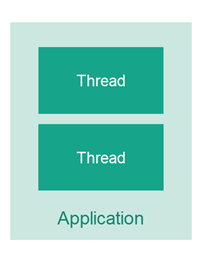
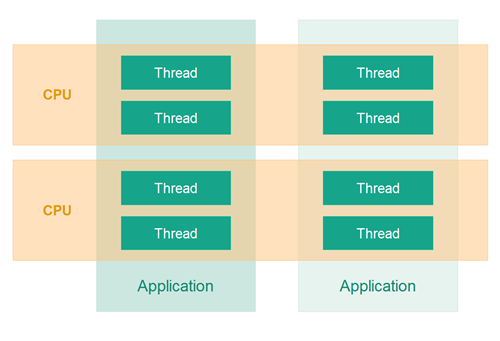
What is Multithreading?

Multithreading means that you have multiple threads of execution inside the same application. A thread is like a separate CPU executing your application. Thus, a multithreaded application is like an application that has multiple CPUs executing different parts of the code at the same time.



A thread is not equal to a CPU though. Usually a single CPU will share its execution time among multiple threads, switching between executing each of the threads for a given amount of time. It is also possible to have the threads of an application be executed by different CPUs.



**Why Multithreading?**

There are several reasons as to why one would use multithreading in an application. Some of the most common reasons for multithreading are:

1. Better utilization of a single CPU.
2. Better utilization of multiple CPUs or CPU cores.
3. Better user experience with regards to responsiveness.
4. Better user experience with regards to fairness.

I will explain each of these reasons in more detail in the following sections.

**Better Utilization of a Single CPU**

One of the most common reasons is to be able to better utilize the resources in the computer. For instance, if one thread is waiting for the response to a request sent over the network, then another thread could use the CPU in the meantime to do something else. Additionally, if the computer has multiple CPUs, or if the CPU has multiple execution cores, then multithreading can also help your application utilize these extra CPU cores.

**Better Utilization of Multiple CPUs or CPU Cores**

If a computer contains multiple CPUs or the CPU contains multiple execution cores, then you need to use multiple threads for your application to be able to utilize all of the CPUs or CPU cores. A single thread can at most utilize a single CPU, and as I mentioned above, sometimes not even completely utilize a single CPU.

**Better User Experience with Regards to Responsiveness**

Another reason to use multithreading is to provide a better user experience. For instance, if you click on a button in a GUI and this results in a request being sent over the network, then it matters which thread performs this request. If you use the same thread that is also updating the GUI, then the user might experience the GUI "hanging" while the GUI thread is waiting for the response for the request. Instead, such a request could be performed by a backgroun thread so the GUI thread is free to respond to other user requests in the meantime.

**Better User Experience with Regards to Fairness**

A fourth reason is to share resources of a computer more fairly among users. As example imagine a server that receives requests from clients, and only has one thread to execute these requests. If a client sends a requests that takes a long time to process, then all other client's requests would have to wait until that one request has finished. By having each client's request executed by its own thread then no single task can monopolize the CPU completely.

**Multithreading vs. Multitasking**

Back in the old days a computer had a single CPU, and was only capable of executing a single program at a time. Most smaller computers were not really powerful enough to execute multiple programs at the same time, so it was not attempted. To be fair, many mainframe systems (corporate data imcenter)have been able to execute multiple programs at a time for many more years than personal computers.

**Multitasking**

Later came multitasking which meant that computers could execute multiple programs (AKA tasks or processes) at the same time. It wasn't really "at the same time" though. The single CPU was shared between the programs. The operating system would switch between the programs running, executing each of them for a little while before switching.

Along with multitasking came new challenges for software developers. Programs can no longer assume to have all the CPU time available, nor all memory or any other computer resources. A "good citizen" program should release all resources it is no longer using, so other programs can use them.

