SCS Project Documentation

Measuring processes’ execution times in different programming languages

Anda Tatomir,
Group 30431

1. Specification

Develop programs which measure the execution time of memory allocation, memory access(statically and dynamically), thread creation, thread context switch and thread migration. Compare the results for 2-3 programming languages.

2. Glossary

- Microbenchmark: a program that attempts to measure the performance of a “small” operation or sequence of code
- Memory allocation: the process of reserving a portion of computer memory for the execution of processes or programs
- Thread: a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing resources (such as memory)
- Thread context switch: the process of storing the state of a thread, so that it can be restored and execution resumed from the same point later.
- Static memory access: at compile time
- Dynamic memory access: at run time
- Thread migration: the migration of a thread on another physical core, managed by the operating system scheduler. For example, when a thread has to wait for a number of reasons, another software thread may be given permission to execute on that core.

3. Objectives

- Writing microbenchmarks in Java and C/C++ for memory allocation, memory access, thread creation, thread migration and thread context switch, taking into account the factors that might influence the accuracy of the results obtained
- Obtain average execution times in milliseconds for each process, in each programming language, so that it can be determined, by comparison, which programming language has better performance for each task
- In order to obtain results as accurate as possible, each test will be run at least 1000 times and the average of results will be computed and

recorded. Also, the factors that could influence the result will be taken into account and their effect minimised, so that only the execution time of the process we are interested in is measured, not the additional overhead.

4. Bibliographic study

In order to have a correct measure of a process' execution time, we need to write benchmarks that give accurate results. Although the logic of the benchmark code is correct and should produce correct results, there are some factors that have to be taken into account, depending on the language the benchmark is written in, in order to make sure the benchmark does not give, in fact, incorrect results without us realising it.

4.1 Java microbenchmarks

When writing microbenchmarks in Java, the most important thing to consider is that the
compilation process for a Java application is different from that of statically compiled languages like C or C++. A static compiler converts source code directly to machine code that can be directly executed on the target platform, but the Java compiler converts Java source code into portable JVM bytecodes, which are "virtual machine instructions" for the JVM. Unlike static compilers, javac does very little optimization, instead the optimizations that would be done by the compiler in a statically compiled language are performed at runtime.

The current JVM provided by Oracle is called HotSpot because it seeks hot spots of use in the code (places where code is more intensively used) for just-in-time optimization. HotSpot first runs an interpreter that only compiles the code executed frequently. Furthermore, it performs continuous recompilation, meaning that the JIT (Just-In-Time Compiler) as the program runs, Java bytecode is continuously recompiled into machine code. Recompilation can be triggered at unexpected times by the loading of new classes or the execution of code paths that have not yet been traversed in already-loaded classes.

The compiler often optimizes the code so much that little of the instructions in the benchmark are executed, so the test runs faster than expected. This happens because the compiler will realize that the result of the benchmark code is not used for anything, or that the code does nothing useful. Another thing to consider is that introducing code besides the one to be measured means that the test now measures that code also and that the extra code may influence the way the Java JIT optimizes the code. Therefore, the key to writing effective benchmarks is finding a way to reduce or, ideally, eliminate compiler optimizations and making sure that we measure only the instructions we are actually interested in.

- Dead code elimination
    - benchmark programs often do not have an effect on the outcome of the program execution, so the compiler thinks they are dead code
    - a problem for both Java and statically compiled languages
- Warmup
    - In order to measure the compiled performance of the code, not the interpreted one, the JVM has to be “warmed up” by executing the target code enough times that the compiler will have had time to run and replace the interpreted code with compiled code before starting to time the execution and trigger all the necessary initializations
    - Depending on the compilation time and how much faster the compiled code is than the interpreted code, small changes in the number of iterations can result in big differences in the measured execution time
- Garbage collection
    - The time spent in garbage collection can be seen by running the benchmarks with `-verbose:gc`, and the final time should be adjusted accordingly

The tests can be run with `-XX:+PrintCompilation`, `-verbose:gc`, so that it can be observed if the compiler and other parts of the JVM doing unexpected work during your timing phase.

