DOI:10.1145/1866739.1866757

The ICE abstraction may take CS from serial (single-core) computing to effective parallel (many-core) computing.

BY UZI VISHKIN

Using Simple Abstraction to Reinvent Computing for Parallelism

THE RECENT DRAMATIC shift from single-processor computer systems to many-processor parallel ones requires reinventing much of computer science to build and program the new systems. CS urgently requires convergence to a robust parallel general-purpose platform providing good performance and programming easy enough for all CS students and graduates. Unfortunately, ease-of-programming objectives have eluded parallel-computing research over at least the past four decades. The idea of starting with an established easy-to-apply parallel programming model and building an architecture for it has been treated as radical by hardware and software vendors alike. Here, I advocate an even more radical parallel programming and architecture idea: Start with a simple abstraction encapsulating the desired interface between programmers and system builders.

» key insights

- **Computing can be reinvented for parallelism, from parallel algorithms through programming to hardware, preempting the technical barriers inhibiting use of parallel machines.**
- **Moving beyond the serial von Neumann computer (the only successful general-purpose platform to date), computer science will again be able to augment mathematical induction with a simple one-line computing abstraction.**
- **Being able to think algorithmically in parallel is a significant advantage for systems developers and programmers building and programming multi-core machines.**

Parallel Algorithms

The following two examples explore how these algorithms look and the opportunities and benefits they provide to systems developers and programmers.

Example 1. **Given are two variables A and B, each containing some value. The exchange problem involves exchanging their values; for example, if the input to the exchange problem is A=2 and B=5, then the output is A=5 and B=2. The standard algorithm for this problem uses an auxiliary variable X and works in three steps:**

```
X:=A
A:=B
B:=X
```

In order not to overwrite A and lose its content, the content of A is first stored in X, B is then copied to A, and finally the original content of A is copied from X to B. The work in this algorithm is three operations, the depth is three time units, and the space requirement (beyond input and output) is one word.

Given two arrays A[0..n-1] and B[0..n-1], each of size *n*, the array-exchange problem involves exchanging their content, so A(i) exchanges its content with B(i) for every i=0..n-1. The array exchange serial algorithm serially iterates the standard exchange algorithm n times. Here's the pseudo-code:

```
For i =0 to n–1 do
      X:=A( i ) ; A( i ):=B( i ) ; B( i ):=X
```

The work is 3n, depth is 3n, and space is 2 (for X and i). A parallel array-exchange algorithm uses an auxiliary array X[0..n-1] of size n, the parallel algorithm applies concurrently the iterations of the serial algorithm, each exchanging A(i) with B(i) for a different value of i. Note the new pardo command in the following pseudo-code:

```
For i =0 to n–1 pardo
      X( i ):=A( i ) ; A( i ):=B( i ) ; B( i ):=X( i )
```

This parallel algorithm requires 3n work, as in the serial algorithm. Its depth has improved from 3n to 3. If the size of the array n is 1,000 words, it would constitute speedup by a factor of 1,000 relative to the serial algorithm. The increase in space to 2n (for array X and n concurrent values of i) demonstrates a cost of parallelism.

Example 2. **Given is the directed graph with nodes representing all commercial airports in the world. An edge connects node u to node v if there is a nonstop flight from airport u to airport v, and s is one of these airports. The problem is to find the smallest number of nonstop flights from s to any other airport. The WD algorithm works as follows: Suppose the first i steps compute the fewest number of nonstop flights from s to all airports that can be reached from s in at most i flights, while all other airports are marked "unvisited."**

Step i+1 concurrently finds the destination of every outgoing flight from any airport to which the fewest number of flights from s is exactly i, and for every such destination marked "unvisited" requires i+1 flights from s. Note that some "unvisited" nodes may have more than one incoming edge. In such a case the arbitrary CRCW convention implies that one of the attempting writes succeeds. While we don't know which one, we do know all writes would enter the number i+1; in general, however, arbitrary CRCW also allows different values.

