Is Parallel Programming Hard, And If So, Why?

Paul E. McKenney
IBM Linux Technology Center
paulmck@linux.vnet.ibm.com

Manish Gupta
IBM India Research Laboratory
mgupta@us.ibm.com

Maged M. Michael
IBM Thomas J. Watson Research Center
magedm@us.ibm.com

Phil Howard, Joshua Triplett, Jonathan Walpole
Computer Science Department
Portland State University
pwh@cecs.pdx.edu,{tripletj,walpole}@cs.pdx.edu

ABSTRACT

Of the 200+ parallel-programming languages and environments created in the 1990s, almost all are now defunct. Given that parallel systems are now well within the budget of the typical hobbyist or graduate student, it is not unreasonable to expect a new cohort in excess of several *thousand* parallel languages and environments to appear in the 2010s. If this expected new cohort is to have more practical impact than did its 1990s counterpart, a robust and widely applicable framework will be required that encompasses exactly what, if anything, is hard about parallel programming. This paper revisits the fundamental precepts of concurrent programming to outline such a framework.

1. INTRODUCTION

Whatever difficulties might be faced by parallel programmers cannot be blamed on a dearth of parallel programming languages and environments, given that Mattson [9] lists more than 200 developed in the 1990s. As noted above, the 2010s might well see thousands of such languages. This situation cries out for a conceptual framework for parallel programming, as such a framework will be needed to select the most effective languages and environments.

However, such a framework is even more urgently needed to guide the creators of parallel-programing languages and environments. Although there are many loud assertions that parallel programming is extremely difficult, if not in fact impossible, there has been relatively little focus on identifying precisely what is difficult about parallel programming in general, as opposed to what is difficult about specific parallel programming methodologies. It is important to note that parallel programming cannot be analyzed purely theoretically, but must also take human factors into account.

In a perfect world, such a framework would therefore be constructed based on human factors studies [4], however, such studies on parallel programming are quite limited [3, 4, 7, 16]. This paper therefore takes a different approach, basing an evaluation and comparison framework on the tasks required for parallel programming over and above those required for sequential programming. Design and evaluation of parallel programming languages and environments can then be based on assisting developers with these tasks.

One challenge to any such framework is the many styles of parallel programming. We address this challenge drawing on different fields: Paul from operating-system kernels, Manish from distributed computing, Maged from parallel algorithms, and Jon from parallel video applications. We enlisted Josh and Phil, both Ph.D. candidates, to combat expert's bias. This paper distills the resulting framework from a series of often-spirited discussions within this group. We believe this framework will prove generally applicable.

It is also important to note that we believe that parallel programming is feasible, and hold up the large number of parallel operating systems, databases, and scientific applications as evidence. We find it particularly impressive that a number of these examples were created as open-source software projects, which leads us to believe that the requisite skills are readily available. Open-source projects have the additional advantage that their entire source base may be analyzed at any time by anyone, providing unprecedented opportunities for parallel-programming self-education.

Section 2 gives a high-level summary of the framework, Section 3 provides more detail on the capabilities defined by this framework, Section 4 applies the framework to a few heavily used parallel programming languages and environments, and finally, Section 5 presents concluding remarks.

2. EVALUATION AND COMPARISON FRAMEWORK OVERVIEW

Parallel languages and environments must provide four basic capabilities, and their overall effectiveness may be measured in three dimensions. These capabilities and measures of effectiveness are introduced in the following sections.

2.1 Parallel Capabilities

The four parallel capabilities are work partitioning, parallel access control, resource partitioning, and interaction with hardware. Note that these capabilities need not be fully controlled by the developer. On the contrary, it can be quite beneficial to embed these capabilities into the programming language or environment so as to ease the developer's job, in turn increasing productivity.

Parallel execution cannot occur without *work partitioning*, which ranges from the extremely fine-grained execution of superscalar CPUs to the equally coarse-grained structure of SETI@HOME. Each partition of work is termed a "thread".

In sequential code, the sole thread owns all resources. In contrast, partitioning work across multiple threads requires *parallel access control* to prevent destructive interference among concurrent threads. Resources include data structures and hardware devices, and the partitioning may either be static (as in classic distributed shared-nothing systems) or dynamic (as in transactional systems). Parallel access control concerns both the syntax of the accesses and the semantics of the underlying synchronization mechanisms.

The best performance and scalability is obtained via *resource partitioning* [2], permitting parallelism to increase with problem size. Such partitioning may be strict, or may involve replication of some or all of the relevant resources.

