acmqueue

Case Study
Photoshop Scalability: Keeping It Simple

Clem Cole and Russell Williams discuss Photoshop's long history with parallelism, and what they now see as the main challenge.

Over the past two decades, Adobe Photoshop has become the de facto image-editing software for digital photography enthusiasts, artists, and graphic designers worldwide. Part of its widespread appeal has to do with a user interface that makes it fairly straightforward to apply some extremely sophisticated image editing and filtering techniques. Behind that façade, however, stands a lot of complex, computationally demanding code. To improve the performance of these computations, Photoshop's designers became early adopters of parallelism—in the mid-1990s—through efforts to access the extra power offered by the cutting-edge desktop systems of the day that were powered by either two or four processors. At the time, Photoshop was one of the only consumer desktop applications to offer such a capability.

Photoshop's parallelism, born in the era of specialized expansion cards, has managed to scale well for the two- and four-core machines that have emerged over the past decade. As Photoshop's engineers prepare for the eight- and 16-core machines that are coming, however, they have started to encounter more and more scaling problems, primarily a result of the effects of Amdahl's law and memory-bandwidth limitations.

In this ACM Case Study, Adobe Photoshop principal scientist Russell Williams speaks with Clem Cole, architect of Intel's Cluster Ready program, about how the Photoshop team is addressing these challenges. Although in their current positions they represent different aspects of the parallel-computing landscape, both have long histories of tackling parallelism at the operating-system level.

Prior to joining the Photoshop development team, Williams had a long career as an operating-system designer at companies such as Apple, where he worked on the Copland microkernel, and Elxsi, where he helped develop mini-supercomputers. The diversity of that background now allows him to maintain a well-rounded perspective on parallelism at different levels of the stack.

Cole is a veteran operating-system developer with years of experience in Unix kernel and tool development. His current efforts to advance methods that take advantage of multiple processors—using Intel's next generation of multicore chips—makes him a fitting interviewer for Williams, whose work in large part builds on top of the platforms Cole helps to create at Intel.

While Photoshop comes with a unique set of problems and constraints, many of the engineering challenges it presents will undoubtedly seem familiar to any software engineer who has ever attempted to achieve parallelism in an application. Still, to get a handle on the issues Photoshop's engineers are facing today, we must first consider the application's history with parallelism over the past 15 years.

COLE You've been writing software for a long time, and for the past 11 years you've been working with Photoshop and have become increasingly engaged with its parallel aspects. Which parts of that have proved to be easy and what has turned out to be surprisingly hard?

WILLIAMS The easy part is that Photoshop has actually had quite a bit of parallelism for a long

time. At a very simplistic level, it had some worker threads to handle stuff like asynchronous cursor tracking while also managing asynchronous I/O on another thread. Making that sort of thing work has been pretty straightforward because the problem is so simple. There's little data shared across that boundary, and the goal is not to get compute scaling; it's just to get an asynchronous task going.

I should note, however, that even with that incredibly simple task of queuing disk I/O requests so they could be handled asynchronously by another thread, the single longest-lived bug I know of in Photoshop ended up being nestled in that code. It hid out in there for about 10 years. We would turn on the asynchronous I/O and end up hitting that bug. We would search for it for weeks, but then just have to give up and ship the app without the asynchronous I/O being turned on. Every couple of versions we would turn it back on so we could set off looking for the bug again.

COLE I think it was Butler Lampson who said that the wonderful thing about serial machines is you can halt them and look at everything. When we're working in parallel, there's always something else going on, so the concept of stopping everything to examine it is really hard. I'm actually not shocked your bug was able to hide in the I/O system for that long.

WILLIAMS It turned out to be a very simple problem. Like so many other aspects of Photoshop, it had to do with the fact that the app was written first for the Macintosh and then moved over to Windows. On the Macintosh, the set file position call is atomic—a single call—whereas on Windows, it's a pair of calls. The person who put that in there didn't think about the fact that the pair of calls has to be made atomic whenever you're sharing the file position across threads.