The standard serial algorithm for this problem[9] is called breadth-first search, and the parallel algorithm described earlier is basically breadth-first search with one difference: Step i+1 described earlier allows concurrent-writes. In the serial version, breadth-first search also operates by marking all nodes whose shortest path from s requires i+1 edges after all nodes whose shortest path from s requires i edges. The serial version then proceeds to impose a serial order. Each newly visited node is placed in a first-in-first-out queue data structure.

Three lessons are drawn from this example: First, the serial order obstructs the parallelism in breadth-first search; freedom to process in any-order nodes for which the shortest path from s has the same length is lost. Second, programmers trained to incorporate such serial data structures into their programs acquire bad serial habits difficult to uproot; it may be better to preempt the problem by teaching parallel programming and parallel algorithms early. And third, to demonstrate the performance advantage of the parallel algorithm over the serial algorithm, assume that the number of edges in the graph is 600,000 (the number of nonstop flight links), and the smallest number of flights from airport s to any other airport is no more than five. While the serial algorithm requires 600,000 basic steps, the parallel algorithm requires only six. Meanwhile, each of the six steps may require longer wall clock time than each of the 600,000 steps, but the factor 600,000/6 provides leeway for speedups by a proper architecture.

I begin by proposing the Immediate Concurrent Execution (ICE) abstraction, followed by two contributions supporting this abstraction I have led:

XMT. A general-purpose many-core explicit multi-threaded (XMT) computer architecture designed from the ground up to capitalize on the on-chip resources becoming available to support the formidable body of knowledge, known as parallel random-access machine (model), or PRAM, algorithmics, and the latent, though not widespread, familiarity with it; and

Workflow. A programmer's workflow links ICE, PRAM algorithmics, and XMT programming. The ICE abstraction of an algorithm is followed by a description of the algorithm for the synchronous PRAM, allowing ease of reasoning about correctness and complexity, which is followed by multithreaded programming that relaxes this synchrony for the sake of implementation. Directly reasoning about soundness and performance of multithreaded code is generally known to be error-prone. To circumvent the likelihood of errors, the workflow incorporates multiple levels of abstraction; the programmer must establish only that multithreaded program behavior matches the synchronous PRAM-like algorithm it implements, a much simpler task. Current XMT hardware and software prototypes and demonstrated ease-of-programming and strong speedups suggest that CS may be much better prepared for the challenges ahead than many of our colleagues realize.

A notable rudimentary abstraction—that any single instruction available for execution in a serial program executes immediately—made serial computing simple. Abstracting away a hierarchy of memories, each with greater capacity but slower access time than its predecessor, along with different execution time for different operations, this Immediate Serial Execution (ISE) abstraction has been used by programmers for years to conceptualize serial computing and ensure support by hardware and compilers. A program provides the instruction to be executed next at each step (inductively). The left side of Figure 1 outlines serial execution as implied by this ISE

abstraction, where unit-time instructions execute one at a time.

The rudimentary parallel abstraction I propose here is that indefinitely many instructions available for concurrent execution execute immediately, dubbing the abstraction Immediate Concurrent Execution. A consequence of ICE is a step-by-step (inductive) explication of the instructions available next for concurrent execution. The number of instructions in each step is independent of the number of processors, which are not even mentioned. The explication falls back on the serial abstraction in the event of one instruction per step. The right side of Figure 1 outlines parallel execution as implied by the ICE abstraction. At each time unit, any number of unit-time instructions that can execute concurrently do so, followed by yet another time unit in which the same execution pattern repeats, and so on, as long as the program is running.

How might parallelism be advantageous for performance? The PRAM answer is that in a serial program the number of time units, or "depth," is the same as the algorithm's total number of operations, or "work," while in the parallel program the number of time units can be much lower. For a parallel program, the objective is that its work does not much exceed that of its serial counterpart for the same problem, and its depth is much lower than its work. (Later in the article, I note the straightforward connection between ICE and the rich PRAM algorithmic theory and that ICE is nothing more than a subset of the work-depth model.) But how would a system designer go about building a computer system that realizes the promise of ease of programming and strong performance?

Outlining a comprehensive solution, I discuss basic tension between the PRAM abstraction and hardware implementation and a workflow that goes through ICE and PRAM-related abstractions for programming the XMT computer architecture.