Finally, it is necessary to *interact with hardware*. This capability is often implemented in compilers and operating systems, thus allowing developers to ignore it. However, special-purpose hetergeneous multicore hardware may grow in importance given the limited growth in single-threaded performance of general-purpose CPUs.

These four capabilities, work partitioning, parallel access control, resource partitioning, and interacting with hardware, are required of all parallel languages and environments. However, design and evaluation of these capabilities requires measures of effectiveness, as discussed in the following section.

2.2 Parallel Measures of Effectiveness

The three measures of effectiveness are performance, productivity, and generality.

The primary measure of effectiveness for a parallel language or environment is *performance*. After all, if performance is not an issue, why worry about parallelism? Performance is often measured by its proxy, *scalability*, and often normalized against some input, as in performance per watt. Some may object that software reliability is always more important than performance, which is true, but no more so than in sequential programs. Furthermore, true fault tolerance requires a globally applied design discipline. Although non-blocking primitives and transactions avoid inter-thread synchronization dependencies, they are no substitute for the discipline required to ensure that any failed work will eventually be completed.

Furthermore, as the cost of hardware decreases relative to the cost of labor, the importance of *productivity* continues to increase. We believe that this increasing importance of productivity will motivate significant automation, which has in fact existed for some decades in the form of SQL, which often requires little or no effort to run efficiently on a parallel machine. Productivity concerns will also increase the importance of compatibility with legacy software, because reuse of existing software can be extremely productive.

Finally *generality* is important, as the development, maintenance, and training costs of a more-general language or environment may be amortized over a greater number of end users, reducing the per-user development cost.

Although the ideal parallel language or environment would offer optimal performance, high productivity, and full generality, this nirvana simply does not exist. A given language or environment rates highly on at most two of these three measures. For example, highly productive environments such as Ruby on Rails typically are tailored to a specific application area. Similarly, highly specialized environments can take advantage of powerful optimizations and parallelization techniques that are not generally applicable. Generality and productivity are often in direct conflict, as illustrated by Figure 1. High productivity seems to require a small abstract distance between the environment and the user (SQL, spreadsheets, Matlab, etc.), while broad generality seems to require that the environment remain close to either the hardware (C, C++, Java, assembly language) or some abstraction (Haskell, Prolog, Snobol).

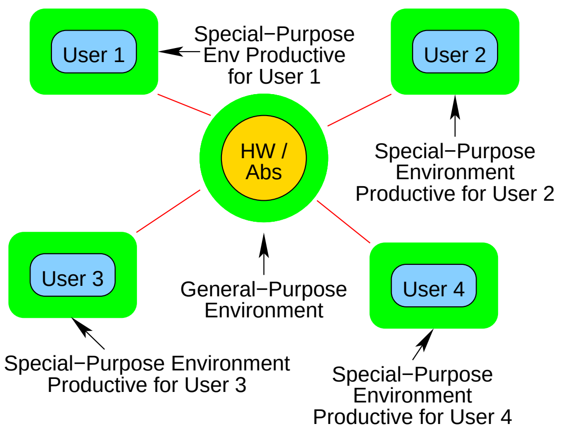
User 1
Special−Purpose Env Productive for User 1
User 2
HW / Abs
Special−Purpose Environment Productive for User 2
User 3
General−Purpose Environment
User 4
Special−Purpose Environment Productive for User 3
Special−Purpose Environment Productive for User 4

Figure 1: Generality vs. Productivity

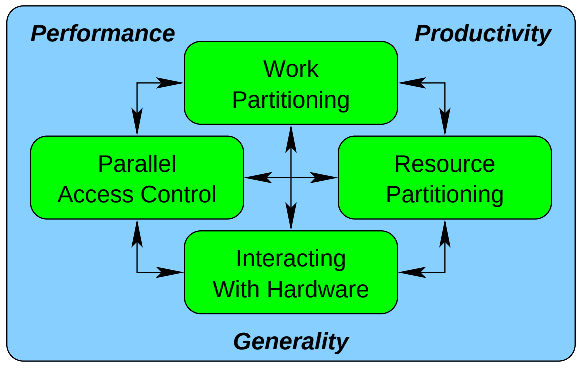

Figure 2: Parallel Evaluation Framework

2.3 Framework Summary

Our evaluation framework includes three measures of effectiveness (performance, productivity, generality) and four capabilities (work partitioning, parallel access control, resource partitioning, interacting with hardware), as shown in Figure 2. The next section fleshes out these capabilities.

3. CAPABILITIES

Each of the following sections focuses on a few of the issues surrounding each of the capabilities introduced in Section 2.