COLE Now, of course, you can look back and say that's obvious.

WILLIAMS In fact, the person who originally put that bug in the code was walking down the hallway one of the many times we set off looking for that thing, smacked his forehead, and realized what the problem was—10 years after the fact.

Anyway, the other big area in Photoshop where we've had success with parallelism involves the basic image-processing routines. Whenever you run a filter or an adjustment inside Photoshop, it's broken down into a number of basic image-processing operations, and those are implemented in a library that's accessed through a jump table. Early on, that allowed us to ship accelerated versions of these "bottleneck routines," as they're called. In the 1990s, when companies were selling dedicated DSP (digital signal processor) cards for accelerating Photoshop, we could patch those bottlenecks, execute our routine on the accelerator card, and then return control to the 68-KB processor.

That gave us an excellent opportunity to put parallelism into the app in a way that didn't complicate the implementations for our bottleneck-routine algorithms. When one of those routines was called, it would be passed a pointer—or two or three pointers—to bytes in memory. It couldn't access Photoshop's software-based virtual memory and it couldn't call the operating system; it was just a math routine down at the bottom. That gave us a very simple way—prior to getting down to the math routine—of inserting something that would slice up the piece of memory we wanted to process across multiple CPUs and then hand separate chunks of that off to threads on each CPU.

COLE The key there is that you had an object that could be broken apart into smaller objects without the upper-level piece needing to worry about it. It also helps that you had a nice, clean place to make that split.

WILLIAMS The other nice aspect is that the thing on the bottom didn't need to know about synchronization. It was still nothing more than a math routine that was being passed a source pointer—or maybe a couple of source pointers and counts—along with a destination pointer. All

the synchronization lived in that multiprocessor plug-in that inserted itself into the jump table for the bottleneck routines. That architecture was put into Photoshop in about 1994. It allowed us to take advantage of Windows NT's symmetric multiprocessing architecture for either two or four CPUs, which was what you could get at the time on a very high-end machine. It also allowed us to take advantage of the DayStar multiprocessing API on the Macintosh. You could buy multiprocessor machines from DayStar Digital in the mid- to late-'90s that the rest of the Mac operating system had no way of taking advantage of—but Photoshop could.

Photoshop has well over a decade of using multiple processors to perform the fundamental image-processing work it does on pixels. That has scaled pretty well over the number of CPUs people have typically been able to obtain in a system over the past decade—which is to say either two or four processors. Essentially, no synchronization bugs ended up surfacing in those systems over all that time.

COLE That's an amazing statement! Is there an insight associated with that that you can share? What do you think the rest of us can learn from that?

WILLIAMS I think the big win came from not having to write synchronization for the processing routines that were to be parallelized. In other words, people could write convolution kernels or whatever else it was they needed to do in terms of pixel processing without having to worry about getting all those synchronization issues right. If acquiring one asynch I/O thread was all it took for us to introduce a bug that managed to elude us for 10 years, then it's clear that minimizing synchronization issues is very important.

That said, the way synchronization was approached 10 years ago involved the use of far more error-prone synchronization primitives than what we've got available to us today. Things like "enter critical section" and "leave critical section" on Windows could be really fast, but they were also very error prone. Trying to keep track of whether you've put critical sections every place you might need them, and whether or not you've remembered to leave as many times as you entered, that can all tend to be very difficult and error prone.

The nettlesome bug that managed to remain obscured within Photoshop's synchronization code for 10 years serves to illustrate just how tricky parallel programming can be. But it also highlights how much progress has been made in terms of improved resources for managing some of this complexity. Had Photoshop's synchronization been written today using C++'s stack-based locking, for example, it's unlikely a bug of that sort would have been introduced. As processors get more cores and grow in complexity, the need will only intensify for new tools and better programming primitives for hiding the complexity from developers and allowing them to code at higher levels of abstraction.

