SCHOOL OF COMPUTING

UNIVERSITY OF TEESSIDE

MIDDLESBROUGH

TS1 3BA

Alternative Programming Approaches

BSc Computer Science

Kiril Anastasov

10 January 2014

Supervisor: Philips, Peter (Dr.)

Abstract

To begin with, the goal of this project is to undertake suitable research on properties such as isolated processes, pure message passing between processes, the ability to detect errors in remote processes, the ability to determine what error caused a process to crash and to write a report for a non-technical audience explaining how these properties assist in the production of robust applications suitable for today's multi-core computers. The report will also indicate how these features are implemented in Erlang and other similar programming languages.

Acknowledgements

I am grateful for the help and support of my supervisor Philips, Peter (Dr). Without the on-going support he gave me, it would not have been possible to complete the report. He was able to get in touch despite the time and place, in order to help for achieving all goals. He made the effort to personally get to know me and my project and give feedback where it was needed. Last but not least, he made me go the extra mile and taught me the habit of doing more than I am required to for which I will be always grateful.

By: Kiril Anastasov L1087591@live.tees.ac.uk

CONTENTS

Table of Figures

By: Kiril Anastasov L1087591@live.tees.ac.uk

Introduction

One of the most challenging responsibilities in the plan of a fault-tolerant machine is to validate that it will meet its reliability desires which requires creating a number of prototypes. The first prototype is of the error fault environment that is normal. Other models require the structure and performance of the design. It is then required to control how well the fault-tolerance mechanisms work by analytic studies and fault imitations. The results, in the form of error changes, fault-rates, latencies, and coverage, are used in reliability estimate reproductions all of which is achieved with Erlang programming language. A number of probabilistic reproductions have been established using Markov and semi-Markov methods to forecast the reliability of fault-tolerant machines as a meaning of interval. These prototypes have been implemented in numerous computer-aided design tools (Rennels, 2000).

Research

Erlang was designed for programming concurrent and easy to break systems at Ericsson. In 1986 Joe Armstrong created the first version of the language when it could be used only within Ericsson, but later after about 12 years in 1998 it was released as open source software. The name Erlang came after the Danish mathematician Agner Erlang, on the same principle as the programming language Pascal was named after the French mathematician Bless Pascal (Armstrong, 2007). Since the multicore processors started to gain popularity, the interest in Erlang was increasing proportionally because Erlang deals with multicore computers naturally. The processes in the language were created and managed not by the operating system like most of the languages but by Erlang runtime system. Erlang was used within Ericsson and was required to be a fault-tolerant, distributed, real-time application which was achieved through concurrency that belongs to Erlang, not as in the other languages to the operating system.

Ericsson had been building highly reliable fault-tolerant systems for years. There was at most four minutes of downtime in a year and hot code swapping which allows software to be replaced without stopping the system in Ericsson (Armstrong, 2010). There was already a template how to do it which was developed in the mid-1970s and has been without much change since then. The way Erlang was constructed was based on

By: Kiril Anastasov L1087591@live.tees.ac.uk

physically isolated components communicating through well-defined protocol, so there are no data references to data structure residing on other machines when the message has been passed.

The base hard to break technique used by Gray, J. in the design of the hard to break Tandem machine was used to build Erlang applications. When a machine stops working, the crash is noticed by another computer in the chain and the person who is using the service is not supposed to notice the failure of the network (Agha, 1986). The stock value of the company will fall substantially if half of the customers lose signal for just a few hours. Component isolation was the key to build reliable systems. When a component fails, the chance of all other components failing at the same time will be relatively small. This approach works because when a single process stops working due to a fatal software problem, the rest of the processes in the structure are not affected by the crash.

