The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard

Peter Van Roy
Department of Computing Science and Engineering
Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

The era of programming with single processors has ended. Decades of prophesies have at last come true: programming with multiple processors has now entered the mainstream. Two forces have caused this transition to happen now. First, the Internet, which is a network of many loosely coupled processors. It had been gaining relevance for many years, but only recently has it achieved sufficient bandwidth and reliability to permit real distributed applications. The second force is the emergence of multi-core processors. Each of these forces brings a challenge for developers, but the two challenges are completely different in nature.

1. The challenge of multi-core processors

A multi-core processor combines two or more processing elements (called cores) in a single package, on a single die or multiple dies. The cores share the interconnect to the rest of the system and often share on-chip cache memory. The challenge of programming multi-core processors is real, but it is not a technical challenge. It is a purely sociological challenge. Technically, we have known since the 1980s how to program multi-core processors (in the guise of shared-memory multiprocessors) and how to write programs for them (in terms of parallel algorithms). There is a simple, natural, and powerful approach for programming these machines: dataflow programming. Many languages and systems implement this approach (see, e.g., Wikipedia for a long list). They are descendants of the venerable Id, Id Nouveau, SISAL, and other early dataflow languages. Google's well-publicized MapReduce is one of the most popular new tools that takes advantage of dataflow ideas [1], but these ideas are not new. In fact, they date from the 1970s [2]. A good exposition is given in chapter 4 of [3]. The basic insight is that there exists a form of concurrent programming, deterministic concurrency, that has no race conditions, is as easy to program as sequential programs, and can exploit parallel processors as a bonus. Deterministic concurrency is enjoying a renaissance thanks to clusters and multi-core processors.

2. The challenge of loosely coupled systems

A loosely coupled networked system consists of a set of processors and a network connecting them. It is useful to generalize this to a set of agents, which includes humans as well as processors. The challenge is how to get these independent networked agents (processors and humans) to collaborate and coordinate with each other in real time. The first problem is that no one agent has global knowledge (there is no "God's-eye view" of the whole system inside the system). The only way to know what another agent is doing is to ask (send a message and wait for an answer). In the meantime, the agent might have changed what it is doing. The second problem is partial failure: an agent might fail (leave the system or start behaving strangely). The other agents should somehow find out about this and compensate, so that the system still works. Both global knowledge and partial failure are low-level technical problems. We can overcome them (to some degree) by using the right algorithms, such as clock synchronization, distributed snapshots, and fault tolerance.

There was a real proliferation of work in distributed algorithms in the 1990s, leading to a deep understanding of how to solve the problems of lack of global knowledge and partial failure [4,5]. Here is a sample of what these algorithms can do. We now understand how to make an all-or-none broadcast (all receive or none receive) that works even though there might be processor failures during the algorithm. We understand how to make consensus (agreement among many parties) even though there may be communication problems or processor failures during the algorithm. We understand how to find agreement when there are malicious agents that do their best to sabotage the algorithm (Byzantine agreement, the best possible algorithm, can achieve agreement only when strictly less than 1/3 of agents are malicious). There are dozens of variations on these algorithms, depending on different communication models, failure models, and algorithm requirements. At this point, the reader might ask, with all these algorithms to choose from, how should I design my system? In fact, there is a simple answer that is often right: build it as a decentralized system!

2.1 Decentralized systems

A good way to build a loosely coupled system is as a decentralized system. That is, each computing node is by default independent of all the others. Each computing node contains the whole application and works even if there is no communication whatsoever between nodes. The system is then extended so that each node can use information from other nodes when it is available. Two important parts of such a design are the split protocol and the merge protocol. Split defines what happens when a connected node no longer communicates with other nodes, and merge defines how two independent nodes become connected again. The merge protocol is based on data coherence and may need input from the highest level of the system (e.g., human users) to resolve coherence issues. Based on this idea, we are building a general application framework for decentralized systems in the SELFMAN project [6]. The framework consists of a structured peer-to-peer storage layer with a transaction protocol built on top. The transaction protocol uses the Paxos algorithm, a distributed uniform consensus algorithm, to ensure it works well on the Internet.

Another example of a good decentralized system design is the Mercurial version control system [7]. Mercurial is a tool for software development by a team. Each team member has a local copy of the whole source code repository and can work in isolation. Different nodes can be merged at any time, which combines the work of different team members. New nodes can be created at any time by cloning and given to new team members. In this way, Mercurial supports software development by a team whose membership can change rapidly and whose Internet connectivity is highly irregular.

2.2 Conflicting goals

Loosely coupled systems have problems at a higher level than the simple technical problems of global knowledge and partial failure. These are the high-level problems of conflicting goals and emergent behavior. The first appears in peer-to-peer file sharing: in that setting it is sometimes called the "freeloader problem". To solve it, you need to build the system so that each agent's goals overlap with the overall system's goals. Designing a system in this way is not easy and almost always requires some adjustment during the system's deployment. The BitTorrent family of protocols and tools is a good example from computing. BitTorrent allows people to download and share large files, increasing performance and reliability by using collaboration [8]. The incentive scheme in BitTorrent is tuned so that freeloaders are discouraged.

2.3 Emergent behavior

The second high-level problem, emergent behavior, is not really a problem. It is an opportunity. Emergent behavior is what happens when the system as a whole shows novel behavior that is not shared by any of its parts. All complex systems show emergent behavior. For example, a single note from a Beethoven symphony impacts the listener as a nondescript sound, but brought together in the way Beethoven intended, the notes can impact the listener at a higher level (emotional and intellectual). Google (again) gives us a good example from computing: a Web page, when taken by itself, is hard to evaluate regarding its usefulness, correctness, and popularity. But taken together, all Web pages do give useful information, which can be extracted with the PageRank algorithm [9]. The apparent intelligence of Google Search is an emergent property.

3. Conclusions

Computing with multiple processors has finally emerged into the mainstream. In this position paper, I give a brief overview of the main challenges. There are two forms of multiple processor computing: shared-memory (e.g., multi-core) processors and loosely coupled processors (e.g., Internet), each with its own challenges. For multi-core processors, the main challenges are sociological. The technical problems were all solved long ago with the invention of dataflow programming. What remains is to educate programmers and to bring dataflow ideas into mainstream languages. For loosely coupled processors, the challenges are those of distributed systems: lack of global knowledge, partial failure, conflicting goals, and emergent behavior. These challenges pose technical problems, but they are also opportunities. I recommend that a loosely coupled system should always be designed to be decentralized by default, with collaboration between nodes added afterwards.

Acknowledgements

The author thanks Raphaël Collet for suggesting to use Mercurial as an example of a decentralized system. This work is funded by the European Union in the SELFMAN project (6FP contract 34084).

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, Dec. 2004.

[2] Kahn, Gilles. "The Semantics of a Simple Language for Parallel Programming." In *IFIP Congress*, pages 471–475, Aug. 1974.

[3] Van Roy, Peter, and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, 2004.

[4] Lynch, Nancy. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, 1996.

[5] Guerraoui, Rachid, and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, 2006.

[6] Van Roy, Peter, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefani, Roland Yap, and Thierry Coupaye. "Self Management for Large-Scale Distributed Systems: An Overview of the SELFMAN Project." Submitted for publication, Oct. 2007. At: www.ist-selfman.org.

[7] Mercurial distributed version control system. At: www.selenic.com/mercurial/wiki/.

[8] Cohen, Bram. "Incentives Build Robustness in BitTorrent." In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[9] Brin, Sergey, and Larry Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." *Computer Networks and ISDN Systems*, 30 (1-7), pages 107–117, April 1998.