

# Parallel Thoughts

---

*James Larus  
Microsoft Research  
May 2007*

Parallel processing is back with a vengeance. Dual-core processors are the norm in most desktop and laptop computers today. Quad-core processors will be widely available later this year. These Multicore<sup>1</sup> processors embody a radically new computer architecture that presents fundamental challenges to the software industry. Below, I'll sketch these challenges and point out some possible solutions.

Multicore processors are the first major break in the evolution of von Neumann computer architectures, which stretches back to the 1940's. From a programmer's perspective, the sixty years of computer evolution was seamless, with only business and minor technical disruptions at the introduction of mainframes, virtual memory, caches, minicomputers, microprocessors, RISC processors, etc. None of these hardware innovations deviated from von Neumann's sequential fetch-execute model, and so none required changes to the basic programming model or even programming languages.

Existing programs ran (faster) on new computers, which enabled new versions of programs to offer new functionality. As software became increasingly complex, developers controlled complexity by using high-level languages, objects, components, and libraries to build abstractions, which provided rich functionality while hiding implementation details. The inevitable tradeoff between performance and possibility was deferred because the underlying hardware doubled in speed every eighteen months.

Multicore processors ended this remarkable era of innovation since a Multicore processor will not run existing software any faster than the prior sequential processor. Existing software, for the most part, was not written to run on multiple processors, so it does not benefit as Multicore becomes increasingly parallel. For sequential software, sixty years of ever-improving computers ended circa 2005.<sup>2</sup>

Obviously, software cannot stagnate. We must rethink software, to produce applications that exploit multiple processors in a manner that will scale as the number of cores on a chip increase. This shift presents two enormous challenges: finding and developing applications that benefit from coarse-grain parallelism and finding more effective ways of writing parallel software.

The first problem may be the more difficult. Commonly used applications appear to offer few opportunities for concurrency, as they are not CPU-bound, are written sequentially, or both. Amdahl's Law<sup>3</sup> extracts a high penalty for sequential code, so scalable parallel programs must be overwhelming concurrent to scale with computers that become twice as parallel every second year. Few applications outside of scientific computing and servers have this characteristic.

---

<sup>1</sup> The terms Multicore and Manycore offer a distinction without a difference, so I'll just use the first term.

<sup>2</sup> It is worth noting that this claim is not true for software that migrates to a server, i.e., software as a service (SaaS). Multicore provides an exponentially improving hardware basis for servers. While Multicore may not improve the software for an individual user, it makes the service increasingly attractive. Only compelling new applications that require substantial computing power on the desktop will check this trend.

<sup>3</sup> The maximum speedup of a parallel program is  $1/(1-P)$ , where P is the fraction of the computation that executes in parallel. If P is 75%, the maximum improvement is a factor of 4—hence achieving full performance on this year's Multicore processors requires a program that can execute three-fourths of its computation in parallel.

To compound the problem, most of us do not think concurrently. Parallel programs are hard to design, develop, debug, and test. Developers were not taught parallel algorithms and data structures, which currently appear appreciable more complex than their sequential analogs.

Finally, no single parallel programming model offers the generality and universality of the von Neumann model. Data parallelism, task parallelism, and message passing are the most common parallel models. Each is appropriate for some parallel problems and wildly inappropriate for others.

Applications for Multicore must look beyond the functionality of existing programs, which have evolved in a world of sequential computing. New, expensive data types, such as high definition video streams, rich animation, voice recognition, and natural language understanding, can enhance computers' user interfaces and enable access to sophisticated computing without a keyboard and mouse.

In addition, systems should become more introspective by using spare processors to monitor user and program behavior to detect and correct security, correctness, and performance problems. A compelling illustration is SuperFetch<sup>4</sup> in Windows Vista, which monitor a user and builds a predictive model that the OS uses to fetch applications from the disk to memory, so, for example, Outlook starts instantly when I check my mail first thing in the morning.

To be honest, these ideas are only a starting point. Many of the new interface modalities have been studied for decades and are still not ready for widespread use. Video and animation may not be useful or appealing in many situations. Monitoring probably should not consume more resources than accomplishing a task, so it may only keep half of the processors busy.

Providing better ways to write parallel programs is the second challenge. The existing parallel programming model of threads and explicit synchronization is inadequate, as it is difficult and error-prone, even for expert programmers. Its constructs are entirely analogous to assembly language, both in their direct connections with hardware mechanisms and in the difficulty of using them systematically and correctly.

Future programming languages and tools will need to support all three common parallel programming models (data parallelism, task parallelism, and message passing). A given language may not directly support more than one model, but it must be possible to compose components written with different models, in different languages into a single application. Non-local tools will need to support all models, so a developer explore and understand an application's aggregate behavior.

Data parallelism is a regular, understandable, and predictable programming model, which has had considerable success at fine granularity (e.g., numeric computation on arrays) and extremely coarse granularity (e.g., Google's Map-Reduce). Language and library support is necessary both to bridge the middle granularity appropriate for Multicore and to enable non-numeric computations. As an example, programming language need an effect system to specify a computation's side-effects, so data parallel operations can be written safely in non-functional languages.

Task parallelism offers few constraints on how parallelism is structured or applied to a problem, but at the same time, the generality of the model makes it difficult to impose program structure or systematically detect errors. Better programming languages and constructs can improve this situation. For example, Transactional Memory<sup>5</sup> replaces low-level mutual exclusion with the *atomic* construct that expresses a developer's intent rather than error-prone implementation details. Other language

---

<sup>4</sup> <http://www.microsoft.com/windows/products/windowsvista/features/details/superfetch.mspx>

<sup>5</sup> Larus, J. R. and R. Rajwar (2006). *Transactional Memory*, Morgan & Claypool.

constructs, such as better effect systems or specification of data structure mutability, could help prevent errors due to data sharing.

Message passing provides a high level of isolation between concurrent tasks, but relies on less flexible communications primitives. In the Singularity<sup>6</sup> system at Microsoft Research, we have found that explicit channels annotated with a specification of the communication protocol alleviate some of these difficulties by enabling tools to find systematically minor message-passing errors.

Moreover, it is unclear that programming languages are an effective way to expose parallelism to the vast majority of developers, whose background and training make it difficult for them to learn a new programming paradigm. An appealing alternative is libraries that hide their parallel implementation details and offer a sequential interface. Even if performance is lost in alternating between sequential and parallel execution, retaining the familiar programming model would allow developers to use Multicore without retraining.

Beyond programming languages, Multicore processors require a new infrastructure of parallel run-time systems, parallel libraries, and parallel debugging and performance tools. The existing infrastructure treats parallelism as an afterthought and focuses on the sequential aspects of computation. This relationship must be reversed for Multicore programmers.

This discussion has only touched on part of the research and development required for Multicore. For example, computer architecture should consider alternative architectures that offer a more usable programming model. Operating systems need to evolve to support increasingly heterogeneous systems that execute several parallel applications simultaneously. User interfaces need to support a more asynchronous, less sequential style of interaction between a user and parallel applications.

---

<sup>6</sup> <http://research.microsoft.com/os/singularity/>