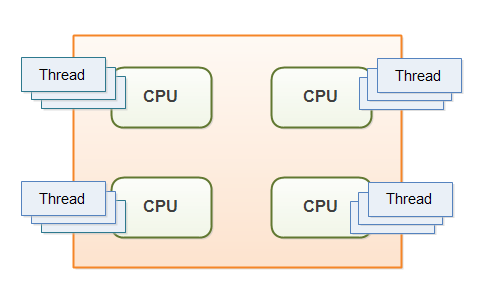
**Multithreading**

Later yet came multithreading which mean that you could have multiple threads of execution inside the same program. A thread of execution can be thought of as a CPU executing the program. When you have multiple threads executing the same program, it is like having multiple CPUs execute within the same program.

**Multithreading is Hard**

Multithreading can be a great way to increase the performance of some types of programs. However, mulithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultaneously. This can result in errors not seen in a single threaded program. Some of these errors may not be seen on single CPU machines, because two threads never really execute "simultaneously". Modern computers, though, come with multi core CPUs, and even with multiple CPUs too. This means that separate threads can be executed by separate cores or CPUs simultaneously.

Multithreading on a multi-CPU computer



If a thread reads a memory location while another thread writes to it, what value will the first thread end up reading? The old value? The value written by the second thread? Or a value that is a mix between the two? Or, if two threads are writing to the same memory location simultaneously, what value will be left when they are done? The value written by the first thread? The value written by the second thread? Or a mix of the two values written?

Without proper precautions any of these outcomes are possible. The behaviour would not even be predictable. The outcome could change from time to time. Therefore it is important as a developer to know how to take the right precautions - meaning learning to control how threads access shared resources like memory, files, databases etc. That is one of the topics this Java concurrency tutorial addresses.

**Multithreading and Concurrency in Java**

Java was one of the first languages to make multithreading easily available to developers. Java had multithreading capabilities from the very beginning. Therefore, Java developers often face the problems described above. That is the reason I am writing this trail on Java concurrency. As notes to myself, and any fellow Java developer whom may benefit from it.

The trail will primarily be concerned with multithreading in Java, but some of the problems occurring in multithreading are similar to problems occurring in multitasking and in distributed systems. References to multitasking and distributed systems may therefore occur in this trail too. Hence the word "concurrency" rather than "multithreading".

**Concurrency Models**

The first Java concurrency model assumed that multiple threads executing within the same application would also share objects. This type of concurrency model is typically referred to as a "shared state concurrency model". A lot of the concurrency language constructs and utilities are designed to support this concurrency model.

However, a lot has happened in the world of concurrent architecture and design since the first Java concurrency books were written, and even since the Java 5 concurrency utilities were released

The shared state concurrency model causes a lot of concurrency problems which can be hard to solve elegantly. Therefore, an alternative concurrency model referred to as "shared nothing" or "separate state" has gained popularity. In the separate state concurrency model the threads do not share any objects or data. This avoids a lot of the concurrent access problems of the shared state concurrency model.

New, asynchronous "separate state" platforms and toolkits like Netty, Vert.x and Play / Akka and Qbit have emerged. New non-blocking concurrency algorithms have been published, and new non-blocking tools like the LMax Disrupter have been added to our toolkits. New functional programming parallelism has been introduced with the Fork and Join framework in Java 7, and the collection streams API in Java 8.

With all these new developments it is about time that I updated this Java Concurrency tutorial. Therefore, this tutorial is once again work in progress. New tutorials will be published whenever time is available to write them.

The most significant benefits of multithreading are:

1. Better CPU utilization.
2. Simpler program design in some situations.
3. More responsive programs.
4. More fair division of CPU resources between different tasks.