Some many-core architectures are likely to become mainstream, meaning they must be easy enough to program by every CS major and graduate. I am not aware of other many-core architectures with PRAM-like abstraction. Allowing programmers to view a computer operation as a PRAM would make it easy to program,[10] hence this article should interest all such majors and graduates.

Until 2004, standard (desktop) computers comprised a single processor core. Since 2005 when multi-core computers became the standard, CS has appeared to be on track with a prediction[5] of 100+-core computers by the mid-2010s. Transition from serial (single-core) computing to parallel (many-core) computing mandates the reinvention of the very heart of CS, as these highly parallel computers must be built and programmed differently from the single-core machines that dominated standard computer systems since the inception of the field almost 70 years ago. By 2003, the clock rate of a high-end desktop processor had reached 4GHz, but processor clock rates have improved only barely, if at all, since then; the industry simply did not find a way to continue improving clock rates within an acceptable power budget.[5] Fortunately, silicon technology improvements (such as miniaturization) allow the amount of logic a computer chip can contain to keep growing, doubling every 18 to 24 months per Gordon Moore's 1965 prediction. Computers with an increasing number of cores are now expected but without significant improvement in clock rates. Exploiting the cores in parallel for faster completion of a computing task is today the only way to improve performance of individual tasks from one generation of computers to the next.

Unfortunately, chipmakers are designing multi-core processors most programmers can't handle,[19] a problem of broad interest. Software production has become a key component of the manufacturing sector of the economy. Mainstream machines most programmers can't handle cause significant decline in productivity of manufacturing, a concern for the overall economy. Andy Grove, former Chairman of the Board of Intel Corp., said in the 1990s that the software spiral—the cyclic process of hardware improvements leading to software improvements leading back to hardware improvements—was an engine of sustained growth for IT for decades to come. A stable application-software base that could be reused and enhanced from one hardware generation to the next was available for exploitation. Better performance was assured with each new generation, if only the hardware could run serial code faster. Alas, the software spiral today is broken.[21] No broad parallel-computing application software base exists for which hardware vendors are committed to improving performance. And no agreed-upon parallel architecture allows application programmers to build such a base for the foreseeable future. Instating a new software spiral could indeed be a killer app for general-purpose many-core computing; application software developers would put it to good use for specific applications, and more consumers worldwide would want to buy new machines.

This robust market for many-core-based machines and applications leads to the following case for government support: Foremost among today's challenges is many-core convergence, seeking timely convergence to

Figure 1. Serial execution based on the serial ISE abstraction vs. parallel execution based on the parallel ICE abstraction.

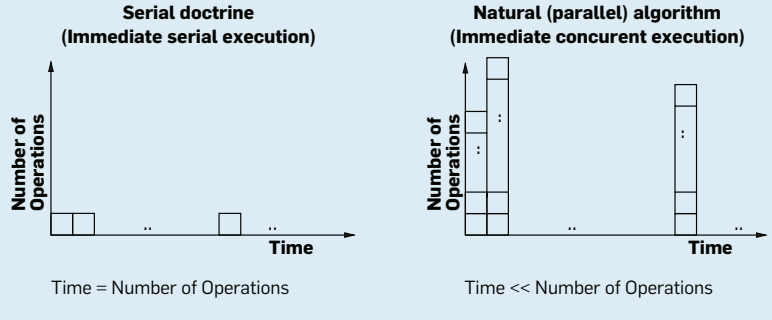
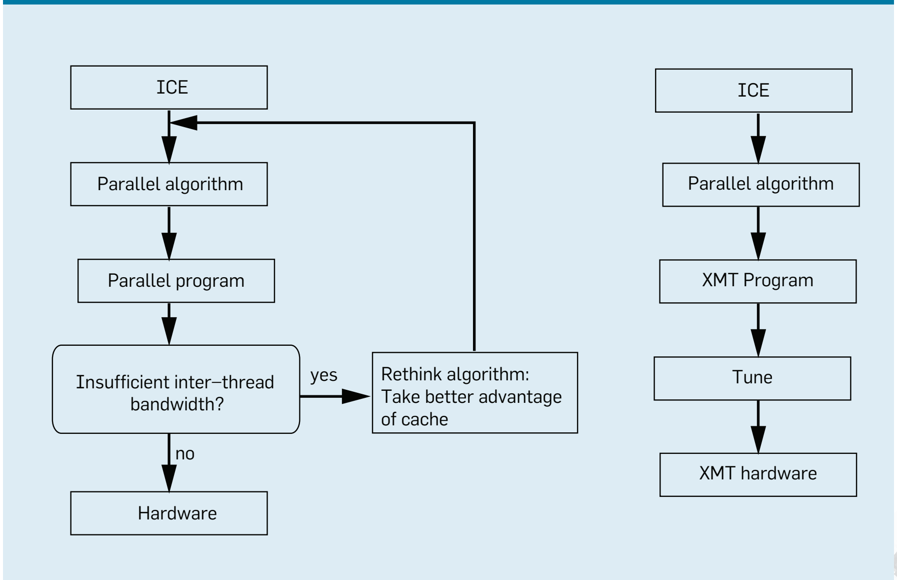