At the same time, software architects also need to keep an eye on some other fundamental issues. For example, despite using less-sophisticated synchronization primitives in the original design, the Photoshop team was able to essentially forget about complex thread-synchronization problems, in part because the image-processing problem itself was so amenable to parallelization. Also, however, Photoshop's architecture made it possible to establish some very clean object boundaries, which in turn made it easy for programmers to slice up objects and spread the resulting pieces across multiple processors. Which is to say that the architects of Photoshop were keenly aware of where their best opportunities for parallelization existed, and they designed the application accordingly.

Generalizing on this, it's clear that—with or without advances in tools and programming abstractions—in order for developers to fully leverage the multicore architectures that are

coming, they'll need to be adept at identifying those parts of a program that can benefit most from parallelization. It's in these portions of the code that new tools, techniques, and parallel programming abstractions are likely to have the greatest impact.

COLE As operating-system designers, we both grew up in a world where we had to deal with parallelism. It's not always clear that the solutions we came up with for our operating systems proved to be the right ones. In an earlier conversation, you mentioned your experience with creating and removing mutexes. We've gotten smarter over the years. We've learned how to do things that are more efficient, but that doesn't mean it has gotten any easier. What do we have up our sleeves to make it easier?

WILLIAMS Parallelism remains difficult in a couple of ways. It's one thing to ask for a new Photoshop filter for processing a grid of pixels to do that in parallel. It's quite another thing to say, "I'm going to parallelize and thus speed up the way that Photoshop runs JavaScript actions." For example, I've got a JavaScript routine that opens 50 files one after the other and then performs a set of 50 steps on each one. I don't have control over that script. My job is just to make that—or any other script the user has to run—faster.

I could say, "Rewrite all your scripts so we can design a completely new interface that will let you specify that all these images are to be processed in parallel." That's one answer, but it would require a lot of work on the user's part, as well as on our part. And it would still leave us with the problems associated with parallelizing the opening of an image, parsing the image contents, interpreting the JavaScript, running the key command object through the application framework, and updating the user interface—all of which typically is tied into an app framework and thus involves calling some horrendously sequential script interpreter. Once you start looking at the Amdahl's law numbers on something like that, it soon becomes apparent that trying to get that to parallelize eight ways would just be completely hopeless.

At the other end of the spectrum you might find, for example, a mathematical algorithm that conceptually lends itself to parallelism simply because it has a bunch of data structures it needs to share. So how hard would it be to correctly implement that mathematically parallelizable algorithm?

I think we've made some incremental gains in our ability to deal with parallelism over the past 20 years, just as we've made stepwise progress on all other programming fronts. Remember that back in the '70s, there was a lot of talk about the "software crisis," regarding how software was getting more and more complicated, to the point where we couldn't manage the bugs anymore. Well, in response to that, there was no huge breakthrough in software productivity, but we did realize a bunch of incremental gains from object-oriented programming, improved integrated development environments, and the emergence of better symbolic debugging and checker tools that looked for memory leaks. All of that has helped us incrementally improve our productivity and increase our ability to manage complexity.

I think we're seeing much the same thing happen with parallelism. That is, whereas the earliest Photoshop synchronization code was written in terms of "enter critical section, leave critical section," we now have tools such as Boost threads and OpenGL, which essentially are programming languages, to help us deal with those problems. If you look at Pixel Bender [the Adobe library for expressing the parallel computations that can be run on GPUs], you'll find it's at a much higher level and so requires much less synchronization work of the person who's coding the algorithm.

COLE The key is that you try to go to a higher level each time so you have less and less of the detail to deal with. If we can automate more of what happens below that, we'll manage to become more efficient.

You also mentioned that we have better tools now than we did before. Does that suggest we'll need even better tools to take our next step? If so, what are we missing?