**Better CPU Utilization**

Imagine an application that reads and processes files from the local file system. Lets say that reading a file from disk takes 5 seconds and processing it takes 2 seconds. Processing two files then takes

5 seconds reading file A

2 seconds processing file A

5 seconds reading file B

2 seconds processing file B

-----------------------

14 seconds total

When reading the file from disk most of the CPU time is spent waiting for the disk to read the data. The CPU is pretty much idle during that time. It could be doing something else. By changing the order of the operations, the CPU could be better utilized. Look at this ordering:

5 seconds reading file A

5 seconds reading file B + 2 seconds processing file A

2 seconds processing file B

-----------------------

12 seconds total

The CPU waits for the first file to be read. Then it starts the read of the second file. While the second file is being read in by the IO components of the computer, the CPU processes the first file. Remember, while waiting for the file to be read from disk, the CPU is mostly idle.

In general, the CPU can be doing other things while waiting for IO. It doesn't have to be disk IO. It can be network IO as well, or input from a user at the machine. Network and disk IO is often a lot slower than CPU's and memory IO.

**Simpler Program Design**

If you were to program the above ordering of reading and processing by hand in a singlethreaded application, you would have to keep track of both the read and processing state of each file. Instead you can start two threads that each just reads and processes a single file. Each of these threads will be blocked while waiting for the disk to read its file. While waiting, other threads can use the CPU to process the parts of the file they have already read. The result is, that the disk is kept busy at all times, reading from various files into memory. This results in a better utilization of both the disk and the CPU. It is also easier to program, since each thread only has to keep track of a single file.

**More Responsive Programs**

Another common goal for turning a singlethreaded application into a multithreaded application is to achieve a more responsive application. Imagine a server application that listens on some port for incoming requests. when a request is received, it handles the request and then goes back to listening. The server loop is sketched below:

while(server is active){

listen for request

process request

}

If the request takes a long time to process, no new clients can send requests to the server for that duration. Only while the server is listening can requests be received.

An alternate design would be for the listening thread to pass the request to a worker thread, and return to listening immediately. The worker thread will process the request and send a reply to the client. This design is sketched below:

while(server is active){

listen for request

hand request to worker thread

}

This way the server thread will be back at listening sooner. Thus more clients can send requests to the server. The server has become more responsive.

The same is true for desktop applications. If you click a button that starts a long task, and the thread executing the task is the thread updating the windows, buttons etc., then the application will appear unresponsive while the task executes. Instead, the task can be handed off to a worker thread. While the worker thread is busy with the task, the window thread is free to respond to other user requests. When the worker thread is done it signals the window thread. The window thread can then update the application windows with the result of the task. The program with the worker thread design will appear more responsive to the user.

**More Fair Distribution of CPU Resources**

Imagine a server that is receiving requests from clients. Imagine then, that one of the clients sends a request that takes a long time to process - e.g. 10 seconds. If the server processed all tasks using a single thread, then all requests following the request that was slow to process would be forced to wait until the full request has been processed.

By dividing the CPU time between multiple threads and switching between the threads, then the CPU can share its execution time more fairly between several requests. Then even if one of the requests is slow, other requests that are faster to process can be executed concurrently with the slower request. Of course this means that executing the slow request will be even slower, since it will not have the CPU solely allocated to processing just it. However, the other requests will have to wait shorter time to be processed, because they do not have to wait for the slow tasks to finish before they can be processed. In the event that there is only the slow request to process, then the CPU can still be solely allocated to the slow task.

**Multithreading Costs**

1. More complex design
2. Context Switching Overhead

Going from a singlethreaded to a multithreaded application doesn't just provide benefits. It also has some costs. Don't just multithread-enable an application just because you can. You should have a well-founded expectation that the benefits gained by doing so are larger than the costs. When in doubt, try measuring the performance or responsiveness of the application, instead of just guessing.

**More complex design**

Though some parts of a multithreaded applications are simpler than a singlethreaded application, other parts are more complex. Code executed by multiple threads accessing shared data need special attention. Thread interaction is far from always simple. Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix.

**Context Switching Overhead**

When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.

Context switching isn't cheap. You don't want to switch between threads more than necessary.

**Increased Resource Consumption**

A thread needs some resources from the computer in order to run. Besides CPU time a thread needs some memory to keep its local stack. It may also take up some resources inside the operating system needed to manage the thread. Try creating a program that creates 100 threads that does nothing but wait, and see how much memory the application takes when running.