The problem that Armstrong had was that he was required to design and build telecom switching systems. Ericsson needed very reliable and fault-tolerant systems. Telecom systems are built in mind that they will work for decades, centuries a very long time and they are required to handle software and hardware errors in an appropriate manner. We are going to examine the main requirements for building and designing such a system. The first requirement is concurrency. For a telecom switch to operate well it needs to serve a lot of people at the same time, multithreaded and effectively take care of all the activities simultaneously. If the people need to wait in a queue then the process will be too slow. The second requirement is soft real-time. Most of the activities are required to be done with a deadline. Some of the important operations are important in a way that when an operation fails to be delivered for the deadline the whole operation will be stopped. Other secondary operations are checked with a timer, when the timer starts before the operation is done then the operation is started again. Programming such systems in an effective way can be very challenging even for experienced programmers. The third requirement is that the system is distributed. Telecom switching systems have a history of being distributed, so the system was required to be assembled in a manner that will allow going from a single-node to multi-node distributed system relatively easy. The fourth requirement is the hardware interaction. Telecom switching systems consist of big quantities of exterior hardware that is required to be supervisory and observing. This indicates writing effective drivers for the hardware and swapping between different hardware drivers should not be very difficult. The fifth requirement is large software systems. All of the Ericsson switches consist of hundreds of thousands of code which will require the software systems to work with a lot of code. The sixth requirement is complex functionality. Telecom switching systems consist of intricate functionality. The market can provide more pressure to boost the development of systems with a lot of complex structures. While a system is working the chances are that it will be reformed and stretched in different situations. Software changes and upgrades need to be done with hot code swapping or without disturbing the system.

By: Kiril Anastasov L1087591@live.tees.ac.uk

The seventh requirement is continuous operation. Telecom systems are built in mind to work for decades without stopping. This indicates that the hardware and the software operations should not be felt by the users. Eight is quality requirements. Even with a case of errors the telecom system should be working on decent levels and telephone interactions are predictable to be tremendously unfailing. The last requirement is fault tolerance. Even from the beginning we know that there are going to be some errors, and software and hardware architecture needs to be designed to prevent as much as possible the faults and provide service to the customers (Däcker, 2000).

Because in Erlang the system processes and the concurrency are part of the language and not delivered by the operating system there are a few advantages. The first advantage is that the concurrent programs run in the same manner on different operating systems and it does not matter in which way the processes are implemented in any operating system. The main difference between the operating systems may be seen with different CPU and memory. The second advantage is that Erlang based processes are much better than the ones in the operating system processes. Creating a new process in Erlang is very effective and faster than the most programming languages. Third, Erlang does not require much of the operating system and Erlang make use of a few operating services.

Erlang is structured using huge numbers of interactive similar processes because it can organize the system as a set of interactive processes. The second reason is the potential efficiency. Because Erlang is designed to be implemented as a number of independent multithreaded processes, it can use multi core processors more effectively. Last is fault isolation which is achieved through multithreaded processes with no data sharing between the processes. From the last three uses of concurrency only the last one is considered essential. The rest can be delivered with a scheduler that permits time allocation among processes. The third essential characteristic should be achieved in isolated process. The processes cannot share data and are permitted to communicate through message passing in order to limit the penalties of an error. When two processes share data like a pointer in the memory, there is a chance of corrupting the data (Armstrong, 2003). Armstrong solved the problem with independent comparable processes, and providing way to observe and restart the processes.

In Erlang, concurrency is very important that Armstrong made a new style called Concurrency Oriented Programming (COP) (Armstrong, 2010). This was done because he wanted to make a difference between Concurrency Oriented Programming (COP) and other languages like Java and Haskell. COP is very good to model and cooperate with the real world because it uses some of the properties of Java or Object Oriented Programming (OOP) which are polymorphism or many forms and overloading. What is meant by concurrency is that a number of events occur instantaneously. In a real world everything is happening concurrently and we are able to respond to all of the changes on subconscious level of our mind. For example when you are walking in the supermarket there are other people there and you can calculate (with relatively good

By: Kiril Anastasov L1087591@live.tees.ac.uk

precision) which way to walk in order not to push someone else. Almost everything in the real word in concurrent and there are rare things that are happening sequentially. In the computing world this is reversed. The sequential order is considered as normal and the concurrent state in trying to be escaped. This is happening because most of the languages do not offer good support for concurrency and the concurrency in the language is offered by the operating system and not by the language itself.