WILLIAMS Debugging multithreaded programs *at all* remains really hard. Debugging GPU-based programming, whether in OpenGL or OpenCL, is still in the Stone Age. In some cases you run it and your system blue-screens, and then you try to figure out what just happened.

COLE That much we're aware of. We've tried to build stronger libraries so that programmers don't have to worry about a lot of the low-level things anymore. We're also creating better libraries of primitives, such as open source TBB (Threading Building Blocks). Do you see those as the kinds of things developers are looking to suppliers and the research community for?

WILLIAMS Those things are absolutely a huge help. We're taking a long hard look at TBB right now. Cross-platform tools are also essential. When somebody comes out with something that's Windows only, that's a nonstarter for us—unless there is an exact-equivalent technology on the Mac side as well. The creation of cross-platform tools such as Boost or TBB is hugely important to us.

The more we can hide under more library layers, the better off we are. The one thing that ends up limiting the benefit of those libraries, though, is Amdahl's law. For example, say that as part of some operation we need to transform the image into the frequency domain. There's a parallel implementation of FFT (Fast Fourier Transform) we can just call, and maybe we even have a library on top of that to decide whether or not it makes any sense to ship all that down to the GPU to do a GPU implementation of FFT before sending it back. But that's just one step in our algorithm. Maybe there's a parallel library for the next step, but getting from the FFT step to the step where we call the parallel library is going to require some messing around. It's with all that inter-step setup that Amdahl's law just kills you. Even if you're spending only 10 percent of your time doing that stuff, that can be enough to keep you from scaling beyond 10 processors.

Still, the library approach is fabulous, and every parallel library implementation of common algorithms we can lay our hands on is greatly appreciated. Like many of the techniques we have available to us today, however, it starts to run out of steam at about eight to 16 processors. That doesn't mean it isn't worth doing. We're definitely headed down the library path ourselves because it's the only thing we can even imagine working if we're to scale to eight to 16 processors.

For the engineers on the Photoshop development team, the scaling limitations imposed by Amdahl's law have become all too familiar over the past few years. Although the application's current parallelism scheme has scaled well over two- and four-processor systems, experiments with systems featuring eight or more processors indicate performance improvements that are far less encouraging. That's partly because as the number of cores increases, the image chunks being processed, called *tiles*, end up getting sliced into a greater number of smaller pieces, resulting in increased synchronization overhead. Another issue is that in between each of the steps that process the image data in parallelizable chunks, there are sequential bookkeeping steps. Because of all this, Amdahl's law quickly transforms into Amdahl's wall.

Photoshop's engineers tried to mitigate these effects by increasing the tile size, which in turn made each of the sub-pieces larger. This helped to reduce the synchronization overhead, but it

presented the developers with yet another parallel-computing bugaboo: memory-bandwidth limitations. Compounding the problem was the fact that Photoshop cannot interrupt pixel-processing operations until an entire tile is complete. So to go too far down the path of increasing tile sizes would surely result in latency issues, as the application would become unable to respond to user input until it had finished processing an entire tile.

Although Williams remains confident his team can continue to improve Photoshop's scaling in the near future through the use of better tools, libraries, and incremental changes to the current approach to parallel processing, eventually those techniques will run out of steam. This means the time has come to start thinking about migrating the application to a different approach altogether that involves new parallel methods and the increased use of GPUs.

COLE I think you already have some interesting ways of splitting things apart for image processing, but for your base application, have you considered other programming paradigms, such as MPI (message passing interface)?

WILLIAMS No, we haven't because we've been occupied with moving from the four-core world to the eight- to 16-core world, and what we see is that Photoshop is just going to be stuck in that world for the next few years. Another reason we haven't looked all that seriously at changing to a message-passing-style interface is that it would require such a huge re-architecture effort and it's not at all clear what the win would be for us there.