# **Concurrency Models**

* Concurrency Models and Distributed System Similarities
* Shared State vs. Separate State
* Parallel Workers
* Parallel Workers Advantages
* Parallel Workers Disadvantages

1. Shared State Can Get Complex
2. Stateless Workers
3. Job Ordering is Nondeterministic

* Assembly Line

1. Reactive, Event Driven Systems
2. Actors vs. Channels

* Assembly Line Advantages

1. No Shared State
2. Stateful Workers
3. Better Hardware Conformity
4. Job Ordering is Possible

* Assembly Line Disadvantages
* Functional Parallelism
* Which Concurrency Model is Best?

Concurrent systems can be implemented using different concurrency models. A concurrency model specifies how threads in the the system collaborate to complete the tasks they are are given. Different concurrency models split the tasks in different ways, and the threads may communicate and collaborate in different ways. This concurrency model tutorial will dive a bit deeper into the most popular concurrency models in use at the time of writing (2015 - 2019).

**Concurrency Models and Distributed System Similarities**

The concurrency models described in this text are similar to different architectures used in distributed systems. In a concurrent system different threads communicate with each other. In a distributed system different processes communicate with each other (possibly on different computers). Threads and processes are quite similar to each other in nature. That is why the different concurrency models often look similar to different distributed system architectures.

Of course distributed systems have the extra challenge that the network may fail, or a remote computer or process is down etc. But a concurrent system running on a big server may experience similar problems if a CPU fails, a network card fails, a disk fails etc. The probability of failure may be lower, but it can theoretically still happen.

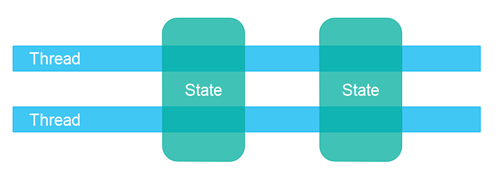
Because concurrency models are similar to distributed system architectures, they can often borrow ideas from each other. For instance, models for distributing work among workers (threads) are often similar to models of load balancing in distributed systems. The same is true of error handling techniques like logging, fail-over, idempotency of tasks etc.

**Shared State vs. Separate State**

One important aspect of a concurrency model is, whether the components and threads are designed to share state among the threads, or to have separate state which is never shared among the threads.

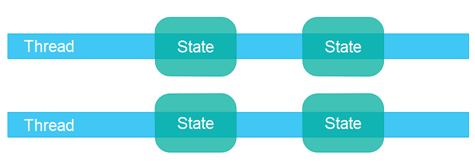
Shared state means that the different threads in the system will share some state among them. By state is meant some data, typically one or more objects or similar. When threads share state, problems like race conditions and deadlock etc. may occur. It depends on how the threads use and access the shared objects, of course.

Two threads with shared state.



Separate state means that the different threads in the system do not share any state among them. In case the different threads need to communicate, they do so either by exchanging immutable objects among them, or by sending copies of objects (or data) among them. Thus, when no two threads write to the same object (data / state), you can avoid most of the common concurrency problems.

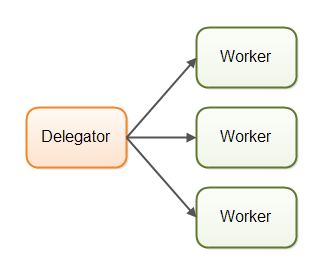
Two threads with separate state.



Using a separate state concurrency design can often make some parts of the code easier to implement and easier to reason about, since you know that only one thread will ever write to a given object. You don't have to worry about concurrent access to that object. However, you might have to think a bit harder about the application design in the big picture, to use separate state concurrency. It's worth it though, I feel. Personally I prefer separate state concurrency designs.