Concurrent Object Programming Languages (COPL) contain six major assets. First of all COPL are required to maintain processes. Every process has its own virtual machine. Secondly, when there is more than one process operating on the same computer the processes must be highly isolated. In case of an error in a process, should not affect another process, unless the agenda clearly command so. Another point is that each process is required to be made with unique id or Process Identifier (PID). Following, processes are required not to share states between other processes so they communicate by distribution of messages. Provided you know the PID of a certain process then you can communicate with it. Furthermore, message passing is done with unsecure transfer. Last Processes are able to find errors in other processes and the reasons for it as well. The aim of COPL is to deliver concurrency. This is achieved through objects characterized as processes and messages passing through processes are asynchronous (Armstrong, 2010).

Process isolation is one of the main principles of COPL and for achieving fault-tolerant systems. When there is more than one process on the same computer, they should be as if they are running on different machines. There are a few consequences of process isolation. First of all processes do not divide semantics because they are supposed to run on another computer. Second, message passing is the only way to send data among processes. This is the only way to send data because the processes do not share data. Third, isolation assumes that messages are sent concurrently. When the process message is synchronous then the sender of the message could be identified and dealt with it appropriately. Last because processes do not share data, everything that needs to perform distributed calculation needs to be derivative. There is no guarantee if a message has arrived because nothing is shared and there is no guarantee that the message has been received, unless you get a validation back that the message is direct the right way. Even if at first sight programming such a system may seem demanding, according to Armstrong (2003) programming a system according to the rules described above is easier than programming a system with locks, semaphores and shared data. Not only is it easier to program such a system but the system is scalable and fault-tolerant with reasonable effort. The architecture of Erlang allows us to add more processes and not to disturb the original system because the software grips groups of isolated processes. Because the system was written with the thought of not reliable message passing it is able to work in the presence of message transient mistakes (Armstrong, 2003).

By: Kiril Anastasov L1087591@live.tees.ac.uk

It is vital that the names of the processes are hidden, so it should be impossible to deduct the name of a process and communicate with it. Processes know their own names and the names of the processes that they have created like in the OOP class know the properties and have access to the properties of the sub-classes. In order for Erlang to work concurrently we need a way to trace the names of the processes so that we can send a message to that process. The more the name of a process is spread the more the security starts to decline. The route of revealing names to other processes in precise style is called name distribution problem where lies security of the system. Once the PID is known to another process then the process can communicate and the system is at risk.

Message passing is submitting 3 main rules. The first message passing is expected to be atomic or it is either sent completely or not at all. Following, a message passing is expected to be in order. If there is a number of messages that are being send they are supposed to be received in the same order. Furthermore, messages contain only names or PIDS and they are not supposed to contain indicators to data structures. When a message is sent we can hope that it has arrived. The only way to know that it has arrived is to receive a confirmation message. Message passing is used for synchronisation or casual ordering in distributed systems.

Isolation of components and message passing is architecturally adequate to keep from a software fault but it is not enough to require the conduct of a system or to find out which module has failed. Fully isolated components are essential to link by message passing and the communication protocols that are used needs to be stated among the components. By agreeing the communication protocol that should be minded between two mechanisms it can effortlessly be tested if any of the components involved has disrupted the protocol.

Error handling in Erlang is quite diverse to the rest of the languages. For example it uses the philosophy let some other process deal with the error. To prevent the failure of entire distribution of a system at least two machines are required. If there is error in the first machine the second one notices it and attempts to fix it. The same principle is used in Erlang but instead of machines it uses processes (Agha, 1986). So if the first process has some error then the second one notices the error and attempt to fix it. The Erlang perspective works in the opposite way as the one in the sequential programming languages. There the error is attempted to be handled with try and catch statement or something else depending on the case. In sequential languages like Java the code that might throw an exception is enclosed. The isolated handling of errors has a few compensations. First the code which has the error and the error-handling code are performed in diverse threads. Second the code which solved the error in not disordered with the code that uses the exception. Third the procedure works in a distributed system so when updating code it needs slight alteration to the error-handling part. Last it is possible that the systems are built and tested on a single node system but used on a multi-node system with a lot of changes (Armstrong, 2003).