Figure 2. Right column is a workflow from an ICE abstraction of an algorithm to implementation; left column may never terminate.



a robust many-core platform coupled with a new many-core software spiral to serve the world of computing for years to come. A software spiral is basically an infrastructure for the economy. Since advancing infrastructures generally depends on government funding, designating software-spiral rebirth a killer app also motivates funding agencies and major vendors to support the work. The impact on manufacturing productivity could further motivate them.

Programmer Workflow

ICE requires the lowest level of cognition from the programmer relative to all current parallel programming models. Other approaches require additional steps (such as decomposition[10]). In CS theory, the speedup provided by parallelism is measured as work divided by depth; reducing the advantage of ICE/PRAM to practice is a different matter.

The reduction to practice I have led relies on the programmer's workflow, as outlined in the right side of Figure 2. Later, I briefly cover the parallel-algorithms stage. The step-by-step PRAM explication, or "data-parallel" instructions, represents a traditional tightly synchronous outlook on parallelism. Unfortunately, tight step-by-step synchrony is not a good match with technology, including its power constraints.

To appreciate the difficulty of implementing step-by-step synchrony in hardware, consider two examples: Memories based on long tightly synchronous pipelines of the type seen in Cray vector machines have long been out of favor among architects of high-performance computing; and processing memory requests takes from one to 400 clock cycles. Hardware must be made as flexible as possible to advance without unnecessary waiting for concurrent memory requests.

To underscore the importance of the bridge the XMT approach builds from the tightly synchronous PRAM to relaxed synchrony implementation, note three known limitations with power consumption of multi-core architectures: high power consumption of the wide communication buses needed to implement cache coherence; basic *nm* complexity of cache-coherence traffic (given *n* cores and *m* invalidations) and implied toll on inter-core bandwidth; and high power consumption needed for a tightly synchronous implementation in silicon in these designs. The XMT approach addresses all three by avoiding hardware-supported cache-coherence altogether and by significantly relaxing synchrony.

Workflow is important, as it guides the human-to-machine process of programming; see Figure 2 for two workflows. The non-XMT hardware implementation on the left side of the figure may require revisiting and changing the algorithm to fit bandwidth constraints among threads of the computation, a programming process that doesn't always yield an acceptable outcome. However, the XMT hardware allows a workflow (right side of the figure) that requires tuning only for performance; revisiting and possibly changing the algorithm is generally not needed. An optimizing compiler should be able to do its own tuning without programmer intervention, as in serial computing.

Most of the programming effort in traditional parallel programming (domain partitioning, load balancing) is generally of lesser importance for exploiting on-chip parallelism, where parallelism overhead can be kept low and processor-to-memory bandwidth high. This observation drove development of the XMT programming model and its implementation by my research team. XMT is intended to provide a simpler parallel programming model that efficiently exploits on-chip parallelism through multiple design elements.

