iMatix
Corporation

Multithreaded Magic with ØMQ

Fast multicore applications in any language with no pain

A Whitepaper from iMatix

Table of Contents

Abstract

In this whitepaper iMatix Corporation's Pieter Hintjens and Martin Sustrik examine the difficulties of building concurrent (multithreaded) applications and what this means for enterprise computing. The authors argue that a lack of good tools for software designers means that neither chip vendors not large businesses will be able to fully benefit from more than 16 cores per CPU, let alone 64 or more. They then examine an ideal solution, and explain how the ØMQ framework for concurrent software design is becoming this ideal solution. Finally they explain ØMQ's origins, and the team behind it.

Going multi-core

Until a few years ago, concurrent programming was synonymous with high-performance computing (HPC) and multithreading was what a word processor did to re-paginate a document while also letting you edit it. Multi-core CPUs were expensive and rare, and limited to higher-end servers. We achieved speed by getting more and more clock cycles out of single cores, which ran hotter and hotter.

Today, multi-core CPUs have become commodity items. While clock speeds are stable at around 2-3GHz, the number of cores per chip is doubling every 18-24 months. Moore's Law still applies. The spread of multi-core CPUs out from the data centre will continue so that netbooks and portable devices come with 2 or 4 cores and top-end server CPUs come with 64 cores. And this growth will continue, indefinitely.

Several factors drive this evolution. First, the need for CPU producers to compete. Whether or not we can use the power, we prefer to buy more capacity. Second, hitting the clock cycles ceiling, CPU designers have found multi-core to be the next way of scaling their architectures and offering more competitive products. Third, at the low end, the spread of multitasking operating systems like Android mean that those additional cores can immediately translate into performance. And lastly, at the high end, the high cost of a data centre slot for a blade computer (estimated by one investment bank at $50,000 per year) pushes users to demand more cores per blade.

It has been clear for some years that the future is massively multi-core, but the software industry is lagging behind. At the *High Performance on Wall Street 2008* event, speaker after speaker said the same thing: our software tools are not keeping up with hardware evolution.

The most widely used languages, C and C++, do not offer any support for concurrency. Programmers roll their own by using threading APIs. Languages that do support concurrency, such as Java, Python, .NET, and Ruby, do it in a brute-force manner. Depending on the implementation - there are over a dozen Ruby interpreters, for example - they may offer "green threading" or true multithreading. Neither approach scales, due to reliance on locks. It is left to exotic languages like Erlang to do it right. We'll see what this means in more detail later.

I could end this article by telling everyone to just switch to Erlang but that's not a realistic answer. Like many clever tools, it works only for very clever developers. But most developers - including those who more and more need to produce multithreaded applications - are just average.

So let's look at what the traditional approach gets wrong, what Erlang gets right, and how we can apply these lessons to all programming languages. Concurrency for average developers, thus.

The Painful State of the Art

A technical article from Microsoft titled "*Solving 11 Likely Problems In Your Multithreaded Code*" demonstrates how painful the state of the art is. The article covers .Net programming but these same problems affect all developers building multithreaded applications in conventional languages.

The article says, "*correctly engineered concurrent code must live by an extra set of rules when compared to its sequential counterpart.*" This is putting it mildly. The developer enters a minefield of processor code reordering, data atomicity, and worse. Let's look at what those "extra rules" are. Note the rafts of new terminology the poor developer has to learn.

- ***Forgotten synchronization***. When multiple threads access shared data, they step on each others' work. This causes "race conditions": bizarre loops, freezes, and data corruption bugs. These effects are timing and load dependent, so non-deterministic and hard to reproduce and debug. The developer must use locks and semaphores, place code into critical sections, and so on, so that shared data is safely read or written by only one thread at a time.

- ***Incorrect granularity***. Having put all the dangerous code into critical sections, the developer can still get it wrong, easily. Those critical sections can be too large and they cause other threads to run too slowly. Or they can be too small, and fail to protect the shared data properly.

- ***Read and write tearing***. Reading and writing a 32-bit or 64-bit value may often be atomic. But not always! The developer could put locks or critical sections around everything. But then the application will be slow. So to write efficient code, he must learn the system's memory model, and how the compiler uses it.

- ***Lock-free reordering***. Our multithreaded developer is getting more skilled and confident and finds ways to reduce the number of locks in his code. This is called "lock-free" or "low-lock" code. But behind his back, the compiler and the CPU are free to reorder code to make things run faster. That means that instructions do not necessarily happen in a consistent order. Working code may randomly break. The solution is to add "memory barriers" and to learn a lot more about how the processor manages its memory.