Kiril Anastasov L1087591@live.tees.ac.uk

To mark the individuality among processes which accomplish work, and processes which are stronger at handling errors, the words worker and supervisor processes are used. So the worker process is required to do the work and the supervisor process is supposed to supervise the worker. So when there is an error in the worker process the supervisor identify it and tries to handle the error. The good thing about it is that there is a clear division of the labour so the worker does not need to worry about the handling of the errors and vice versa. We can also have distinct processes which are only dealing with the errors. Also the workers and the supervisors can be used on separate computers. Last the usual case is that the error handling code is general and the worker code is more precise.

Erlang have an extraordinary method of dealing with exceptions which is let it crash or the idea is that some other process is going to repair the error. So in event of an error the program would just crash. Exceptions occur either when the run-time system does not know what to do or the errors occur when the programmer cannot find a solution. When an exception is generated by the run-time system, but the programmer knows what to do in order to fix the error than this may not be counted as an error because he can write a code to handles the case. When the programmer does not know what to do this is when the errors happen. Often this is the case because there is not clear specification to be followed by the programmer.
To create a system fault-tolerant we establish the software into an order of tasks that must be executed. The top level task is supposed to run the application to certain conditions. If the task cannot be done then the system will try to do a lower level task. If the lowest level task in the system cannot be done then the system will crash. This method uses the divide and conquers strategy so if something cannot be done then try to do something easier. Simpler tasks are done by simpler software so that there is higher chance of success. In a case of failure what matters is the protecting of the system and the reason that it may fail, so that something can be done to prevent it. For this it is required an error log so even if the system fails we will have the reason why it failed. Depending where the event is detected it depends whether it is an exception, error or failure. At the lowest level in the system of Erlang the virtual machine notices an error. It could be division by zero or a pattern matching error. So the error can only throw an exception on this level. At the next level, it is not mandatory that the exception is caught. The error could be positively adjusted then no damage is done and the processes can go on as usual. If the exception cannot be corrected than another exception will be thrown. If there is not catch statement than the process will fail. The related processes which cooperate with the unsuccessful signals may work as there is no error. So what starts as extraordinary condition in the virtual machine might try to fix the error.
When a program is run the run-time system does not know what the error is. The only hint is that we know something is not right when there is an exception. The run-time system generates the exceptions automatically when it is not clear what to be done by the system. If we try to divide by 0 then an exception will be generated because the

By: Kiril Anastasov L1087591@live.tees.ac.uk

system does not know what to do. The programmer is responsible for deciding whether an exception is error or not. In the real systems it is hard to have all the specifications so the programmer is responsible for the cases where there are no clear specifications.

Distributed programs are such that are produced to run on systems of machines and that can organize their actions only by passing messages. The main reasons that distributed systems are desired are, first is the performance of our applications. When we run in parallel diverse parts of the program on diverse computers we can achieve better speed. Second we have a more reliable system when we run it on multiple computers because if one computer stops working, the operations can go on working on another machine. Third is Scalability when we upgrade the system we will need to add extra computers for more capacity. This operation should be relatively easy and is not supposed to need big changes in the architecture (Armstrong, 2003). Last many programs like chat server are distributed by inheritance. If we have a large number of Facebook users in Europe then it will make sense to place the calculation assets in Europe not in Asia. There are two main models of distributed systems. The first one is Distributed Erlang which delivers a technique that allows the applications to run on very united computers. Programs are supposed to run on Erlang nodes and applications usually run in a trusted network on the same LAN and with a firewall. The second one is socket-based distribution which uses TCP and IP sockets. The main advantage of this distributed system is that applications can be written in unsafe environments. On the other hand, this model is not as powerful as Distributed Erlang but it is safer. Most of the people are used to sequential programming and know that when you write a distribute program it is extra challenging. That is why there are techniques for writing distributed programs which are very helpful and useful even if they are simple (Armstrong, 2003).

OTP or Open Telecom Platform is an application and a group of libraries and methods that are used for constructing big, fault-tolerant, distributed systems which was developed by Eriksson for the telecommunication systems. There are a numbers of influential gears in OTP written in Erlang like FTP server and CORBA ORB. In addition, OTP has gears for structure telecom applications with enactments of H248, SNMP and ASN.1-to-Erlang cross-compiler (Armstrong, 2003). The central concept of the OTP behavior is valuable for developing your own applications using OTP. A behavior captures shared behavioral designs like the callback module. OTP is valuable because it provided assets like fault tolerance, scalability and hot code swapping by the behavior. The writer of the callback does not have to think about things as fault tolerance because it is delivered by the behavior. In Java behavior is something like a container so you do not have to worry about it either. So the behavior solves the nonfunctional parts of the problem, while the callback solved the functional parts. The good part about this is that the nonfunctional parts of the problem like hot code

By: Kiril Anastasov L1087591@live.tees.ac.uk

swapping are the same for all the programs, whereas the functional parts are usually diverse.