The XMT architecture uses a high-bandwidth low-latency on-chip interconnection network to provide more uniform memory-access latencies. Other specialized XMT hardware primitives allow concurrent instantiation of as many threads as the number of available processors, a count that can reach into the thousands. Specifically, XMT can perform two main operations: forward (instantly) program instructions to all processors in the time required to forward the instructions (for one thread) to just one processor; and reallocate any number of processors that complete their jobs at the same time to new jobs (along with their instructions) in the time required to reallocate one processor. The high-bandwidth, low-latency interconnection network and low-overhead creation of many threads allow efficient support for the fine-grain parallelism used to hide memory latencies and a programming model for which locality is less an issue than in designs with less bandwidth. These mechanisms support dynamic load balancing, relieving programmers from having to directly assign work to processors. The programming model is simplified further by letting threads run to com-

pletion without synchronization (no busy-waits) and synchronizing access to shared data with prefix-sum (fetch-and-add type) instructions. These features result in a flexible programming style that accommodates the ICE abstraction and encourages program development for a range of applications.

The reinvention of computing for parallelism also requires pulling together a number of technical communities. My 2009 paper[26] sought to build a bridge to other architectures by casting the abstraction-centric vision of this article as a possible module in them, identifying a limited number of capabilities the module provides and suggesting a preferred embodiment of these capabilities using concrete "hardware hooks." If it is possible to augment a computer architecture through them (with hardware hooks or other means), the ICE abstraction and the programmer's workflow, in line with this article, can be supported. The only significant obstacle in today's multi-core architectures is their large cache-coherent local caches. Their limited scalability with respect to power gives vendors more reasons beyond an easier programming model to let go of this obstacle.

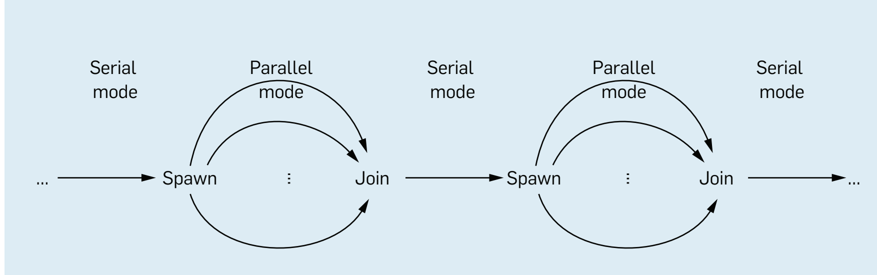
PRAM parallel algorithmic approach. The parallel random-access machine/model (PRAM) virtual model of computation is a generalization of the random-access machine (RAM) model.[9] RAM, the basic serial model behind standard programming languages, assumes any memory access or any operation (logic or arithmetic) takes unit-time (serial abstraction). The formal PRAM model assumes a certain number, say, p of processors, each able to concurrently access any location of a shared memory in the same time as a single access. PRAM has several submodels that differ by assumed outcome of concurrent access to the same memory location for either read or write purposes. Here, I note only one of them—the Arbitrary Concurrent-Read Concurrent-Write (CRCW) PRAM—which allows concurrent accesses to the same memory location for reads or writes; reads complete before writes, and an arbitrary write (to the same location, unknown in advance) succeeds. PRAM algorithms are essentially prescribed as a sequence of rounds and, for each round, up to p processors execute concurrently. The performance objective is to minimize the number of rounds. The PRAM parallel-algorithmic approach is well-known and has never been seriously challenged by any other parallel-algorithmic approach in terms of ease of thinking or wealth of knowledgebase. However, PRAM is also a strict formal model. A PRAM algorithm must therefore prescribe for each and every one of its p processors the instruction the processor executes at each time unit in a detailed computer-program-like fashion that can be quite demanding. The PRAM-algorithms theory mitigates this instruction-allocation scheme through the work-depth (WD) methodology.

This methodology (due to Shiloach and Vishkin[20]) suggests a simpler way to allocate instructions: A parallel algorithm can be prescribed as a sequence of rounds, and for each round, any number of operations can be executed concurrently, assuming unlimited hardware. The total number of operations is called "work," and the number of rounds is called "depth," as in the ICE abstraction. The first performance objective is to reduce work, and the immediate second one is to reduce depth. The methodology of restricting attention only to work and depth has been used as the main framework for the presentation of PRAM algorithms[16,17] and is in my class notes on the XMT home page http://www.umiacs.umd.edu/users/vishkin/XMT/. Deriving a full PRAM description from a WD description is easy. For concreteness, I demonstrate WD descriptions on two examples, the first concerning parallelism, the second concerning the WD methodology (see the sidebar "Parallel Algorithms").