COLE The reason I ask is that Intel is obviously looking to enable as many cores in a box as possible, and you mentioned you had previously had problems with memory bandwidth. That's part of the reason why another division of Intel has become interested in the NUMA (non-uniform memory architecture) way of putting things together. I certainly feel we're going to have applications that have both threadish parts and heavily parallel parts, and we're going to see the processors inside of workstations become more cluster-like in many ways. We may not necessarily go off-chip or out of the box, but we're going to break memory up somehow. And we're going to have to do lots of other things to give back some memory bandwidth just because that's going to have a huge impact for somebody like you.

WILLIAMS This comes up in a number of different ways. We get asked a lot about how we're going to handle something like Larrabee [the engineering chip for Intel's MIC—Many Integrated Cores—architecture]. The answer is: basically nothing for now. The reason is that any of these future architectures that promise to solve some particular parallelism problem or some particular part of the performance problem are all kind of speculative at this point. Photoshop, on the other hand, is a mass-market application. So, unless we are fairly certain there are going to be millions of one of these platforms out there, we can't afford to bet our software's architectural direction on that. Right now, we don't see desktop architectures moving beyond the mild and cache-coherent form of NUMA we see today.

As a rule, we avoid writing large amounts of processor-specific or manufacturer-specific code, although we do some targeted performance tuning. For us, life will start to get interesting once we can use libraries such as OpenGL, OpenCL, and Adobe's Pixel Bender—or any higher-level libraries that take advantage of these libraries—to get access to all that GPU power in a more general way.

We've also been looking at the change Intel's Nehalem architecture presents in this area. On all previous multicore CPUs, a single thread could soak up essentially all of the memory bandwidth.

Given that many of our low-level operations are memory-bandwidth-limited, threading them would have only added overhead. Our experience with other multicore CPUs is that they become bandwidth limited with only a couple of threads running, so parallel speedups are limited by memory bandwidth rather than by Amdahl's law or the nature of the algorithm. With Nehalem, each processor core is limited to a fraction of the chip's total memory bandwidth, so even a large memcpy can be sped up tremendously by being multithreaded.

COLE I actually was just trying to make more of an architectural statement than anything. Rest assured, you're going to see just as many cores as we can possibly get in there, but at a certain point, what I refer to as "conservation of memory bandwidth" starts to become the big tradeoff; that's when other architectural tricks are going to have to be used that will have some software impact. The question is, if you can't fire a gun and get everybody to change software overnight, at what point does it become economically interesting for a company such as Adobe to say, "OK, if I know I'm going to have to deal with a cluster in a box, I've got to slowly start moving my base so I'll be able to do that"?

WILLIAMS I suspect we'll end up seeing the same progression we saw with SMP. That is, the hardware and operating-system support will be introduced such that these platforms will be able to run multiple programs, or multiple instances of programs, not yet modified to take advantage of the new architecture. This has already proved to be true with SMP and GPU use. There will be some small handful of applications that will absolutely depend on being able to leverage the brand-new capability right away—as was the case with video games and 3D rendering applications and their need to take advantage of GPUs as soon as possible. The bulk of applications, however, will not start to take significant advantage of new architectures until: (a) there's an installed base; (b) there's software support; and (c) there's a clear direction for where things are heading.

I assume the NUMA and MPI stuff is at the research-lab level at this juncture. And even though the MIC chip is on its way, it's still unclear to us what the programming API will be other than OpenGL and DirectX.

Now, just to throw a question back at you: what do you see the progression being in terms of how the programming API and operating-system support is going to be rolled out, since people like me are going to need that if we're to take advantage of these kinds of architectural innovations?

COLE As we develop specialty hardware, be it GPUs or network engines, the machine is becoming essentially a federation of processing elements designed to handle specific tasks. The graphics processor was highly tuned for doing its job of displaying pixels and performing certain mathematical functions that are important to the graphics guys and the gamers. Then other people came along and said, "Hey, I want to be able to do those same functions. Can I get at them?" That's when engineers like me in the operating-systems world start scratching our heads and saying, "Yeah, well, I suppose we could expose that."

