DOI:10.1145/1743546.1743560

Peter J. Denning and Jack B. Dennis

The Profession of IT

The Resurgence of Parallelism

Parallel computation is making a comeback after a quarter century of neglect. Past research can be put to quick use today.

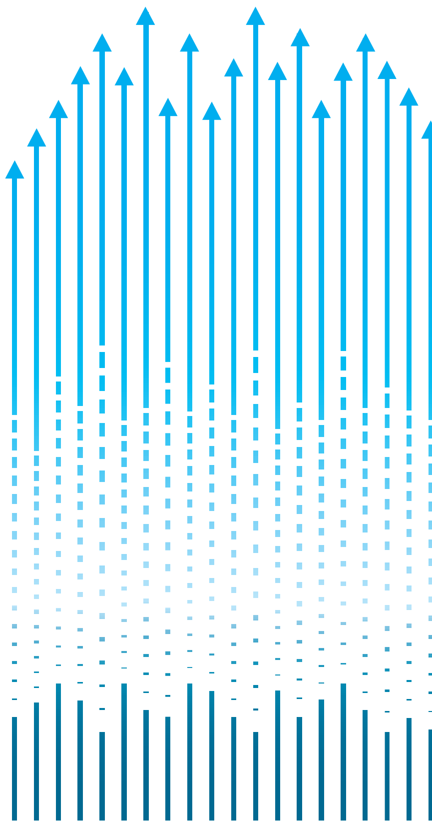
MULTI-CORE CHIPS ARE a new paradigm!" "We are entering the age of parallelism!" These are today's faddish rallying cries for new lines of research and commercial development. Is this really the first time when computing professionals seriously engaged with parallel computation? Is parallelism new? Is parallelism a new *paradigm*?

Déjà Vu All Over Again

Parallel computation has always been a means to satisfy our never-ending hunger for ever-faster and ever-cheaper computation.[4] In the 1960s and 1970s, parallel computation was extensively researched as a means to high-performance computing. But the commercial world stuck with a quest for faster CPUs and, assisted by Moore's Law, made it to the 2000s without having to seriously engage with parallel computation except for supercomputers. The parallel architecture research of the 1960s and 1970s solved many problems that are being encountered today. Our objective in this column is to recall the most important of these results and urge their resurrection.

Shared Memory Multiprocessing

The very first multiprocessor architecture was the Burroughs B5000, designed beginning in 1961 by a team led by Robert Barton. It was followed by the B5500 and B6700, along with a defense version, the D850. The architecture survives today in the reverse polish notation HP calculators and in the UniSys ClearPath MCP machines.



Those machines used shared memory multiprocessors in which a crossbar switch connected groups of four processors and memory boxes. The operating system, known as Automatic Scheduling and Operating Program (ASOP), included many innovations. Its working storage was organized as a stack machine. All its code was "reentrant," meaning that multiple processors could execute the same code simultaneously while computing on separate stacks. The instruction set, which was attuned to the Algol language, was very simple and efficient even by today's RISC standards. A newly spawned process's stack was linked to its parent's stack, giving rise to a runtime structure called "cactus stack." The data memory outside of the stacks was laid out in *segments*; a segment was a contiguous sequence of locations with base and bound defined by a *descriptor*. Segments were moved automatically up and down the memory hierarchy, an early form of virtual memory not based on paging. Elliot Organick's masterful descriptions of these machines make for refreshing and worthwhile reading today.[9,12]

The Burroughs systems were an important influence on research seeking efficient and reliable parallel program structures. A group of researchers at Brown University and General Electric Research Laboratories produced a set of reports on a "contour model" of nested multitask computations in 1971.[12] Those reports give a remarkably clear picture of a parallel programming runtime environment that would suit today's languages well and would resolve many contemporary problems considered as "research challenges." It is a tragedy these ideas have disappeared from the curriculum.

The Burroughs machines disappeared not because of any defect in their architecture, but because of IBM's massive success in marketing the 360 series systems. Moreover, in a process reminiscent of Clayton Christensen's *Innovator's Dilemma*, the low-end assembler-language minicomputers, originally designed to run laboratory instruments, grew up into the minicomputer and then microcomputer market, untainted by any notions of parallel programming.

With the introduction of RISC architectures in the early 1980s, much of the research for high-performance computers was rechanneled toward exploiting RISC for fast chips. It looked at the time that sophisticated compilers could make up for missing functions in the chips.