A few decades ago there were two models of concurrency. The first one was shared state which was chosen by most of the people. The other model is message passing which is used in Erlang and a few other languages like Occam (Armstrong, 2003). In message passing there is not shared state. All the calculation is done in the processes and the only way to share data is with asynchronous message passing. In shared state the point is that state is mutable or changeable in languages like Java, C++ and Pascal. Armstrong (2003) believes that the state can be changes as long as there is only one process doing the change, but when there are more processes sharing and modifying the same state then we have a problem. So to deal with this issue in Java there are locks to stop the multiple processes sharing and modifying but there may be one problem with it. If the program crashes in the crucial region then there are going to be serious consequences because the other programs will not know what to do. These kinds of problems are difficult to deal with because when you test the problem on a single processor it reacts on one way and when you test it on a multicore processer you get a different result. Erlang deals with this kind of issues in different manner because it uses diverse kind of concurrency for message passing so there is no mutable data and no need for locks or the synchronized keyword. It is relatively easy to parallelize by breaking the solution of the problem into a number of parallel processes which is called concurrency-oriented programming, invented by Armstrong.

If a few simple rules are followed then your Erlang program can run n times faster on an n core processor without changing the program. For example you need to have a lot of processes and make sure that they do not interfere with each other and you have no sequential bottlenecks in the program. If you have created a code that does not use correct principles to make parallel process than the program might not be faster but with some changes it will be faster on multi core processor. Since the multi-core processors are common it is important that we use them in the most efficient way as possible. So making something run 100 times faster can be very exciting and even possible. However there are few rules that need to be followed in order to achieve high speed. The first one is use lots of processes. The second one is to avoid side effects. The third one is to avoid sequential bottlenecks and the last one is to write a code that follows the principle of small messages and big computations.

