores to speed up computation

Multithreading – shares hardware resources such as caches and RAM of a single core or multilple cores in order to avoid idling of unused resources.

A multithreaded program can but does not necessarily need to utilize distinct CPU course and thus, depending on the situation may also be the definition of multi processing. Moreover multiprocessing does not explicitly exclude the use of shared resources and consequently could be implemented within a multi threading scenario but could also employ heavy weight processes communication over sockets or message passing interfaces.

The actual number of concurrently running threads should be adjusted to roughly match the amount of physical cores of your system since the OS might serialize their execution using expensive context switches if their number exceeds the amount of available cores. This behavior is called oversubscription and should be avoided to prevent performance degradation.

All threads share the resources of the parents system process meaning they can access the same memory space. This is advantageous since threads can respond with low latency and benefit from lightweight inter–thread communication. A drawback of threads is that a thread could easily spy on the data of another thread.

One should use distinct system processes with independent memory spaces to implement security critical applications as realized by the popular Google Chrome browser. Concluding, threads are a lightweight shared memory mechanism for the concurrent execution within the memory space of a single system process in contrast to independent system processes acting on distributed memory and communicating over heavyweight channels such as sockets.

In order to ensure that all Spohn threads have finished their work we have to wait for them in the master thread. This is accomplished with a call to join. Alternatively, one can detach threads meaning the master process kills them during termination without waiting for them.

1. Each thread can only be joined or detached once 2. A detached thread cannot be joined, and vice versa 3. Join or Detached threads cannot be reused 4. All threads have to be joined or detach within the scope of their declaration

The method emplaceback moves the thread objects explicitly. The method pushback moves the thread objects implicitly. We need to store the thread handles explicitly in order to be able to access them a second time during the joint phase.

4.2

Threads can execute functions with arbitrary arguments in return values. However, red objects do not offer a straightforward way to access the return value. This might be acceptable behavior in our hello world example but limits are code to fire and forget scenarios where we launch a certain amount of threads that execute some work without providing any feedback.

The traditional error handling model in the C programming language reserves to return value of a function for the error code. And example is the main function which returns and Imgur indicating whether it terminated successfully or unsuccessfully. Hence, other computer quantities are usually passed beer pointers in the argument list which are subsequently manipulated inside the function body.

Note that the communication over pointers is possible since all thread shared the same memory space in a multi threading scenario. Nevertheless, you have to be aware of a potential pitfall: The memory past via eight pointer in the first for Loop has to be persistent during the execution of threads. In contrast, of beer you bought or objective fine within the body of the first for loop when we destroy immediately after each generation since we leave at scope. Hence, the thread with operate on potentially free memory resulting in a segmentation fault. Concluding, you have to guarantee that the object manipulated in threads remain existent during execution and that there are no data races on shared resources.

C++ 11 provides a mechanism for return value passing specifically designed to fit the characteristics of asynchronous execution. The programmer may define so-called promises that are fulfilled in the future. This is achieved with a pair of tied objects S equals (P, F) where P is a writable view of the state S, the promise, which can be set to a specific value. This signaling step can only be accomplish once and that is called fulfilling the promise. The object F, The future, is a readable view of the state S that can be accessed after being signaled by the promise. Hence, we establish a casual dependency between the promise P in the future F that can be used as synchronization mechanism between a Spawned thread and calling master thread.

This mechanism can be applied to functions that communicate one or more values to the master thread. In the multi variate case you can simply past multiple promises in the argument list. Nevertheless, the described approach seems to be unhandy when returning only a single value. Fortunately, C++11 offers a mechanism that transforms functions to tasks with a corresponding future object handling the return value. The future header provides the function STD:: package\_task that allows for the convenient construction of task objects. A disadvantage to the described approach is that you have to hardcode the signature of the called function and the template parameter of STD:: packaged\_task which, moreover, affects the types of the arguments this might become a problem if you want to store several tasks wrapping distinct functions with different arguments in the same container class.

4.3

Passing many arguments to a thread in an elegant way can be achieved by exploiting the capture mechanism of anonymous functions, so-called lambdas.

Oversubscription is when you spawn to many threads which forces the OS to time slides the execution of threads using expensive context switches.

The automatic capturing of the scope by reference is a viable option for the lambda: we passed the thread identifier ID as const Reference in the arguments and seamlessly transfer the remaining variables by capturing the whole scope on a reference level. Note that capturing the whole school by value is not a tractable options since it would perform redundant copies of the huge matrix a and further would write the results to A copy of B that cannot be accessed in the scope of the master thread.

The excessive invalidation of shared cash lines is called false sharing. False Sharing may drastically influence the run time of your program. Concluding, make sure that you avoid excessive updates of entry stored in the same cash line when using more than one thread in parallel. Moreover, try to cash intermediate results and registers in order to reduce the update frequency to cashed entities as demonstrated with the auxiliary variable ACCUM and the cyclic and block closure.

4.4

The static distributions in the previous section are called static because the assignment pattern of tasks to threads is predetermined at program start. This implies that we have carefully analyze the problem before hand and have subsequently chosen a suitable distribution of threads. This might become difficult if the time taken to process a certain task varies heavily. The case were a few threads still process is their corresponding chunk of tasks while others have already finished their computation is called load in balance.

(Alpha(c + 2i(0)\*c + c^2))/2

We should choose C as small as possible in order to approximately equalize the execution times for each chunk. Concluding, it is advisable to employ static schedules with a small chunk size when fighting load in balance.

Task assignment at runtime is called dynamic scheduling and can be realized on a per task basis or as demonstrated before on a trunk level. Dynamically assigning tasks illuminates access run time at program start by dynamically building the skilled work distributions.