3.1 Work Partitioning

Parallel execution requires that work be partitioned, but partitioning requires great care. For example, partitioning the work unevenly can result in single-threaded execution once the small partitions have completed [1]. In less extreme cases, load balancing may be required to fully utilize available hardware so as to attain maximum performance.

In addition, partitioning of work can complicate handling of global errors and events: a parallel program may need to carry out non-trivial synchronization in order to safely proceed with such global processing.

Partitioning work requires some sort of communication: after all, if a given thread did not communicate at all, it would have no effect and would thus not need to be executed. However, communication incurs overhead, so that careless choices of partition boundaries can result in severe performance degradation.

Furthermore, the number of concurrent threads must often be controlled, as each such thread occupies common resources, for example, space in CPU caches. If too many threads execute concurrently, the CPU caches will overflow, resulting in high cache miss rate, which in turn degrades performance. On the other hand, large numbers of threads are often required to overlap computation and I/O.

Finally, permitting threads to execute concurrently greatly increases the program's state space, which can make the program difficult to understand, degrading productivity. All else being equal, smaller state spaces having more regular structure are more easily understood, but this is a human-factors statement as opposed to a technical or mathematical statement. Good parallel designs might have extremely large state spaces, but nevertheless be easy to understand due to their regular structure, while poor designs can be impenetrable despite having a comparatively small state space. The best designs exploit embarrassing parallelism, or transform the problem to one having an embarrassingly parallel solution. In either case, "embarrassingly parallel" is in fact an embarrassment of riches. The current state of the art enumerates good designs; more work is required to make more general judgements on state-space size and structure.

3.2 Parallel Access Control

Resources are most often in-memory data structures, but can be CPUs, memory (including caches), I/O devices, computational accelerators, files, and much else besides.

The first parallel-access-control issue is whether the form of the access to a given resource is a function of location. For example, in many message-passing environments, local-variable access is via expressions and assignments, while remote-variable acess is via special syntax involving messaging. The POSIX threads environment [15], Structured Query Language (SQL) [8], and partitioned global address-space (PGAS) environments such as Universal Parallel C (UPC) [5] offer implicit access, while Message Passing Interface (MPI) [14] offers explicit access because access to remote data requires explicit messaging.

The other parallel access-control issue is how threads coordinate access to the resources. This coordination is carried out by the very large number of synchronization mechanisms provided by various parallel languages and environments, including message passing, locking, transactions, reference counting, explicit timing, shared atomic variables, and data ownership. Many traditional parallel-programming concerns such as deadlock, livelock, and transaction rollback stem from this coordination. This framework can be elaborated to include comparisions of these synchronization mechanisms, for example locking vs. transactional memory [10], but such elaboration is beyond the scope of this paper.

3.3 Resource Partitioning

The most effective parallel algorithms and systems exploit resource parallelism, so much so that it is usually wise to begin parallelization by partitioning your resources. The resource in question is most frequently data, which might be partitioned over computer systems, mass-storage devices, NUMA nodes, CPU cores (or dies or hardware threads), pages, cache lines, instances of synchronization primitives, or critical sections of code. For example, partitioning over locking primitives is termed "data locking" [2].

Resource partitioning is frequently application dependent, for example, numerical applications frequently partition matrices by row, column, or sub-matrix, while commercial applications frequently partition write-intensive data structures and replicate read-mostly data structures. For example, a commercial application might assign the data for a given customer to a given few computer systems out of a large cluster. An application might statically partition data, or dynamically change the partitioning over time.

Resource partitioning is extremely effective, but it can be quite challenging for complex multilinked data structures.

3.4 Interacting With Hardware

Hardware interaction is normally the domain of the operating system, the compiler, libraries, or other software-environment infrastructure. However, developers working with novel hardware features and components will often need to work directly with such hardware. In addition, direct access to the hardware can be required when squeezing the last drop of performance out of a given system. In this case, the developer may need to tailor or configure the application to the cache geometry, system topology, or interconnect protocol of the target hardware.

In some cases, hardware may be considered to be a resource which may be subject to partitioning or access control, as described in the previous sections.

3.5 Composite Capabilities

Although these four capabilities are fundamental, good engineering practice uses composites of these capabilities. For example, the data-parallel approach first partitions the data so as to minimize the need for inter-partition communication, partitions the code accordingly, and finally maps data partitions and threads so as to maximize throughput while minimizing inter-thread communication. The developer can then consider each partition separately, greatly reducing the size of the relevant state space, in turn increasing productivity. Of course, some problems are non-partitionable but on the other hand, clever transformations into forms permitting partitioning can greatly enhance both performance and scalability [12].

