Concurrency Models

Jakob Jenkov
Last update: 2015-04-13

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

Concurrency Models and Distributed System Similarities

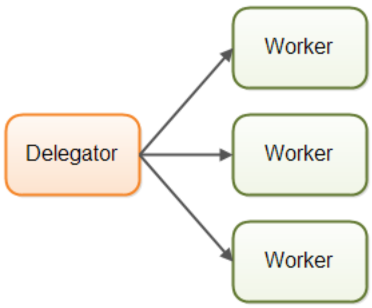
The concurrency models described in this text are similar to different architectures used in distribu systems. In a concurrent system different threads communicate with each other. In a distributed s different processes communicate with each other (possibly on different computers). Threads and processes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

Of course distributed systems have the extra challenge that the network may fail, or a remote con process is down etc. But a concurrent system running on a big server may experience similar prol 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 borro from each other. For instance, models for distributing work among workers (threads) are often sim models of **load balancing in distributed systems**. The same is true of error handling techniques logging, fail-over, idempotency of jobs etc.

Parallel Workers

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



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

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

The parallel worker 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.concurr package** are designed for use with this model. You can also see traces of this model in the design Java Enterprise Edition application servers.

Parallel Workers Advantages

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

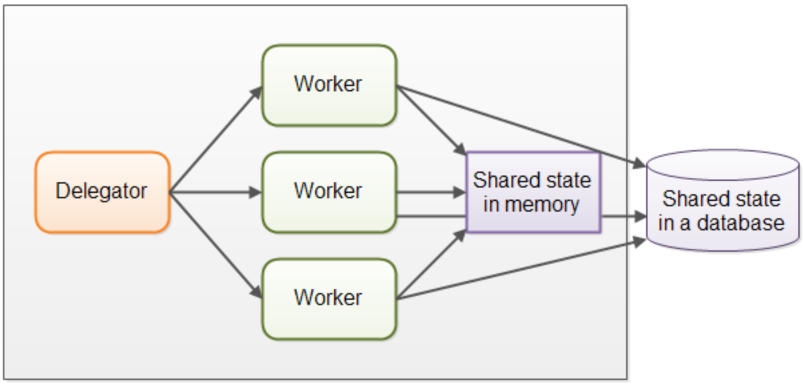
For instance, if you were implementing a web crawler, you could crawl a certain amount of pages different numbers of workers and see which number gives the shortest total crawl time (meaning t highest performance). Since web crawling is an IO intensive job you will probably end up with a fe threads per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

Parallel Workers Disadvantages

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

Shared State Can Get Complex

In reality the parallel worker concurrency model is a bit more complex than illustrated above. The workers often need access to some kind of shared data, either in memory or in a shared database 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 shar is business data, data caches, connection pools to the database etc.

As soon as shared state sneaks into the parallel worker concurrency model it starts getting compl The threads need to access the shared data in a way that makes sure that changes by one thread visible to the others (pushed to main memory and not just stuck in the CPU cache of the CPU exe the thread). Threads need to avoid **race conditions**, **deadlock** and many other shared state con problems.

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

Modern **non-blocking concurrency algorithms** may decrease contention and increase performa non-blocking algorithms are hard to implement.

Persistent data structures are another alternative. A persistent data structure always preserves th 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. A threads keep a reference to the old structure which is still unchanged and thus consistent. The Sc programming contains 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 referen newly added element (which then point to the rest of the list). All other threads still keep a referen previously first element in the list, and to these threads the list appear unchanged. They cannot se newly added element.

over the computer's memory. Modern CPUs are much faster at accessing data sequentially, so or hardware you will get a lot higher performance out of a list implemented on top of an array. An arr stores data sequentially. The CPU caches can load bigger chunks of the array into the cache at a and have the CPU access the data directly in the CPU cache once loaded. This is not really possi 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 every time it needs it, to make sure it is working on the latest copy. This is true no matter whether shared state is kept in memory or in an external database. A worker that does not keep state inte (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 externa database.