The programmer's workflow starts with the easy-to-understand ICE abstraction and ends with the XMT system, providing a practical implementation of the vast PRAM algorithmic knowledge base.

XMT programming model. The programming model behind the XMT framework is an arbitrary concurrent read, concurrent write single program multiple data, or CRCW SPMD, programming model with two executing modes: serial and parallel. The two instructions—spawn and join—specify the beginning and end, respectively, of a parallel section (see Figure 3). An arbitrary number of virtual threads, initiated by a spawn and terminated by a join, share the same code. The workflow relies on the spawn command to extend the ICE abstraction from the WD methodology to XMT programming. As with the respective PRAM model, the arbitrary CRCW aspect dictates that concurrent writes to the same memory location result in an arbitrary write committing. No assumption needs to be made by the programmer beforehand about which one will succeed. An algorithm designed with this property in mind permits each thread to progress at its own speed, from initiating spawn to terminating join, without waiting for other threads—no thread "busy-waits" for another thread. The implied "independence of order semantics" allows XMT to have a shared memory with a relatively weak coherence model. An advantage of this easier-to-implement SPMD model is that it is PRAM-like. It also incorporates the prefix-sum statement operating on a base variable, B, and an increment variable, R. The result of a prefix-sum is that B gets the value B + R, while R gets the initial value of B, a result called "atomic" that's similar to fetch-and-increment in Gottlieb et al.[12]

Figure 3. Serial and parallel execution modes.



Merging with a Single Spawn-Join

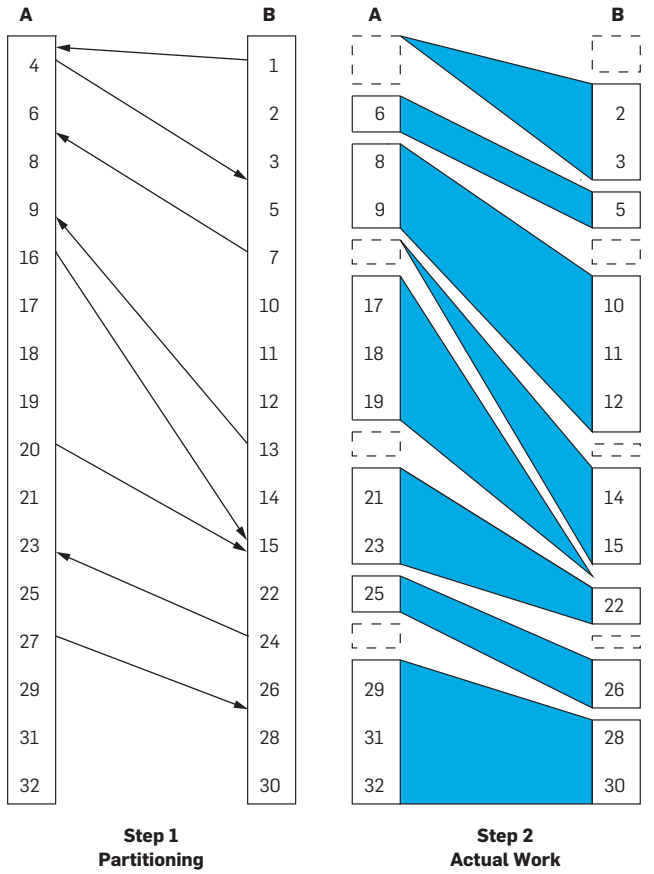
The merging problem takes as input two sorted arrays $A = A[1 \ldots n]$ and $B = B[1 \ldots n]$. Each of these $2n$ elements must then be mapped into an array $C = C[1 \ldots 2n]$ that is also sorted. I first review the Shiloach-Vishkin two-step PRAM algorithm for merging, then discuss its related XMTC programming:

Step 1. Partitioning. This step selects some number x of elements from A at equal distances. In the example in the figure here, suppose the $x = 4$ elements 4, 16, 20, and 27 are selected and ranked relative to array B using x concurrent binary searches. Similarly, x elements from B at equal distances, say, elements 1, 7, 13, and 24, are also selected, then ranked relative to array A using $x = 4$ concurrent binary searches. The step takes $O(\log n)$ time. These ranked elements partition the merging job that must be completed into $2x = 8$ "strips"; in the figure, step 2 includes eight such strips.