The next section shows how this framework may be applied to three heavily used parallel-programming paradigms.

4. CASE STUDIES

This section applies the framework to the "locking plus threads" (L+T), MPI [14], and SQL [8] parallel-programming environments, all three of which have seen significant use in production, and all three of which have large and vital developer and user communities. Please note that L+T covers environments ranging from the Linux kernel [17] to POSIX pthreads [15] to Windows threads [13]. In addition, L+T has been extended by addition of atomic operations, fine-grained synchronization, read-copy update (RCU) [11], and transactional memory [6].

4.1 Measures of Effectiveness

All three environments can provide excellent performance, however, SQL is specialized while L+T and MPI are general purpose. The following sections therefore focus on productivity on a task-by-task basis, given that the easiest work is that which need not be done at all.

4.2 Work Partitioning

The L+T and MPI environments place almost all of the work-partitioning tasks on the shoulders of the developer, who must manually partition the work, control utilization (although in MPI, provisioning a fixed number of threads per CPU trivializes this task), and manage the state space. However, both environments provide primitives that propagate global errors (`abort()` for L+T and `MPI_Abort` for MPI), and most L+T environments provide a scheduler that automatically load-balances across CPUs.

In contrast, the SQL environment automates all work-partitioning tasks. This is not to say that SQL is perfect, in fact, a quick Google of "SQL performance" will return tens of millions of hits. However, the contrast with the L+T and MPI environments is quite sobering: SQL demands much less of the developer than do the other two environments.

4.3 Parallel Access Control

The L+T and the SQL environments boast implicit resource access, while MPI must use explicit messages to access remote data. All three environments require manual access control, with L+T providing locking, reference counting, and shared variables; MPI providing message passing, and SQL providing transactions. However, SQL's transactions are typically implemented using underlying synchronization primitives that are managed fully automatically.

4.4 Resource Partitioning

The L+T environment requires manual partitioning of data over critical sections, storage devices, pages, cache lines, and (best!) synchronization-primitive instances. That said, augmenting L+T with RCU can eliminate the need for partitioning of read-mostly data structures. The MPI environment requires manual partitioning of data over the computer systems in the cluster, and possibly over CPUs, cores, or sockets within each system.

In contrast, some SQL environments will automatically partition data as needed. That said, most SQL environments provide data-placement hints, which is one reason for the many Google hits on "SQL performance". Nevertheless, SQL is an illuminating demonstration of parallel-programming automation.

4.5 Interacting With Hardware

In principle, all of these three environments are machine independent, so that the programmer need not be concerned with hardware interactions. In practice, this laudable principle does not always apply. High-performance L+T applications may need to access variables outside of locks, in which case the developer may need to interact with the memory-ordering model of the system in question. Extreme MPI installations may require special high-performance hardware interconnects. And high-end SQL applications may require attention to the detailed properties of mass-storage systems.

Nevertheless, it is possible to write competent production-quality applications in each of these enviroments without special hardware interactions. This situation might well change should computational accelerators such as GPUs continue to increase in popularity.

4.6 Case Study Summary

Table 1 provides a summary of the prior sections. In this table, capital letters indicate advantages, lower-case letters indicate disadvantages and empty cells indicate non-applicability. The definitions of the letters are as follows: "A" indicates automatic, "e" indicates explicit differentiation between local and remote resource references, "h" indicates weakly automatic augmented by manual hints, "H" indicates automatic optionally augmented by manual hints, "I" indicates implicit access to remote resource references, "m" indicates manual operation, "M" indicates manual operation possible, but usually unnecessary.

	L+T	MPI	SQL
Work Partitioning	m	m	A
Error Processing	A	A	A
Global Processing	m	m	A
Thread Load Balancing	M	M	A
Work Item Load Balancing	m	m	A
Affinity to Resources	m	m	A
Control of Utilization	m	M	A
Parallel Access Control			
Implicit vs. Explicit	I	e	I
Message Passing	m	m	A
Locking	m		A
Transactions			m
Reference Counting	m		A
Shared Variables	m		A
Ownership	m	A	A
Resource Partitioning			
Over Systems		m	H
Over NUMA Nodes	m	m	A
Over CPUs/Dies/Cores	A	A	A
Over Critical Sections	m		A
Over Synchronization Primitives	m		H
Over Storage Devices	m	m	h
Over Pages and Cache Lines	m	m	A
Interacting With Hardware	A	A	A

Table 1: Summary of Framework Case Study (High Productivity in Capitals, Low Productivity in Lowercase)

While the L+T and MPI environments require developers to deal with parallelism, SQL automates almost all of the process, thus providing both performance and productivity. On the other hand, SQL is specific to applications using relational databases, while L+T and MPI are general-purpose environments. Therefore, this table indicates that developers should use SQL in preference to L+T or MPI, but only within SQL's domain. This situation does much to explain the common practice of using multiple parallel programming models in large data-center applications.