By: Kiril Anastasov L1087591@live.tees.ac.uk

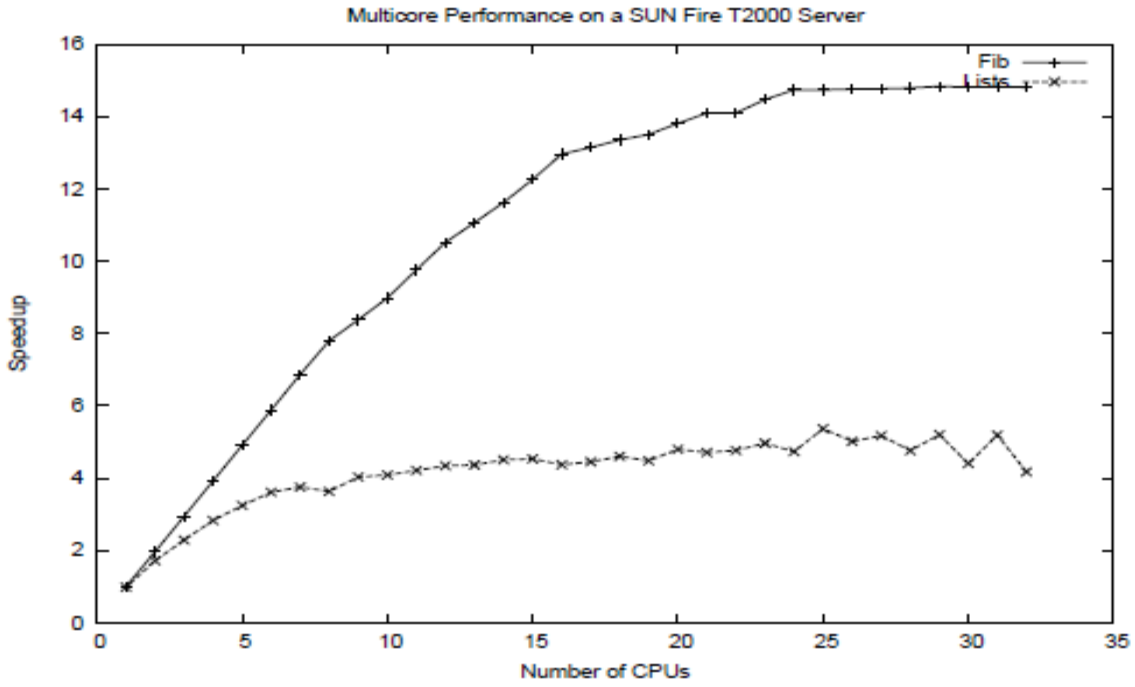

Figure 1 Speed up on multicore CPU (Armstrong 2007)

In figure 1 we can see a clear difference of the speed when we have an Erlang program (Fib line) working on multicore CPU. When the number of cores is between 4 and 20 the growth of the speed is exponential. For the other (ists line) we can still see that there is growth between 4 and 20 cores but not as big as with the (Fib line).

The programming language Haskell has a few advantages. First the code is excessive because it is sensitive and elastic. Second Haskell is safe language to write your code in, maybe even safer than Erlang. The type structure is quite sensitive and potent (Stack Exchange, 2011). Also Haskell has Abstract data type (ADT**)** which is a good. Haskell usually creates fast code. GHC is package for a number of operating systems, distributions and does some inspiring optimizations and super compilations. Haskell's threading system depends on threads with a many-to-one mapping to the operation system threads. This allows you to use more cores without changing your Haskell code. The Erlang VM is doing the same with the help of SMP which is used to parallelism mainly. The thread design also is quick. In the threadring solution Haskell is on the first place while Erlang is about 4 times slower. Haskell supports Exceptions and it is possible to throw exceptions to other threads asynchronously. This is something like linking Erlang processes which is better in Erlang. Monads which are composable computation descriptions are valuable in guaranteeing correctness. So you can limit the user from Input Output in the code unless the code is not safe. The capability to control what the user is able to do when you want to is a powerful type system. The main advantage of Erlang over Haskell is the distributed model in Erlang. Even though there

By: Kiril Anastasov L1087591@live.tees.ac.uk

is distributed Haskell it is not on the same level as the one in Erlang. In addition, it is relatively easy to learn Erlang and it requires a lot of practice to become good with the Haskell code. Haskell's type system has been developed over the years to encourage a relatively flexible, expressive static checking discipline, with several groups of researchers identifying type system techniques that enable powerful new classes of compile-time verification. Scala's is relatively undeveloped in that area.That is, Haskell/GHC provides a logic that is both powerful and designed to encourage type level programming which is fairly unique in the world of functional programming (Scala, 2009). Haskell is considered to be a pure language, with consequences that the outcome of a function rest on only its arguments. In order to permit effects without losing purity, the type system checks the use of results in a program. This functionality creates side-producing programs easier to comprehend and endorse. The type system also shortens reasoning: valuable protection and safety possessions are guaranteed statically, merely by the detail that a program is satisfactory-typed (Stack Exchange, 2011). Even though the programming language OCaml uses message passing concurrency which is rare the concurrency of OCaml is not as good as the one in Erlang developed by Armstrong (Ocaml Programming Manual, 1984). At present, the OCaml runtime does not support running across multiple cores in parallel, so a single OCaml process cannot take advantage of multiple cores. This is unlikely to change directly; the direction the OCaml developers are most interested in taking for increased parallelism seems to be allowing multiple OCaml runtimes to run in parallel in a single process; this will allow for very fast message passing, but will not allow multiple threads to run in parallel in a shared-memory configuration. The major issue is the garbage collector; a few years ago an unacceptable slowdown was introduced in the single-threaded case. There are a few projects that can provide multicore parallelism by using multiple processes some of which are Functory and OCamlnet but in general the people that use Ocaml prefer message passing approaches which is doable with across process boundaries, over single-process shared-memory multithreading so that multiple CPUs can be used effectively (Marlow *et al.,* 2008).