Step 2. Actual work. For each strip the remaining job is to merge a subarrary of A with a subarrary of B, mapping their elements into a subarray of C. Since these $2x$ merging jobs are mutually independent, each is able to concurrently apply the standard linear-time serial merging algorithm.

Consider the following complexity analysis of this algorithm: Since each strip has at most n/x elements from A and n/x elements from B, the depth (or parallel time) of the second step is $O(n/x)$. If $x \leq n/\log n$, the first step and the algorithm as a whole does $O(n)$ work. In the PRAM model, this algorithm requires $O(n/x + \log n)$ time. A simplistic XMTC program requires as many spawn (and respective join) commands as the number of PRAM steps. The reasons I include this example here are that it involves a way to use only a single spawn (and a single join) command to represent the whole merging algorithm and, as I explain in the Conclusion, to demonstrate an XMT advantage over current hardware by comparing it with the parallel merging algorithm in Cormen et al.[9]

Main steps of the ranking/merging algorithm.



Merging in XMTC. An XMTC program spawns $2x$ concurrent threads, one for each of the selected elements in array A or B. Using binary search, each thread first ranks its array element relative to the other array, then proceeds directly (without a join operation) to merge the elements in its strip, terminating just before setting the merging result of another selected element because the merging result is computed by another thread.

To demonstrate the operation of a thread, consider the thread of element 20. Starting with binary search on array B the thread finds that 20 ranks as 11 in B; 11 is the index of 15 in B. Since the index of 20 in A is 9, element 20 ranks 20 in C. The thread then compares 21 to 22 and ranks element 21 (as 21), then compares 23 to 22 to rank 22, 23 to 24 to rank 23, and 24 to 25 but terminates since the thread of 24 ranks 24, concluding the example.

Our experience is that, with little effort, XMT-type threading requires fewer synchronizations than implied by the original PRAM algorithm. The current merging example demonstrates that synchronization reduction is sometimes significant.

The primitive is especially useful when several threads perform a prefix-sum simultaneously against a common base, because multiple prefix-sum operations can be combined by the hardware to form a very fast multi-operand prefix-sum operation. Because each prefix-sum is atomic, each thread returns a different R value. This way, the parallel prefix-sum command can be used to implement efficient and scalable inter-thread synchronization by arbitrating an ordering among the threads.

The XMTC high-level language implements the programming model. XMTC is an extension of standard C, augmenting C with a small number of commands (such as spawn, join, and prefix-sum). Each parallel region is delineated by spawn and join statements, and synchronization is achieved through the prefix-sum and join commands. Every thread execut-

ing the parallel code is assigned a unique thread ID, designated $. The spawn statement takes as arguments the lowest ID and highest ID of the threads to be spawned. For the hardware implementation (discussed later), XMTC threads can be as short as eight to 10 machine instructions that are not difficult to get from PRAM algorithms. Programmers from high school to graduate school are pleasantly surprised by the flexibility of translating PRAM algorithms to XMTC multi-threaded programs. The ability to code the whole merging algorithm using a single spawn-join pair is one such surprise (see the sidebar "Merging with a Single Spawn-Join").

To demonstrate simple code, consider two code examples:

The first is a small XMTC program for the parallel exchange algorithm discussed in the "Parallel Algorithms" sidebar:

```
spawn ( 0 , n-1){
   var x
      x:=A( $ ) ;
    A( $ ):=B( $ ) ;
    B( $ ):=x
}
```

The program spawns a concurrent thread for each of the depth-3 serial-exchange iterations using a local variable x. Note that the join command is implied by the right parenthesis at the end of the program.