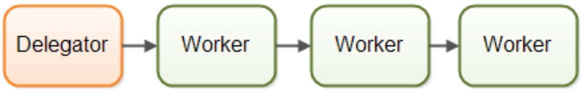
Job Ordering is Nondeterministic

Another disadvantage of the parallel worker model is that the job execution order is nondetermini There is no way to guarantee which jobs are executed first or last. Job A may be given to a worke 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 system at any given point in time. It also makes it harder (if not impossible) to guarantee that one happens before another.

Assembly Line

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

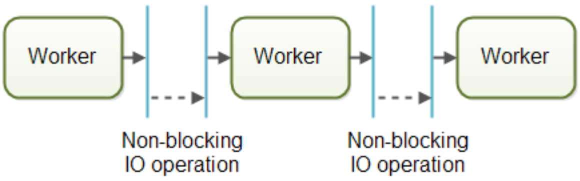


The workers are organized like workers at an assembly line in a factory. Each worker only perforn of the full job. When that part is finished the worker forwards the job to the next worker.

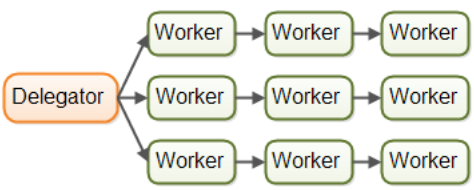
Each worker is running in its own thread, and shares no state with other workers. This is also som referred to as a *shared nothing* concurrency model.

Systems using the assembly line concurrency model are usually designed to use non-blocking IO blocking IO means that when a worker starts an IO operation (e.g. reading a file or data from a ne connection) the worker does not wait for the IO call to finish. IO operations are slow, so waiting fo 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 statu written) is passed on to another worker.

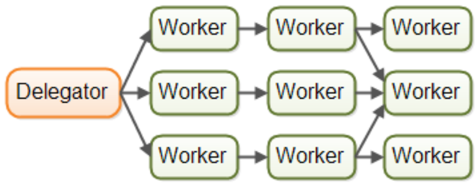
With non-blocking IO, the IO operations determine the boundary between workers. A worker does much as it can until it has to start an IO operation. Then it gives up control over the job. When the operation finishes, the next worker in the assembly line continues working on the job, until that to start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



In reality, the jobs may not flow along a single assembly line. Since most systems can perform mo one job, jobs flows from worker to worker depending on the job that needs to be done. In reality th could be multiple different virtual assembly lines going on at the same time. This is how job flow th assembly line system might look in reality:



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*, *driven systems*. The system's workers react to events occurring in the system, either received fro outside world or emitted by other workers. Examples of events could be an incoming HTTP reque 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, more will come in the future. Some of the more popular ones seems to be:

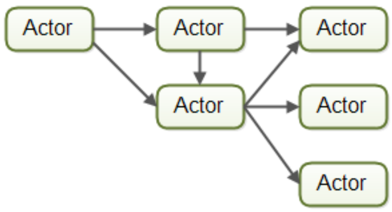
- **Vert.x**
- Akka
- Node.JS (JavaScript)

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

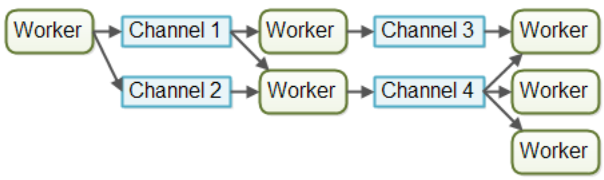
Actors vs. Channels

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 othe Messages are sent and processed asynchronously. Actors can be used to implement one or more 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 t messages (events) on different channels. Other workers can then listen for messages on these ch 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 kr about what workers will process the job later in the assembly line. It just needs to know what char forward the job to (or send the message to etc.). Listeners on channels can subscribe and unsubs without affecting the workers writing to the channels. This allows for a somewhat looser coupling I workers.