- ***Lock convoys***. Too many threads may ask for a lock on the same data and the entire application grinds to a halt. Using locks is, we discover as we painfully place thousands of them into our code, inherently unscalable. Just when we need things to work properly, they do not. There's no real solution to this except to try to reduce lock times and re-organize the code to reduce "hot locks" (i.e. real conflicts over shared data).
- ***Two-step dance***. In which threads bounce between waking and waiting, not doing any work. This just happens in some cases, due to signaling models, and luckily for our developer, has no workarounds. When the application runs too slowly, he can tell his boss, "I think it's doing an N-step dance", and shrug.
- ***Priority inversion***. In which tweaking a thread's priority can cause a lower-priority threads to block a higher-priority thread. As the Microsoft article says, "*the moral of this story is to simply avoid changing thread priorities wherever possible.*"

This list does not cover the hidden costs of locking: context switches and cache invalidation that ruin performance.

Learning Erlang suddenly seems like a good idea.

Cost of the Traditional Approach

Apart from the difficulty of finding developers who can learn and properly use this arcane knowledge, the state of the art is expensive in other ways:

- It is significantly more expensive to write and maintain such code correctly. We estimate from our own experience that it costs at least as 10 times and perhaps up to 100 times more than single-threaded code. And this while the cost of coding is falling elsewhere, thanks to cheaper and better tools and frameworks.
- The approach does not scale beyond a few threads. Multithreaded applications mostly use two threads, sometimes three or four. This means we can't expect to fully exploit CPUs with 16 and more cores. So while we can get concurrency (for that word processor), we do not get scalability.
- Even if the code is multithreaded, it does not really benefit from multiple cores. Individual threads are often blocking each other. And developers do not realize that their fancy multithreaded application is essentially reduced to single threadedness. Tools like Intel's *ThreadChecker* help diagnose this but there is no practical solution except to start designing the application again.
- Even in the best cases, where applications are designed to avoid extensive locking, they do not scale above 10 cores. Locking, even when

sparse and well-designed, does not scale. The more threads and CPU cores are involved, the less efficient it becomes. The law of diminishing returns hits particularly hard here.

- To scale further, our developer must switch to 100% lock-free algorithms for data sharing. He is now into the realm of black magic. He exploits hardware instructions like compare-and-swap (CAS) to flip between instances of a data structure without locks. The code is now another order of magnitude more difficult and the skill set even more rare. Lose one developer, and the entire application can die.

All of this adds up to a mountain of cost. Yes, we can put lovely multi-core boxes into our data centers. But when we ask our development teams to build code to use those, and they start to follow industry practice, the results are horrendous.

For CPU producers, this cost creates a hard ceiling, where buyers will decide to stop buying newer CPUs because they cannot unlock the power of those extra cores.

Let's see how an ideal solution would work, and whether we can apply that to the real world of Java, C, C++, and .NET.

Towards an Ideal Solution

Of all the approaches taken to multithreading, only one is known to work properly. That means, it scales to any number of cores, avoids all locks, costs little more than conventional single-threaded programming, is easy to learn, and does not crash in strange ways. At least no more strangely than a normal single-threaded program.

Ulf Wiger summarizes the key to Erlang's concurrency thus:

- Fast process creation/destruction
- Ability to support >> 10 000 concurrent processes with largely unchanged characteristics.
- Fast asynchronous message passing.
- Copying message-passing semantics (share-nothing concurrency).
- Process monitoring.
- Selective message reception.

The key is to pass information as messages rather than shared state. To build an ideal solution is fairly delicate but it's more a matter of perspective than anything else. We need the ability to deliver messages to queues, each queue feeding one process. And we need to do this without locks. And we need to do this locally, between cores, or remotely, between boxes. And we need to make this work for any programming language. Erlang is great in theory but in practice, we need something that works for Java, C, C++, .Net, even Cobol. And which connects them all together.

If done right, we eliminate the problems of the traditional approach and gain some extra advantages:

- Our code is thread-naive. All data is private. Without exception, all of the "likely problems of multithreading code" disappear: no critical sections, no locks, no semaphores, no race conditions. No lock convoys, 3am nightmares about optimal granularity, no two-step dances.
- Although it takes care to break an application into tasks that each run as one thread, it becomes trivial to scale an application. Just create more instances of a thread. You can run any number of instances, with no synchronization (thus no scaling) issues.
- The application never blocks. Every thread runs asynchronously and independently. A thread has no timing dependencies on other threads. When you send a message to a working thread, the thread's queue holds the message until the thread is ready for more work.
- Since the application overall has no lock states, threads run at full native speed, making full use of CPU caches, instruction reordering, compiler optimization, and so on. 100% of CPU effort goes to real work, no matter how many threads are active.

This functionality is packaged in a reusable and portable way so that programmers in any language, on any platform, can benefit from it.

If we can do this, what do we get? The answer is nothing less than: perfect scaling to any number of cores, across any number of boxes. Further, no extra cost over normal single-threaded code. This seems too good to be true.

The ØMQ Framework

ØMQ (ZeroMQ) started life in 2007 as an iMatix project to build a low-latency version of our OpenAMQ messaging product, with Cisco and Intel as partners. From the start, ØMQ was focused on getting the best possible performance out of hardware. It was clear from the start that doing multithreading "right" was the key to this.

