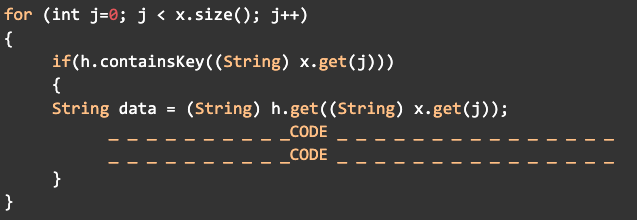
Java pplication running a lot of resources

* Includes memory
* Network bandwidth
* No. of cycles

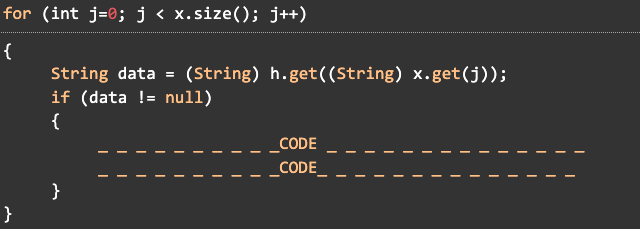
Minimizing the number of resources used by it called java performance tuning.

* Garbage collector does not remove the unused object present in heap memory resulting in poor performance of application.

1. Use Profiler
   * For checking each and every part of byte code
   * To determine inefficient part of code
   * Net beans profiler,Java Visual VM
2. Write less code
3. Premature optimization
4. Write efficient algorithms
   * Instead of using array , use hashmapt o increase efficiency.
5. Java heap
   * Java heap space sjpuld be expanded just up to ½ of the server total random access memory. If more then results in performance issues.
6. Tuning garbage collection
   * Keep an eye on outliers along the average time taken for transaction .
7. For concatenating string avoid using + operator . Use StringBuilder for string concatenation
8. Use primitives instead of Objects
   * Primitives are stored in stack and objects store in heap
   * Use primitives instead of Wrapper class
9. Accessing collection should require as few method as possible



Best way



1. Be care ful in using synchronization

**Try-with-resources Statement**

The try-with-resources statement is a try statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.

The following example reads the first line from a file. It uses an instance of FileReader and BufferedReader to read data from the file. FileReader and BufferedReader are resources that must be closed after the program is finished with it:

static String readFirstLineFromFile(String path) throws IOException {  
 **try (FileReader fr = new FileReader(path);  
 BufferedReader br = new BufferedReader(fr))** {  
 return br.readLine();  
 }  
}

**Multiple Catch Block in Java**

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in the catch block.

Catching multiple exceptions in a single catch block reduces code duplication and increases efficiency. The bytecode generated while compiling this program will be smaller than the program having multiple catch blocks as there is no code redundancy.

**Syntax:**

try {   
 // code  
}   
catch (ExceptionType1 | Exceptiontype2 ex){   
 // catch block  
}

**Important Points:**

**1.**If all the exceptions belong to the same class hierarchy, we should be catching the base exception type. However, to catch each exception, it needs to be done separately in their catch blocks.

**2.**Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions can not be caught in one statement. For Example

// Not Valid as Exception is an ancestor of   
// NumberFormatException  
catch([NumberFormatException](https://docs.oracle.com/javase/7/docs/api/java/lang/NumberFormatException.html" \t "_blank) | Exception ex)

**3.**All the exceptions must be separated by vertical bar pipe **|**.

**Streams over Loops**

**Streams are a more declarative style**. Or a more **expressive** style. It may be considered better to declare your intent in code, than to describe *how* it’s done:

return people  
 .filter( p -> p.age() < 19)  
 .collect(toList());

… says quite clearly that you’re filtering matching elements from a list, whereas:

List<Person> filtered = new ArrayList<>();  
 for(Person p : people) {  
 if(p.age() < 19) {  
 filtered.add(p);  
 }  
 }  
 return filtered;

Says “I’m doing a loop”. The purpose of the loop is buried deeper in the logic.

Streams are often **terser**. The same example shows this. Terser isn’t always better, but if you can be terse and expressive at the same time, so much the better.