Tools for measuring Java performance:

- JMH(*Java Microbenchmark Harness*): a toolkit provided by OpenJDK that helps you implement Java microbenchmarks correctly
- VisualVM: a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine It allows you to trace a running Java program and see its the memory and CPU consumption. It can be also used to create a memory heap dump to analyze the objects in the heap

4.2 Memory access

Memory access time can be measured by performing several read or write operations at different memory locations.

The time required to retrieve a byte or a word from memory depends on whether the data is in the processor's cache or not. If the data is not in any of the cache layers, the main memory will have to be accessed, so the amount of time will be larger.

In C++, a comparison can be made between accessing memory using pointer arithmetic and array indexes.

In Java, memory cannot be normally directly referenced, because it is managed by the JVM. However, using the `sun.misc.Unsafe` class there is a way to directly allocate and deallocate memory as well as write and read it.

4.3 Memory allocation

- Dynamic memory allocation

  Memory allocated at runtime (ex: using malloc function in C), on the heap. In C, persistent until free()(delete for C++) is called. In Java, persistent until the garbage collector determines it is no longer accessible, when it will be automatically destroyed.

- Static memory allocation

  Is allocated at compile time, before the execution of the program begins, usually on the stack. Size of allocated memory cannot be changed after the initial allocation.

In Java, when declaring an object only a reference is created, which is put on the stack. Also, all the parameters and global variables of a method go on the stack. Memory is allocated only when an instance of an object is created, using new() and the memory is allocated on the heap.

The programming language does not offer the possibility to let the programmer decide if an object should be generated in the stack, but if an object is used only with a thread or method, the JVM may decide to create the object on the stack.

4.4 Thread context switch

Thread context switch can usually occur if:

- The thread has run for its given amount of time before it has to return control to the kernel that will decide who's next.
- The thread was preempted. Occurs when another thread needs CPU time and has a higher priority (the thread that handles mouse/keyboard input)
- The thread is blocking on some operation or just called sleep() to stop running.

So, for measuring thread context switch, a number of threads could be synchronized using mutexes or put to sleep. However, the benchmark will most likely measure the overhead introduced by the call to sleep or locking/unlocking mutexes.

4.5 Thread migration

In C++ thread affinity can be set by using the `pthread_setaffinity_np` on Linux and by using `SetThreadAffinityMask()` on Windows

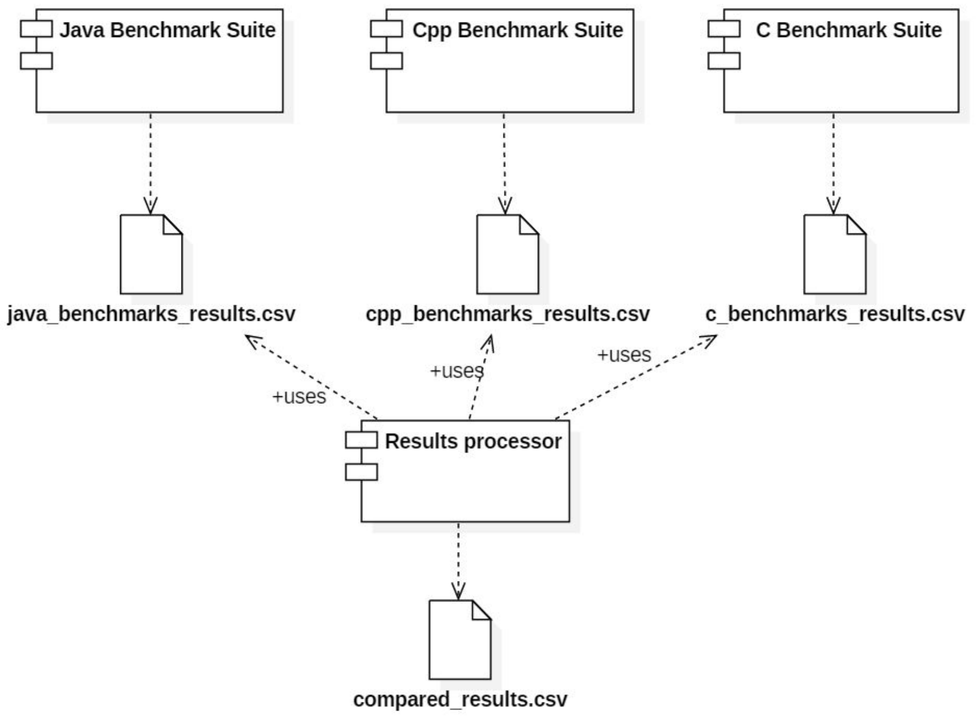
Java does not have native support for processor affinity but it can be set in two ways: on Linux, using the `taskset` command, or by using an open-source library, like Java-Thread-Affinity.