We wrote then in a technical white paper that:

"*Single threaded processing is dramatically faster when compared to multi-threaded processing, because it involves no context switching and synchronisation/locking. To take advantage of multi-core boxes, we should run one single-threaded instance of an AMQP implementation on each processor core. Individual instances are tightly bound to the particular core, thus running with almost no context switches.*"

ØMQ is special, and popular, for several reasons:

- It is fully open source and is supported by a large active community. There are over 50 named contributors to the code, the bulk of them from outside iMatix.

- It has developed an ultra-simple API based on BSD sockets. This API is familiar, easy to learn, and conceptually identical no matter what the language.
- It implements real messaging patterns like topic pub-sub, workload distribution, and request-response. This means ØMQ can solve real-life use cases for connecting applications.
- It seems to work with every conceivable programming language, operating system, and hardware. This means ØMQ connects entire applications as well as the pieces of applications.
- It provides a single consistent model for all language APIs. This means that investment in learning ØMQ is rapidly portable to other projects.
- It is licensed as LGPL code. This makes it usable, with no licensing issues, in closed-source as well as free and open source applications. And those who improve ØMQ automatically become contributors, as they publish their work.
- It is designed as a library that we link with our applications. This means there no brokers to start and manage, and fewer moving pieces means less to break and go wrong.
- It is simple to learn and use, with a learning curve of roughly one hour.

And it has odd uses thanks to its tiny CPU footprint. As Erich Heine writes, "the [ØMQ] perf tests are the only way we have found yet which reliably fills a network pipe without also making cpu usage go to 100%".

Most ØMQ users come for the messaging and stay for the easy multithreading. No matter whether their language has multithreading support or not, they get perfect scaling to any number of cores, or boxes. Even in Cobol.

One goal for ØMQ is to get these "sockets on steroids" integrated into the Linux kernel itself. This would mean that ØMQ disappears as a separate technology. The developer sets a socket option and the socket becomes a message publisher or consumer, and the code becomes multithreaded, with no additional work.

Summary and Conclusion

Inevitably, CPUs big and small are moving toward multi-core architectures. But traditional software engineering has no tools to deal with this. The state of the art is expensive, fragile, and limited to a dozen cores at most, with great effort. Specialized languages such as Erlang do multithreading "right", but remain out of reach of the worlds mainstream programmers. As a consequence, the potential of massively-multi-core systems remains untapped, and CPU producers find a software market that cannot use their latest products.

Hope comes in the shape of iMatix Corporation's ØMQ. From its origins as a messaging system, this small and modest library provides the power of a language like Erlang, but in a form that any developer can use. ØMQ looks like BSD sockets, it is trivial to learn, and yet lets developers create applications that scale perfectly to tens of thousands of processes, running over many boxes with as many cores as wanted.

Unlike traditional multithreading, which uses locks to share data, ØMQ passes messages between threads. Internally, it uses black magic "lock-free" algorithms, but these are hidden from the developer. The result is that ØMQ application code looks just like ordinary single-threaded code, except that it processes messages.

The ØMQ project was, from birth, built around an open source community of expert engineers from iMatix, Cisco, Intel, Novell, and other firms. The project has always focussed on benchmarking, transparent tests, and user-driven growth. Today the community develops the bulk of the code outside the ØMQ "core", including language bindings for everything from Ada to OpenCOBOL, including Java, C++, .Net, Python, and even Erlang.

For developers who want a simple messaging bus, ØMQ uses familiar APIs and takes literally less than an hour to install, learn and use.

For architects who need low-latency messaging for market data distribution or high-performance computing, ØMQ is an obvious candidate. It is free to use, easy to learn, and is extremely fast (measured on Infiniband at 13.4 usec end-to-end latencies and over 8M messages a second).

For architects who need scalability to multiple cores, ØMQ provides all that they need, no matter what their target languages and operating systems. As a plus, they can stretch the resulting applications across more boxes when they need to.

About iMatix Corporation

At iMatix we've done message-oriented multithreading since 1991, when we used this approach to build fast and scalable front-ends for OpenVMS. In 1996 we used a similar approach to build a fast lightweight web server (Xitami). The Xitami framework was single core, using "green threading" rather than real OS threads. We used this framework for many server-side applications.

From 2004 we worked with JPMorganChase to design AMQP, and its first implementation, OpenAMQ. JPMorgan migrated their largest trading application to OpenAMQ in 2006. Today OpenAMQ runs the Dow Jones Industrial Average and is widely used in less well-known places.

In 2009, iMatix released ØMQ/2.0 with its simpler socket-style API, and in 2010 announced that it would cease support for AMQP in its products, citing irremediable failures in the AMQP development process.

iMatix CEO Pieter Hintjens was the main designer of AMQP/0.6 and the editor of the de-facto standard AMQP/0.9.1. Martin Sustrik, architect and maintainer of ØMQ, worked for three years in the iMatix OpenAMQ team, and was project leader for the AMQP working group.