## Parallel Workers

The first concurrency model is what I call the *parallel workers* model. Incoming jobs are assigned to different workers. Here is a diagram illustrating the parallel workers concurrency model:



In the parallel workers concurrency model a delegator distributes the incoming jobs to different workers. Each worker completes the full job. The workers work in parallel, running in different threads, and possibly on different CPUs.

If the parallel workers model was implemented in a car factory, each car would be produced by one worker. The worker would get the specification of the car to build, and would build everything from start to end.

The parallel workers concurrency model is the most commonly used concurrency model in Java applications (although that is changing). Many of the concurrency utilities in the **[java.util.concurrent Java package](https://jenkov.com/java-util-concurrent/index.html)** are designed for use with this model. You can also see traces of this model in the design of the Java Enterprise Edition application servers.

The parallel workers concurrency model can be designed to use both shared state or separate state, meaning the workers either has access to some shared state (shared objects or data), or they have no shared state.

## Parallel Workers Advantages

The advantage of the parallel workers concurrency model is that it is easy to understand. To increase the parallelization level of the application you just add more workers.

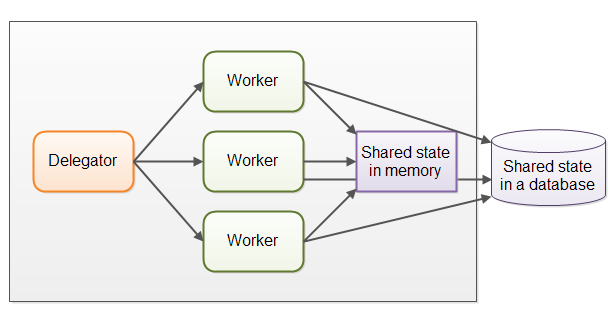
For instance, if you were implementing a web crawler, you could crawl a certain amount of pages with different numbers of workers and see which number gives the shortest total crawl time (meaning the highest performance). Since web crawling is an IO intensive job you will probably end up with a few threads per CPU / core in your computer. One thread per CPU would be too little, since it would be idle a lot of the time while waiting for data to download.

## Parallel Workers Disadvantages

The parallel workers concurrency model has some disadvantages lurking under the simple surface, though. I will explain the most obvious disadvantages in the following sections.

### **Shared State Can Get Complex**

In case the shared workers need access to some kind of shared data, either in memory or in a shared database, managing correct concurrent access can get complex. The following diagram shows how this complicates the parallel worker concurrency model:



Some of this shared state is in communication mechanisms like job queues. But some of this shared state is business data, data caches, connection pools to the database etc.

As soon as shared state sneaks into the parallel workers concurrency model it starts getting complicated. The threads need to access the shared data in a way that makes sure that changes by one thread are visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU executing the thread). Threads need to avoid [**race conditions**](https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html), [**deadlock**](https://jenkov.com/tutorials/java-concurrency/deadlock.html) and many other shared state concurrency problems.

Additionally, part of the parallelization is lost when threads are waiting for each other when accessing the shared data structures. Many concurrent data structures are blocking, meaning one or a limited set of threads can access them at any given time. This may lead to contention on these shared data structures. High contention will essentially lead to a degree of serialization of execution (eliminating parallelization) of the part of the code that access the shared data structures.

Modern [**non-blocking concurrency algorithms**](https://jenkov.com/tutorials/java-concurrency/non-blocking-algorithms.html) may decrease contention and increase performance, but non-blocking algorithms are hard to implement.

Persistent data structures are another alternative. A persistent data structure always preserves the previous version of itself when modified. Thus, if multiple threads point to the same persistent data structure and one thread modifies it, the modifying thread gets a reference to the new structure. All other threads keep a reference to the old structure which is still unchanged and thus consistent. The Scala standard APIs contain several persistent data structures.

While persistent data structures are an elegant solution to concurrent modification of shared data structures, persistent data structures tend not to perform that well.

For instance, a persistent list will add all new elements to the head of the list, and return a reference to the newly added element (which then point to the rest of the list). All other threads still keep a reference to the previously first element in the list, and to these threads the list appear unchanged. They cannot see the newly added element.