Assembly Line Advantages

The assembly line concurrency model has several advantages compared to the parallel worker m 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 with having to think about all the concurrency problems that may arise from concurrent access to shar This makes it much easier to implement workers. You implement a worker as if it was the only thr performing that work - essentially a singlethreaded implementation.

Stateful Workers

external storage systems. A stateful worker can therefore often be faster than a stateless worker.

Better Hardware Conformity

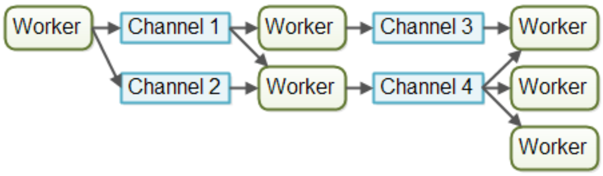
Singlethreaded code has the advantage that it often conforms better with how the underlying harc works. First of all, you can usually create more optimized data structures and algorithms when yo assume the code is executed in single threaded mode.

Second, singlethreaded stateful workers can cache data in memory as mentioned above. When d cached in memory there is also a higher probability that this data is also cached in the CPU cache 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 underlying hardware works. Some developers call this *mechanical sympathy*. I prefer the term ha conformity because computers have very few mechanical parts, and the word "sympathy" in this c used as a metaphor for "matching better" which I believe the word "conform" conveys reasonably 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 mode way that guarantees job ordering. Job ordering makes it much easier to reason about the state of system at any given point in time. Furthermore, you could write all incoming jobs to a log. This log then be used to rebuild the state of the system from scratch in case any part of the system fails. T are written to the log in a certain order, and this order becomes the guaranteed job order. Here is such a design could look:



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

Assembly Line Disadvantages

The main disadvantage of the assembly line concurrency model is that the execution of a job is o spread out over multiple workers, and thus over multiple classes in your project. Thus it becomes 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. H code with many nested callback handlers may result in what some developer call *callback hell*. Ca hell simply means that it gets hard to track what the code is really doing across all the callbacks, a 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 read the code executed pretty much from start to finish. Of course parallel worker code may also spread over many different classes, but the execution sequence is often easier to read from the c

Functional Parallelism

Functional parallelism is a third concurrency model which is being talked about a lot these days (2

The basic idea of functional parallelism is that you implement your program using function calls. F can be seen as "agents" or "actors" that send messages to each other, just like in the assembly li concurrency model (AKA reactive or event driven systems). When one function calls another, that 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. Thi the function execution similar to an atomic operation. Each function call can be executed indepen any other function call.

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

With Java 7 we got the `java.util.concurrent` package contains the **ForkAndJoinPool** which car you implement something similar to functional parallelism. With Java 8 we got parallel **streams** w help you parallelize the iteration of large collections. Keep in mind that there are developers who a 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. Coordinat function calls across CPUs comes with an overhead. The unit of work completed by a function ne be of a certain size to be worth this overhead. If the function calls are very small, attempting to pa 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 reac parallelize (in my opinion).

Additionally, splitting a task over multiple CPUs with the overhead the coordination of that incurs,
makes sense if that task is currently the only task being executed by the the program. However, if
system is concurrently executing multiple other tasks (like e.g. web servers, database servers and
other systems do), there is no point in trying to parallelize a single task. The other CPUs in the co
are anyways going to be busy working on other tasks, so there is not reason to try to disturb them
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)
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
naturally parallel, independent and with no shared state necessary, you might be able to impleme
system using the parallel worker model.

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

You don't even have to code all that assembly line infrastructure yourself. Modern platforms like **V**
has implemented a lot of that for you. Personally I will be exploring designs running on top of platf
like Vert.x for my next projects. Java EE just doesn't have the edge anymore, I feel.

Next: Same-threading

Jakob Jenkov