A thread could be forced at some point to change the core it runs on and measure the time needed to perform this as a difference of two timestamps.

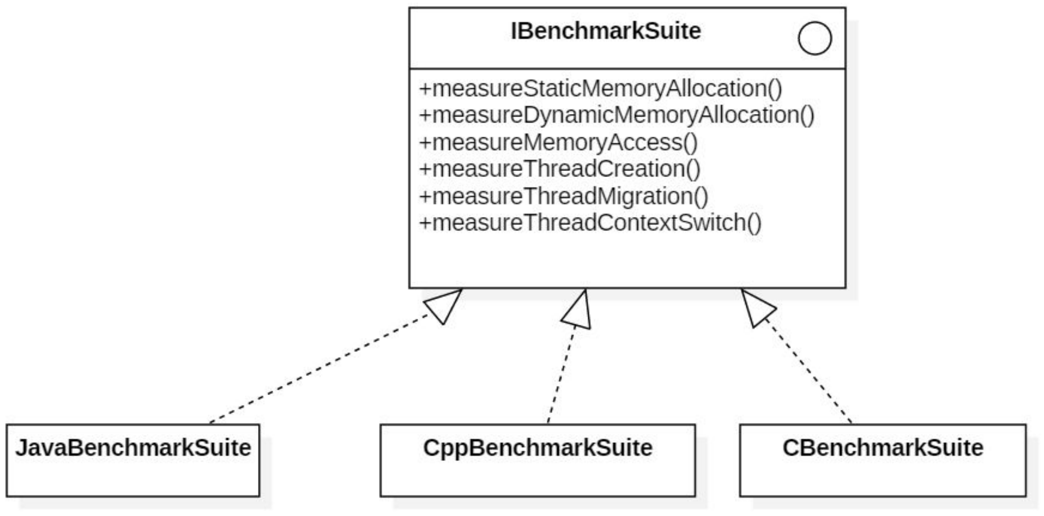
5. Design

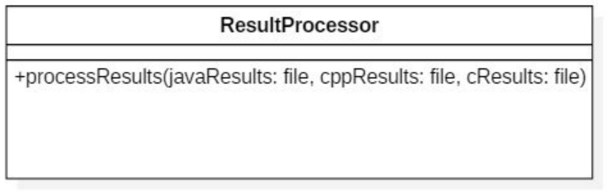
- Component diagram

The system will be composed of 3 programs that implement the same set of tests in each language chosen (Java, C++, C). Each program will write a .csv file containing the results of the microbenchmarks. Then, another module will process the data from the 3 files and write a final .csv file that will group the results by the test that was performed, so the performance of the processes in the 3 programming languages can be easily compared.