Such a persistent list is implemented as a linked list. Unfortunately linked lists don't perform very well on modern hardware. Each element in the list is a separate object, and these objects can be spread out all over the computer's memory. Modern CPUs are much faster at accessing data sequentially, so on modern hardware you will get a lot higher performance out of a list implemented on top of an array. An array stores data sequentially. The CPU caches can load bigger chunks of the array into the cache at a time, and have the CPU access the data directly in the CPU cache once loaded. This is not really possible with a linked list where elements are scattered all over the RAM.

### **Stateless Workers**

Shared state can be modified by other threads in the system. Therefore workers must re-read the state every time they need it, to make sure they are working on the latest copy. This is true no matter whether the shared state is kept in memory or in an external database. A worker that does not keep state internally (but re-reads it every time it is needed) is called *stateless* .

Re-reading data every time you need it can get slow. Especially if the state is stored in an external database.

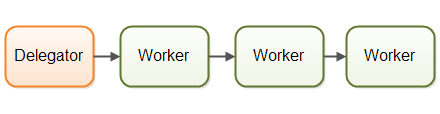
### **Job Ordering is Nondeterministic**

Another disadvantage of the parallel worker model is that the job execution order is nondeterministic. There is no way to guarantee which jobs are executed first or last. Job A may be given to a worker before job B, yet job B may be executed before job A.

The nondeterministic nature of the parallel worker model makes it hard to reason about the state of the system at any given point in time. It also makes it harder (if not impossible) to guarantee that one task finishes before another. This does not always cause problems, however. It depends on the needs of the system.

## Assembly Line

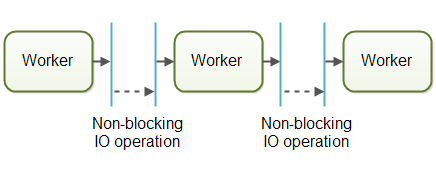
The second concurrency model is what I call the *assembly line* concurrency model. I chose that name just to fit with the "parallel worker" metaphor from earlier. Other developers use other names (e.g. reactive systems, or event driven systems) depending on the platform / community. Here is a diagram illustrating the assembly line concurrency model:



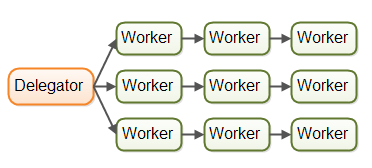
The workers are organized like workers at an assembly line in a factory. Each worker only performs a part of the full job. When that part is finished the worker forwards the job to the next worker.

Systems using the assembly line concurrency model are usually designed to use non-blocking IO. Non-blocking IO means that when a worker starts an IO operation (e.g. reading a file or data from a network connection) the worker does not wait for the IO call to finish. IO operations are slow, so waiting for IO operations to complete is a waste of CPU time. The CPU could be doing something else in the meanwhile. When the IO operation finishes, the result of the IO operation ( e.g. data read or status of data written) is passed on to another worker.

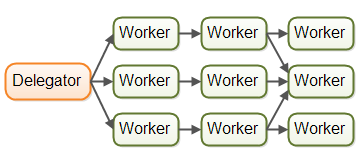
With non-blocking IO, the IO operations determine the boundary between workers. A worker does as much as it can until it has to start an IO operation. Then it gives up control over the job. When the IO operation finishes, the next worker in the assembly line continues working on the job, until that too has to start an IO operation etc.



In reality, the jobs may not flow along a single assembly line. Since most systems can perform more than one job, jobs flows from worker to worker depending on what part of the job that needs to be executed next. In reality there could be multiple different virtual assembly lines running on at the same time. This is how the job flow through an assembly line system might look in reality:



Jobs may even be forwarded to more than one worker for concurrent processing. For instance, a job may be forwarded to both a job executor and a job logger. This diagram illustrates how all three assembly lines finish off by forwarding their jobs to the same worker (the last worker in the middle assembly line):



The assembly lines can get even more complex than this.