**Streams have a strong affinity with functions**. Java 8 introduces lambdas and functional interfaces, which opens a whole toybox of powerful techniques. Streams provide the most convenient and natural way to apply functions to sequences of objects.

**Streams encourage less mutability**. This is sort of related to the functional programming aspect — the kind of programs you write using streams tend to be the kind of programs where you don’t modify objects.

**Streams encourage looser coupling**. Your stream-handling code doesn’t need to know the source of the stream, or its eventual terminating method.

**Streams can succinctly express quite sophisticated behaviour**. For example:

stream.filter(myfilter).findFirst();

Might look at first glance as if it filters the whole stream, then returns the first element. But in fact findFirst() drives the whole operation, so it efficiently stops after finding one item.

**Streams provide scope for future efficiency gains**. Some people have benchmarked and found that single-threaded streams from in-memory Lists or arrays can be slower than the equivalent loop. This is plausible because there are more objects and overheads in play.

But streams scale. As well as Java’s built-in support for parallel stream operations, there are a few libraries for distributed map-reduce using Streams as the API, because the model fits.

**Disadvantages?**

**Performance**: A for loop through an array is extremely lightweight both in terms of heap and CPU usage. If raw speed and memory thriftiness is a priority, using a stream is worse.

**Familiarity**.The world is full of experienced procedural programmers, from many language backgrounds, for whom loops are familiar and streams are novel. In some environments, you want to write code that’s familiar to that kind of person.

**Cognitive overhead**. Because of its declarative nature, and increased abstraction from what’s happening underneath, you may need to build a new mental model of how code relates to execution. Actually you only need to do this when things go wrong, or if you need to deeply analyse performance or subtle bugs. When it “just works”, it just works.

**Debuggers** are improving, but even now, when you’re stepping through stream code in a debugger, it can be harder work than the equivalent loop, because a simple loop is very close to the variables and code locations that a traditional debugger works with.

**Parallel Streams vs Streams**

There are no guarantees that executing a stream in parallel will improve performance. This section looks at some factors that can affect performance.

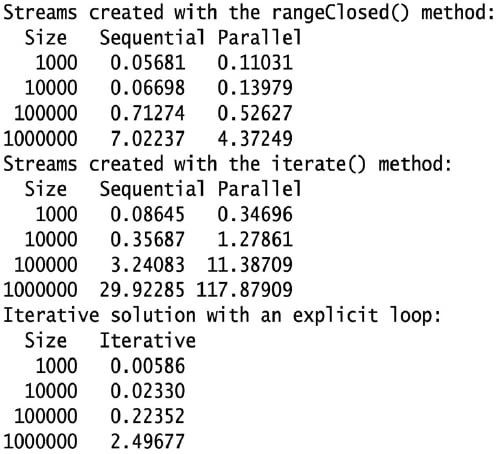
In general, increasing the number of CPU cores and, thereby, the number of threads that can execute in parallel scales performance only up to a threshold for a given size of data, as some threads might become idle if there is no data left for them to process. The number of CPU cores boosts performance to a certain extent, but it is not the only factor that should be considered when deciding whether to execute a stream in parallel.

Inherent in the total cost of parallel processing is the startup cost of setting up the parallel execution. At the onset, if this cost is already comparable to the cost of sequential execution, not much can be gained by resorting to parallel execution.

A combination of the following three factors can be crucial in deciding whether a stream should be executed in parallel:

* **Sufficiently large data size.** The size of the stream must be sufficiently large enough to warrant parallel processing; otherwise, sequential processing is preferable. The startup cost can be too prohibitive for parallel execution if the stream size is too small.
* **Computation-intensive stream operations.**If the stream operations are small computations, the stream size should be proportionately large to warrant parallel execution. If the stream operations are computation-intensive, the stream size is less significant, and parallel execution can boost performance.
* **Easily splitable stream.** If the cost of splitting the stream into substreams is higher than the cost of processing the substreams, employing parallel execution can be futile. A collection such as an ArrayList, a HashMap, or a simple array are efficiently splitable, whereas a LinkedList or I/O-based datasources are least efficient in this regard.

Benchmarking — that is, measuring performance — is strongly recommended before deciding whether parallel execution will be beneficial.



**Figure 1.** Output from the benchmark program