5. CONCLUSIONS

We have constructed a novel framework identifying why parallel programming is more difficult than is sequential programming. We further demonstrated the generality of this framework via a case study comparing three diverse parallel programming environments: "locking plus threads", MPI, and SQL. The framework divides these difficulties into four categories: work partitioning, parallel access control, resource partitioning, and, in rare cases, interacting with special-purpose parallel hardware, none of which are required for sequential programming. However, these difficulties need not necessarily be addressed manually. For example, for the special case of read-mostly data structures, RCU delegates resource partitioning to the underlying hardware cache-consistency protocol. SQL goes much farther, providing an impressive example of pervasive parallel automation. Parallel programming is therefore as hard, or as easy, as we design it to be.

Unfortunately, SQL is not a general-purpose solution, being restricted to database queries and updates. However, we do not know of any parallel programming language or environment that is world class on more than two of the three parallel measures of effectiveness: performance, productivity, and generality. Therefore, researchers interested in highly productive parallel languages and environments should prioritize performance over generality. Similarly, researchers interested in general-purpose parallel programming languages and environments should prioritize performance over productivity, focusing on the needs of expert developers constructing the low-level infrastructure supporting widely used programming environments. It is conceivable that parallel languages and environments might provide world-class productivity and generality at the expense of performance, but such environments will not be interesting unless they demonstrate significant benefits beyond those of existing sequential programming environments.

Of course, we cannot rule out the possibility of a parallel programming language or environment that provides world-class performance, productivity, *and* generality. However, the historic tendency of highly productive environments (such as spreadsheets, databases, and computer-aided design systems) to also be highly specialized augurs against such a nivana.

Acknowledgements

We are indebted to Bob Blainey, Sam Adams, Chet Murthy, and Cait Crawford for many valuable discussions and to Daniel Frye and Kathy Bennett for their support of this effort.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.

6. REFERENCES

[1] Amdahl, G. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings* (Washington, DC, USA, 1967), IEEE Computer Society, pp. 483–485.

[2] Beck, B., and Kasten, B. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 255–275.

[3] Eccles, R., Nonneck, B., and Stacey, D. A. Exploring parallel programming knowledge in the novice. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 97–102.

[4] Eccles, R., and Stacey, D. A. Understanding the parallel programmer. In *HPCS '05: Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 156–160.

[5] El-Ghazawi, T. A., Carlson, W. W., and Draper, J. M. UPC language specifications v1.1. Available: `http://upc.gwu.edu`, May 2003.

[6] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. *The 20th Annual International Symposium on Computer Architecture* (May 1993), 289–300.

[7] Hochstein, L., Carver, J., Shull, F., Asgari, S., and Basili, V. Parallel programmer productivity: A case study of novice parallel programmers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2005), IEEE Computer Society, p. 35.

[8] International Standards Organization. *Information Technology - Database Language SQL.* ISO, 1992.

[9] Mattson, T. Scalable software for many core chips: programming Intel's 80-core research chip. Available: `http://www.psc.edu/seminars/PAS/Mattson.pdf`, June 2007.

[10] McKenney, P. E., Michael, M., and Walpole, J. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. In *Programming Languages and Operating Systems* (New York, NY, USA, October 2007), ACM SIGOPS, pp. 1–5.

[11] McKenney, P. E., and Walpole, J. What is RCU, fundamentally? Available: `http://lwn.net/Articles/262464/`, December 2007.

[12] Metaxas, P. T. Fast dithering on a data-parallel computer. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems* (Cambridge, MA, USA, 1999), IASTED, pp. 570–576.

[13] Microsoft Corporation. DLLs, processes, and threads. Available: `http://msdn.microsoft.com/en-us/library/ms682584(VS.85).aspx`, August 2008.

[14] MPI Forum. Message passing interface forum. Available: `http://www.mpi-forum.org/`, September 2008.

[15] Open Group. The single UNIX specification, version 2: Threads. Available: `http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html`, 1997.

[16] Szafron, D., and Schaeffer, J. Experimentally assessing the usability of parallel programming systems. In *IFIP WG10.3 Programming Environments for Massively Parallel Distributed Systems* (1994), pp. 19.1–19.7.

[17] Torvalds, L. Linux 2.6. Available: `ftp://kernel.org/pub/linux/kernel/v2.6`, August 2003.