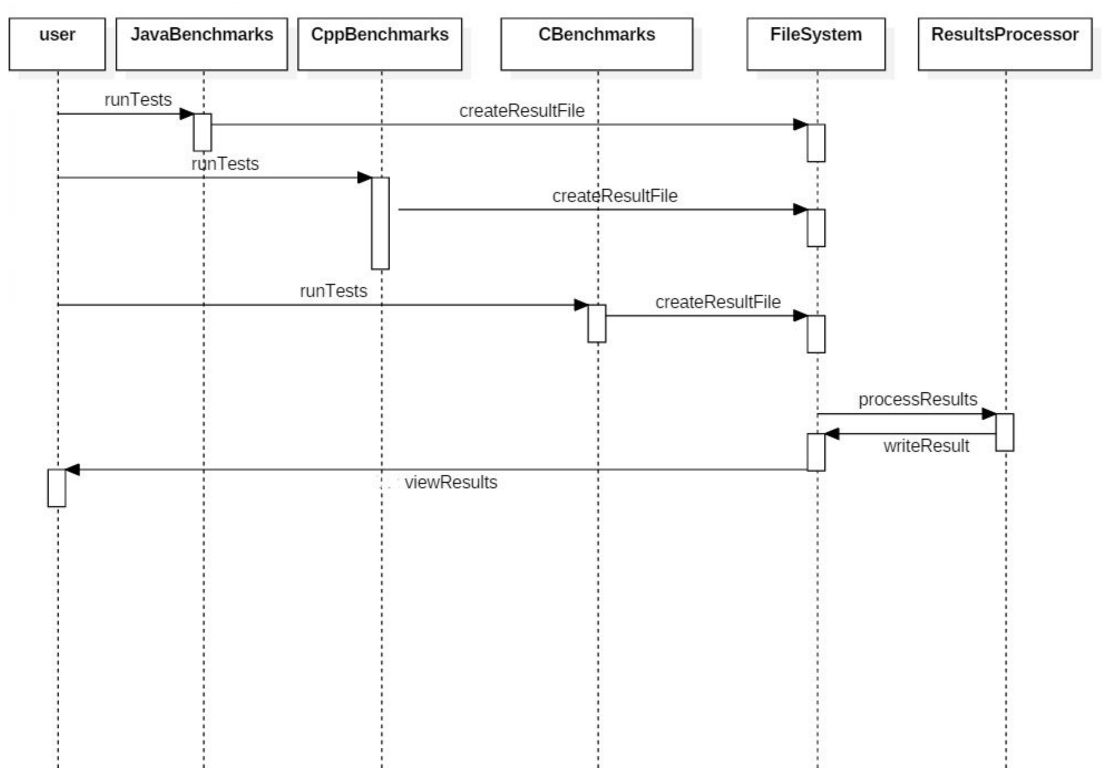


- Class diagram
  Each program will implement the functions described in the IBenchmarkSuite interface.


IBenchmarkSuite
+measureStaticMemoryAllocation()
+measureDynamicMemoryAllocation()
+measureMemoryAccess()
+measureThreadCreation()
+measureThreadMigration()
+measureThreadContextSwitch()
JavaBenchmarkSuite
CppBenchmarkSuite
CBenchmarkSuite


ResultProcessor
+processResults(javaResults: file, cppResults: file, cResults: file)

- Sequence diagram



6. Bibliography

- *Java theory and practice: Anatomy of a flawed microbenchmark*, Brian Goetz,
https://www.ibm.com/developerworks/java/library/j-jtp02225/
- *Java theory and practice: Dynamic compilation and performance measurement*, Brian Goetz, https://www.ibm.com/developerworks/library/j-jtp12214/
- https://stackoverflow.com/questions/504103/how-do-i-write-a-correct-micro-benchmark-in-java
- https://stackoverflow.com/questions/2842695/what-is-microbenchmarking
- http://www.cplusplus.com/reference/thread/thread/
- https://www.techopedia.com/definition/27492/memory-allocation
- https://software.intel.com/en-us/vtune-amplifier-cookbook-os-thread-migration#IDENTIFY
- https://stackoverflow.com/questions/304752/how-to-estimate-the-thread-context-switching-overhead
- https://www.pluralsight.com/blog/software-development/how-to-measure-execution-time-intervals-in-c--
- https://github.com/emilk/ram_bench

- https://dzone.com/articles/stack-vs-heap-understanding-java-memory-allocation
- http://www.vogella.com/tutorials/JavaPerformance/article.html
- https://www.mkyong.com/java/java-write-directly-to-memory/
- https://stackoverflow.com/questions/304752/how-to-estimate-the-thread-context-switching-overhead
- *The Component Diagram,* Donald Bell, https://www.ibm.com/developerworks/rational/library/dec04/bell/index.html