The hardware we depend on is evolving with very fast rates as the number of cores are substituting the chips. So in the near future we could see that the concurrency and the parallelism will be fundamental for the software development. Software developers that need concurrency have found out that shared-state concurrency used by Java C++ and C# are not as powerful as some of the functional language that can provide better concurrency (Miller, 2013).
Most of the programmers know that the software relies on hardware, which for many years depends on chips becoming faster at a relentless pace. According to Moore's Law the speed of a chip will double roughly every 18 months to 24 months (Investopedia, 2012). We could see that the Moore's Law has been right for many years and sometimes even went further of that pace. In the 1970s the chips that were produced by Intel, who had the monopoly of the market, had several thousand

By: Kiril Anastasov L1087591@live.tees.ac.uk

transistors (Scarberry, 2007). Adding the compound interest, which Einstein (20th century) said was the greatest mathematical discovery of all time, it can be calculated that the chips have many millions and even billions of transistors. Up to lately, the growth in chip has made them much quicker, amplified the number of pipelines, and amplified the number and dimensions of on-chip caches. In the beginning of the 21st century the chips became that powerful that their full capacity could not be used fully or their potential was hard to breach. Yet the transistors per chip went on with their unrealistic growth. But they started to increase with multi cores instead with only one core. The speed of the processors is more likely to stay smooth with a small chance of going down a bit. The multicore era is coming with full speed, and it will change a lot of thing about programming something that a lot of amateurs have to deal with.
Most consumers use 2 core computers for normal work, or maybe multi-core computers for more sophisticated needs (Grouchnikov, 2007). An application usually is deployed on multi-core machines. The concurrency model that almost all languages use now is shared state concurrency and can make relatively decent practice of many cores and provide quick and effective software.With the Moore's law we can expect that the number of cores to be doubling every 18 to 24 months so in 10 years we can expect the number of cores to be on a power of 5 or 6. This can seems a bit strange at first but this is what is exponential growth and could be observed in the last few decades. So in the next few decades it is not exactly clear whether the shared-state concurrency models are going to dominate the market as they used to. Even though we may expect something to happen there is only one way to find out for sure.

As already said the most influential and used languages today use the shared state and locks for concurrency. Since Java is the best language so far a standard Java class that deals with locks will be examined.

```
private class Count {

  protected double count = 0.0;

  protected synchronized double incrementMe() {

  return ++count;

  }

}
```

The class Count considered to be thread-safe that keeps a counter (the shared state) and restricts the access to that state by permitting only one thread at a time to modify

the state. In the example of Count is protected by the synchronized keyword so only one thread has access to the method incrementMe and there is no chance of deadlocks, but it could also be delivered by clear Lock objects (Miller, 2013).

Every decent Java or any shared-state concurrency developer has dealt with the difficulties of concurrency. Either by dealing with deadlocks or the necessity for locking that could lead to a contention that is locked.

Some of the difficult parts of the concurrent programming come because when you write a few components that are well written and combine then it does not necessarily produce a system that works as expected. Some of the issues can be dealt with carefully considering the architecture part or by having new layers of locks which will bring extra complexity. However, most of the ways to arrange these problems result in reduced performance and unmaintainable code.

If the shared-state concurrency can balance is a very significant question. Will it balance to 50 or 500 cores? Some parts of your program will logically measure up. If you have got a thread pool occupied on a set of parallel responsibilities, it is completely reasonable to scale that to a bigger number of cores. But in general, a big part of your program does not contain separate tasks that can be simply made parallel. Amdahl's Law is used to calculate the maximum benefit from increasing the parallel parts of your program (Amdahl's Law, 2013). However, when you scale, the non-parallel parts of your program are the ones that matter most for the Amdahl's Law, and they cannot be parallelized.

Conclusion

It is hard to predict what is going to happen but we can examine the most probable output. With the Keifer project the future of Erlang seems to become brighter and brighter. 32 core processors can be optimised well with the concurrency that Erlang can offer. Also with the Sun shipping the Niagara project brings more good news for Erlang. Armstrong (2007) has been waiting and hoping for this to happen for a few decades. With the Erlang prospective that your program will run almost N times faster on an N core processor the spread of Erlang is increasing. We cannot say that Erlang will become the most used language like Java but by increasing of the cores of the processors and the optimisation that Erlang provides the future is definitely bright for it (Armstrong, 2007).