But I wonder if that's what you really want. My experience has been that every time we've had a specialty engine like that and we've tried to feed it to the world, you may have been able to write an application library as you did with Photoshop that was able to call some graphics card, but that typically lived for only a couple of generations. That is, it proved to be cost effective for only that one particular application. So I think the GPU will continue to get smarter and smarter, but it will retain its focus as a graphics engine just the same. That's really where it's going to be most cost effective.

The rest of the box needs to be more general, maybe with a bunch of specialty execution engines

around it that you're able to call up and easily exploit. Then the question becomes: how can the operating system make all those engines available?

Having been one of the early microkernel guys, I smiled when I learned about the early microkernel work you were doing. Many of us in the operating-system community have thought that would be the right way to go.

WILLIAMS Elxsi was a message-based operating system. It was similar to the GNU Hurd of independent processes in that it talked via messages. One of the things that really hit us hard and is showing up today with GPUs in a different context—and, in fact, was the very first thing I thought of when I started looking at NUMA—is that message-based operations are horrendously expensive relative to everything else you do. This is something the video apps have run into as well. They went down this path of doing a rendering graph to represent the stack of effects you had on your video frames, and then they would go down and start rendering and compositing those things. As soon as they got to anything they could do on the GPU, they would send it down there to get processed, and then they would work on it until they hit a node in the compositing stack graph that couldn't be handled on the GPU, at which point they would suck it back into the CPU.

What they found was that even with the fastest-bandwidth cards on the fastest-bandwidth interfaces, one trip was all you got. Anything beyond that meant you would have been better off just staying on the CPU in the first place. When you start moving data around, the bandwidth consumption associated with that can quickly overwhelm the cost of doing the computation. But the GPU vendors are continually improving this.

COLE That's part of why I asked about MPI. I come back to that now only because it seems to be today's popular answer. I'm not saying it *is* the answer; it's just a way of trying to control what has to get shifted and when and how to partition the data so you can write code that will be able to exploit these execution units without having to move lots of data around.

This is why companies such as Intel are exploring interfaces such as RDMA (remote direct memory access), which is something you find inside of IB (InfiniBand). About a year ago, Intel purchased one of the original iWARP (Internet Wide Area RDMA Protocol) vendors, and the company is also heavily committed to the OpenFabrics Alliance's OFED (OpenFabrics Enterprise Distribution) implementations, so we're now exposing that same RDMA interface you find with InfiniBand in both Ethernet form and IB. I certainly think that kind of hardware is going to start showing up inside the base CPU and will become available to you as you try to move those objects around. So you're going to have computational resources and data movement resources, and the processing power will become the federation of all that stuff underneath.

That means the operating system has got to change. And I think you're right: what will happen then is that the apps will become richer and will be able to exploit some of those pieces in the hardware base, provided that the operating system exposes it. That's also when you guys at Adobe will want to start exploiting that, since you'll have customers who already have machines with those capabilities built in.

WILLIAMS When we started to look at NUMA, we ran into some issues with predictability. Ideally, on a big NUMA system, you would want your image to be spread evenly across all the nodes so that when you did an operation, you would be able to fire up each CPU to process its part of the image without having to move things around.

What actually happens, however, is that you've got a stack of images from which somebody makes

a selection, and maybe selects some circle or an area from a central portion of the image containing pixels that live on three out of your 10 nodes. In order to distribute the computation the user then invokes on that selection, you now have to copy those things off to all 10 nodes. You quickly get to a point where your data is fragmented all over the place, and any particular selection or operation is unlikely to get nicely distributed across the NUMA nodes. Then you pay a huge cost to redistribute it all as part of starting up your operation. This, along with the more general issue of bandwidth management, is going to prove to be an even harder parallelism problem than the already well-documented problem people have with correctly locking their data structures.

COLE Yes, we're in violent agreement on that score. Locking your data structures is truly only the beginning. The new tuning problem is going to be exactly the nightmare you just described.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org