### Reactive, Event Driven Systems

Systems using an assembly line concurrency model are also sometimes called *reactive systems*, or *event driven systems*. The system's workers react to events occurring in the system, either received from the outside world or emitted by other workers. Examples of events could be an incoming HTTP request, or that a certain file finished loading into memory etc.

At the time of writing, there are a number of interesting reactive / event driven platforms available, and more will come in the future. Some of the more popular ones seems to be:

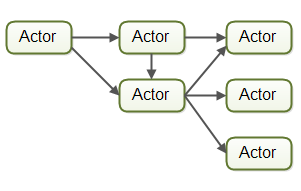
* [**Vert.x**](https://jenkov.com/vert.x/index.html)
* Akka
* Node.JS (JavaScript)

Personally I find Vert.x to be quite interesting (especially for a Java / JVM dinosaur like me).

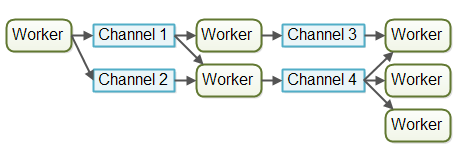
### Actors vs. Channels

Actors and channels are two similar examples of assembly line (or reactive / event driven) models.

In the actor model each worker is called an *actor*. Actors can send messages directly to each other. Messages are sent and processed asynchronously. Actors can be used to implement one or more job processing assembly lines, as described earlier. Here is a diagram illustrating the actor model:



In the channel model, workers do not communicate directly with each other. Instead they publish their messages (events) on different channels. Other workers can then listen for messages on these channels without the sender knowing who is listening. Here is a diagram illustrating the channel model:



At the time of writing, the channel model seems more flexible to me. A worker does not need to know about what workers will process the job later in the assembly line. It just needs to know what channel to forward the job to (or send the message to etc.). Listeners on channels can subscribe and unsubscribe without affecting the workers writing to the channels. This allows for a somewhat looser coupling between workers.

## Assembly Line Advantages

The assembly line concurrency model has several advantages compared to the parallel worker model. I will cover the biggest advantages in the following sections.

### No Shared State

The fact that workers share no state with other workers means that they can be implemented without having to think about all the concurrency problems that may arise from concurrent access to shared state. This makes it much easier to implement workers. You implement a worker as if it was the only thread performing that work - essentially a singlethreaded implementation.

### Stateful Workers

Since workers know that no other threads modify their data, the workers can be stateful. By stateful I mean that they can keep the data they need to operate in memory, only writing changes back the eventual external storage systems. A stateful worker can therefore often be faster than a stateless worker.

### Better Hardware Conformity

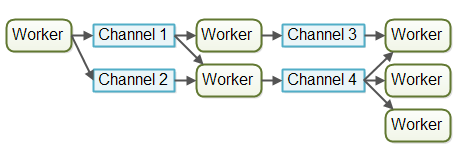
Singlethreaded code has the advantage that it often conforms better with how the underlying hardware works. First of all, you can usually create more optimized data structures and algorithms when you can assume the code is executed in single threaded mode.

Second, singlethreaded stateful workers can cache data in memory as mentioned above. When data is cached in memory there is also a higher probability that this data is also cached in the CPU cache of the CPU executing the thread. This makes accessing cached data even faster.

I refer to it as *hardware conformity* when code is written in a way that naturally benefits from how the underlying hardware works. Some developers call this *mechanical sympathy*. I prefer the term hardware conformity because computers have very few mechanical parts, and the word "sympathy" in this context is used as a metaphor for "matching better" which I believe the word "conform" conveys reasonably well. Anyways, this is nitpicking. Use whatever term you prefer.