With two notable exceptions, most of the projects exploring alternatives to the "von Neumann architecture" expired and were not replaced with new projects or other initiatives. One exception was Arvind's Monsoon Project at MIT,[10] which demonstrated that massive parallelism is readily identified in the functional programming language Haskell, and then readily mapped to a shared memory multiprocessor. (Functional languages generate all their values by evaluating functions without side effects.)

The other project involved a group at the Lawrence Livermore National Laboratory studying scientific codes in the functional language Sisal, a derivative of MIT's Val language; Sisal programs were as efficient as Fortran programs and could be readily compiled to massively parallel shared memory supercomputers.[1,2,11]

The current generations of supercomputers (and data warehouses) are based on thousands of CPU chips running in parallel. Unlike the innovative designs of the Burroughs systems, their hardware architectures conform to the conventional von Neumann machine. Their operating systems are little more than simple schedulers and message passing protocols, with complex functions relegated to applications running on separate host machines.

The point is clear: ideas for arranging multiple processors to work together in an integrated system have been with us for 50 years. What's new?

The parallel architecture research of the 1960s and 1970s solved many problems that are being encountered today.

Determinate Computation

One of the holy grails of research in parallel computation in the 1960s and 1970s was called "determinacy."[7] Determinacy requires that a network of parallel tasks in shared memory always produces the same output for given input regardless of the speeds of the tasks. It should not be confused with a similar word, "deterministic," which would require that the tasks be ordered in the same sequence every time the system runs.

A major result of this research was the "determinacy theorem." A task is a basic computation that implements a function from its inputs to outputs. Two tasks are said to be in conflict if either of them writes into memory cells used by the other. In a system of concurrent tasks, race conditions may be present that make the final output depend on the relative speeds or orders of task execution. Determinacy is ensured if the system is constrained so that every pair of conflicting tasks is performed in the same order in every run of the system. Then no data races are possible. Note that atomicity and mutual exclusion are not sufficient for determinacy: they ensure only that conflicting tasks are not concurrent, but not that they always executed in the same order.

A corollary of the determinacy theorem is that the entire sequence of values written into each and every memory cell during any run of the system is the same for the given input. This corollary also tells us that any system of blocking tasks that communicates by messages using FIFO queues (instead of shared memory) is automatically determinate because the message queues always present the data items in the same order to the tasks receiving them.

Another corollary is that an implementation of a functional programming language using concurrent tasks is determinate because the functions provide their data privately to their successors when they fire. There is no interference among the memory cells used to transmit data between functions.

Determinacy is really important in parallel computation. It tells us we can unleash the full parallelism of a computational method without worrying whether any timing errors or race conditions will negatively affect the results.

Functional Programming and Composability

Another holy grail for parallel system has been modular composability. This would mean that any parallel program can be used, without change, as a component of a larger parallel program.

Three principles are needed to enable parallel program composability. David Parnas wrote about two: information hiding and context independence. Information hiding means a task's internal memory cannot be read or written by any other task. Context independence means no part of a task can depend on values outside the task's internal memory or input-output memory. The third principle is argument noninterference; it says that a data object presented as input to two concurrent modules cannot be modified by either.

Functional programming languages automatically satisfy these three principles; their modules are thus composable.

It is an open question how to structure composable parallel program modules from different frameworks when the modules implement nondeterminate behavior. Transaction systems are an extreme case. Because their parallel tasks may interfere in the records they access, they use locking protocols to guarantee mutual exclusion. Transaction tasks cannot be ordered by a fixed order—their nondeterminacy is integral to their function. For example, an airplane seat goes to whichever task requested it first. The

problem is to find a way to reap the benefits of composability for systems that are necessarily nondeterminate.

Virtual Memory

There are obvious advantages if the shared memory of a parallel multiprocessor could be a virtual memory. Parameters can be passed as pointers (virtual addresses) without copying (potentially large) objects. Compilation and programming are greatly simplified because neither compilers nor programmers need to manage the placement of shared objects in the memory hierarchy of the system; that is done automatically by the virtual memory system.

Virtual memory is essential for modular composability when modules can share objects. Any module that manages the placement of a shared object in memory violates the information hiding principle because other modules must consult it before using a shared object. By hiding object locations from modules, virtual memory enables composability of parallel program modules.