By: Kiril Anastasov L1087591@live.tees.ac.uk

References

Agha, G. (1986) Actors: A model of concurrent computation in distributed systems. In *MIT Series in Artificial Intelligence.* MIT Press, Cambridge, MA.

Amdahl's law (2013) *Amdahl's law description* Available at: http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Amdahl_s_law.html (Accessed: 30 December 2013)

Armstrong, J. (2003), *Making Reliable Distributed Systems in the Presence of Errors.* Ph.D. Thesis, Royal Institute of Technology, Stockholm.

Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World.* The Pragmatic Bookshelf, Raleigh, NC.

Armstrong, J. (2010). A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*.

Claessen, K. and Hughes, J. (2000), Quickcheck: A Lightweight tool for random testing of Haskell programs. In*Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming.* ACM Press, New York, 268–279.

Däcker, B. (2000), *Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction.* Licentiate Thesis, Royal Institute of Technology, Stockholm.

Grouchnikov, K. (2007) *Multicore processing for client-side Java applicaitons*. Available at: http://www.javaworld.com/article/2077768/build-ci-sdlc/multicore-processing-for-client-side-java-applications.html (Accessed: 30 December 2013)

Halloway, S. (2008) *Java next: Immutability*. Available at: http://www.javaworld.com/article/2072366/java-next--4--immutability.html (Accessed: 30 December 2013)

Hoare (1985), C.A.R. *Communicating Sequential Processes.* Prentice Hall, Upper Saddle River, NJ.

Investopedia (2012) *Definition of More's Law* Available at: http://www.investopedia.com/terms/m/mooreslaw.asp (Accessed: 30 December 2013)

Investopedia (2013) *The concept of compounding*. Available at: http://www.investopedia.com/university/beginner/beginner2.asp (Accessed: 30 December 2013)

Marlow, S., Harris, T., James, R and Jones, S. (2008), '*Parallel generational-copying garbage collection with a block-structured heap*', Available at: http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel-gc/par-gc-ismm08.pdf (Accessed: 30 December 2013)

Miller, A. (2013) *Understanding actor concurrency part 1 Actors in Erlang*. Available at: http://www.javaworld.com/article/2077999/java-concurrency/understanding-actor-concurrency--part-1--actors-in-erlang.html (Accessed: 30 December 2013)

Norvell, T. (2007) *Better monitors for Java*. Available at: http://www.javaworld.com/article/2077769/core-java/better-monitors-for-java.html (Accessed: 30 December 2013)

*Occam Programming Manual (*1984*),* Prentice Hall, Upper Saddle River, NJ.

Orbitz-Erlang (2009) *Impressed with Haskell's Concurrency*. Available at: http://orbitz-erlang.blogspot.co.uk/2009/09/impressed-with-haskells-concurrency.html (Accessed: 30 December 2013)

Scala (2009), Scala vs. Erlang Available at: http://www.scala-lang.org/old/node/1070.html (Accessed: 30 December 2013)

Scarberry, R. (2006) *Hyper-threaded Java*. Available at: http://www.javaworld.com/article/2076183/build-ci-sdlc/hyper-threaded-java.html (Accessed: 30 December 2013)

Stack Exchange (2011) *Haskell versus Erlang for web services*. Available at: http://programmers.stackexchange.com/questions/55264/haskell-vs-erlang-for-web-services (Accessed: 30 December 2013)

Rennels, D (2000) Fault-tolerant computing Available at: http://www.cs.ucla.edu/~rennels/article98.pdf (Accessed: 30 December 2013)

Yariv (2008), Erlang vs Scala Available at: http://yarivsblog.blogspot.co.uk/2008/05/erlang-vs-scala.html (Accessed: 30 December 2013)

By: **Kiril Anastasov L1087591@live.tees.ac.uk**

Appendix

Go to the link below ⬇

https://github.com/kanastasov/AlternativeProgrammingApproachesErlang.git

For any of the following:

- ✓ **To download the report.**
- ✓ **To download the Specifications.**
- ✓ **To consult with my work in the IT field.**

By: **Kiril Anastasov L1087591@live.tees.ac.uk**