### Job Ordering is Possible

It is possible to implement a concurrent system according to the assembly line concurrency model in a way that guarantees job ordering. Job ordering makes it much easier to reason about the state of a system at any given point in time. Furthermore, you could write all incoming jobs to a log. This log could then be used to rebuild the state of the system from scratch in case any part of the system fails. The jobs are written to the log in a certain order, and this order becomes the guaranteed job order. Here is how such a design could look:



Implementing a guaranteed job order is not necessarily easy, but it is often possible. If you can, it greatly simplifies tasks like backup, restoring data, replicating data etc. as this can all be done via the log file(s).

## Assembly Line Disadvantages

The main disadvantage of the assembly line concurrency model is that the execution of a job is often spread out over multiple workers, and thus over multiple classes in your project. Thus it becomes harder to see exactly what code is being executed for a given job.

It may also be harder to write the code. Worker code is sometimes written as callback handlers. Having code with many nested callback handlers may result in what some developer call *callback hell*. Callback hell simply means that it gets hard to track what the code is really doing across all the callbacks, as well as making sure that each callback has access to the data it needs.

With the parallel worker concurrency model this tends to be easier. You can open the worker code and read the code executed pretty much from start to finish. Of course parallel worker code may also be spread over many different classes, but the execution sequence is often easier to read from the code.

## Functional Parallelism

Functional parallelism is a third concurrency model which is being talked about a lot these days (2015).

The basic idea of functional parallelism is that you implement your program using function calls. Functions can be seen as "agents" or "actors" that send messages to each other, just like in the assembly line concurrency model (AKA reactive or event driven systems). When one function calls another, that is similar to sending a message.

All parameters passed to the function are copied, so no entity outside the receiving function can manipulate the data. This copying is essential to avoiding race conditions on the shared data. This makes the function execution similar to an atomic operation. Each function call can be executed independently of any other function call.

When each function call can be executed independently, each function call can be executed on separate CPUs. That means, that an algorithm implemented functionally can be executed in parallel, on multiple CPUs.

With Java 7 we got the java.util.concurrent package contains the **[ForkAndJoinPool](https://jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html)** which can help you implement something similar to functional parallelism. With Java 8 we got parallel [**streams**](https://jenkov.com/java-collections/streams.html) which can help you parallelize the iteration of large collections. Keep in mind that there are developers who are critical of the ForkAndJoinPool (you can find a link to criticism in my ForkAndJoinPool tutorial).

The hard part about functional parallelism is knowing which function calls to parallelize. Coordinating function calls across CPUs comes with an overhead. The unit of work completed by a function needs to be of a certain size to be worth this overhead. If the function calls are very small, attempting to parallelize them may actually be slower than a singlethreaded, single CPU execution.

From my understanding (which is not perfect at all) you can implement an algorithm using an reactive, event driven model and achieve a breakdown of the work which is similar to that achieved by functional parallelism. With an even driven model you just get more control of exactly what and how much to parallelize (in my opinion).

Additionally, splitting a task over multiple CPUs with the overhead the coordination of that incurs, only makes sense if that task is currently the only task being executed by the the program. However, if the system is concurrently executing multiple other tasks (like e.g. web servers, database servers and many other systems do), there is no point in trying to parallelize a single task. The other CPUs in the computer are anyways going to be busy working on other tasks, so there is not reason to try to disturb them with a slower, functionally parallel task. You are most likely better off with an assembly line (reactive) concurrency model, because it has less overhead (executes sequentially in singlethreaded mode) and conforms better with how the underlying hardware works.

## Which Concurrency Model is Best?

So, which concurrency model is better?

As is often the case, the answer is that it depends on what your system is supposed to do. If your jobs are naturally parallel, independent and with no shared state necessary, you might be able to implement your system using the parallel worker model.

Many jobs are not naturally parallel and independent though. For these kinds of systems I believe the assembly line concurrency model has more advantages than disadvantages, and more advantages than the parallel worker model.

You don't even have to code all that assembly line infrastructure yourself. Modern platforms like **[Vert.x](https://jenkov.com/vert.x/index.html)** has implemented a lot of that for you. Personally I will be exploring designs running on top of platforms like Vert.x for my next projects. Java EE just doesn't have the edge anymore, I feel.