There are two concerns about large virtual memory. One is that the virtual addresses must be large so that they encompass the entire address space in which the large computations proceed. The Multics system demonstrated that a very large virtual address space—capable of encompassing the entire file system—could be implemented efficiently.[8] Capability-based addressing[5,6] can be used to implement a very large address space.

The other concern about large virtual memory pertains to performance. The locality principle assures us that each task accesses a limited but dynamically evolving working set of data objects.[3] The working set is easily detected—it is the objects used in a recent backward-looking window—and loaded into a processor's cache. There is no reason to be concerned about performance loss due to an inability to load every task's working set into its cache.

Parallelism is not new; the realization that it is essential for continued progress in high-performance computing is.

What about cache consistency? A copy of a shared object will be present in each sharing task's cache. How do changes made by one get transmitted to the other? It would seem that this problem is exacerbated in a highly parallel system because of the large number of processors and caches.

Here again, the research that was conducted during the 1970s provides an answer. We can completely avoid the cache consistency problem by never writing to shared data. That can be accomplished by building the memory as a write-once memory: when a process writes into a shared object, the system automatically creates a copy and tags it as the current version. These value sequences are unique in a determinate system. Determinate systems, therefore, give a means to completely avoid the cache consistency problem and successfully run a very large virtual memory.

Research Challenges

Functional programming languages (such as Haskell and Sisal) currently support the expression of large classes of application codes. They guarantee determinacy and support composability. Extending these languages to include stream data types would bring hazard-free expression to computations involving inter-module pipelines and signal processing. We badly need a further extension to support programming in the popular object-oriented style while guaranteeing determinacy and composability.

We have means to express nondeterminate computation in self-contained environments such as interactive editors, version control systems, and transaction systems. We sorely need approaches that can combine determinate and nondeterminate components into well-structured larger modules.

The full benefits of functional programming and composability cannot be fully realized unless memory management and thread scheduling are freely managed at runtime. In the long run, this will require merging computational memory and file systems into a single, global virtual memory.

Conclusion

We can now answer our original questions. Parallelism is not new; the realization that it is essential for continued progress in high-performance computing is. Parallelism is not yet a paradigm, but may become so if enough people adopt it as the standard practice and standard way of thinking about computation.

The new era of research in parallel processing can benefit from the results of the extensive research in the 1960s and 1970s, avoiding rediscovery of ideas already documented in the literature: shared memory multiprocessing, determinacy, functional programming, and virtual memory.

References

1. Cann, D. Retire Fortran?: A debate rekindled. *Commun. ACM 35*, 8 (Aug. 1992), 81–89.
2. Cann, D. and Feo, J. SISAL versus FORTRAN: A comparison using the Livermore loops. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1990.
3. Denning, P. Virtual memory. *ACM Computing Surveys 2*, 3 (Sept. 1970), 153–189.
4. Denning, P. and Tichy, W. Highly parallel computation. *Science 250* (Nov. 30, 1990), 1217–1222.
5. Dennis, J. and Van Horn, E.C. Programming semantics for multi-programmed computations. *Commun. ACM 9*, 3 (Mar. 1966), 143–155.
6. Fabry, R. Capability-based addressing. *Commun. ACM 17*, 7 (July 1974), 403–412.
7. Karp, R.M. and Miller, R.E. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal of Applied Mathematics 14*, 6 (Nov. 1966), 1390–1411.
8. Organick, E.I. *The Multics System: An Examination of Its Structure*. MIT Press, 1972.
9. Organick, E.I. Computer systems organization: The B5700/B6700. *ACM Monograph Series*, 1973. LCN: 72-88334.
10. Papadopoulos, G.M. and Culler, D.E. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. (1990), 82–91.
11. Sarkar, V. and Cann, D. POSC—A partitioning and optimizing SISAL compiler. In *Proceedings of the 4th International Conference on Supercomputing*. IEEE Computer Society Press, 1990.
12. Tou, J. and Wegner, P., Eds. Data structures in programming languages. *ACM SIGPLAN Notices 6* (Feb. 1971), 171–190. See especially papers by Wegner, Johnston, Berry, and Organick.

Peter J. Denning (pjd@nps.edu) is the director of the Cebrowski Institute for Innovation and Information Superiority at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.

Jack B. Dennis (dennis@csail.mit.edu) is Professor Emeritus of Computer Science and Engineering at MIT and a Principal Investigator in the Computer Science and Artificial Intelligence Laboratory. He is an Eckert-Mauchly Award recipient and a member of the NAE.

Copyright held by author.