The second assumes an array of n integers A. The programmer wishes to "compact" the array by copying all non-zero values to another array, B, in an arbitrary order. The XMTC code is:

```
psBaseReg x=0;
spawn ( 0 , n-1){
       int e ;
       e=1;
       i f (A[ $ ] ) !=0) {
                    ps ( e , x ) ;
                    B[ e ]=A[ $ ]
}
}
```

It declares a variable x as the base value to be used in a prefix-sum command (ps in XMTC), initializing it to 0. It then spawns a thread for each of the n elements in A. A local thread variable e is initialized to 1. If the element of the thread is non-zero, the thread performs a prefix-sum to get a unique index into B where it can place its value.

Other XMTC commands. Prefix-sum-to-memory (psm) is another prefix-sum command, the base of which is any location in memory. While the increment of ps must be 0 or 1, the increment of psm is not limited, though its implementation is less efficient. Single Spawn (sspawn) is a command that can spawn an extra thread and be nested. A nested spawn command in XMTC code must be replaced (by programmer or compiler) by sspawn commands. The XMTC commands are described in the programmer's manual included in the software release on the XMT Web pages.

Tuning XMT programs for performance. My discussion here of performance tuning would be incomplete without a description of salient features of the XMT architecture and hardware. The XMT on-chip general-purpose computer architecture is aimed at the classic goal of reducing single-task completion time. The WD methodology gives algorithm designers the ability to express all the parallelism they observe. XMTC programming further permits expressing this virtual parallelism by letting programmers express as many concurrent threads as they wish. The XMT processor must now provide an effective way to map this virtual parallelism onto the hardware. The XMT architecture provides dynamic allocation of the XMTC threads onto the hardware for better load balancing. Since XMTC threads can be short, the XMT hardware must directly manage XMT threads to keep overhead low. In particular, an XMT program looks like a single thread to the operating system (see the sidebar "The XMT Processor" for an overview of XMT hardware).

The main thing performance programmers must know in order to tune the performance of their XMT programs is that a ready-to-run version of an XMT program depends on several parameters: the length of the (longest) sequence of roundtrips to memory (LSRTM); queuing delay to the same shared memory location (known as queue-read queue-write, or QRQW[11]); and work and depth. Their optimization is a responsibility shared subtly by the architecture, the compiler, and the programmer/algorithm designer.

See Vishkin et al[27] for a demonstration of tuning XMTC code for performance by accounting for LSRTM. As an example, it improves XMT hardware performance on the problem of summing n numbers.

Execution can differ from the literal XMTC code in order to keep the size of working space under control or otherwise improve performance. For example, compiler and runtime methods could perform this modification by clustering virtual threads offline or online and prioritize execution of nested spawns using known heuristics based on a mix of depth-first and breadth-first searches.

Commitments to silicon of XMT by my research team at the University of Maryland include a 64-processor, 75MHz computer based on field-programmable gate array (FPGA) technology developed by Wen[28] and 64-processor ASIC 10mm X 10mm chip using IBM's 90nm technology developed together by Balkan, Horak, Keceli, and Wen (see Figure 4). Tzannes and Cargaea (guided by Barua and me) have also developed a basic yet stable compiler, and Keceli has developed a cycle-accurate simulator of XMT. Both are available through the XMT software release on the XMT Web pages.

Easy to build. An individual graduate student with no prior design experience completed the XMT hardware description (in Verilog) in just over two years (2005–2007). XMT is also silicon-efficient. The ASIC design by the XMT research team at the University of Maryland shows that a 64-processor XMT needs the same silicon area as a (single) current commodity core. The XMT approach goes after any type of application parallelism regardless of how much parallelism the application requires, the regularity of this parallelism, or the parallelism's grain size, and is amenable to standard multiprogramming where the hardware supports several concurrent operating-system threads.

The XMT team has demonstrated good XMT performance, independent software engineers have demonstrated XMT programmability (see Hochstein et al.[14]), and independent education professionals have demonstrated

Eye-of-a-Needle Aphorism

Introduced at a time of hardware scarcity almost 70 years ago, the von Neumann apparatus of stored program and program counter forced the threading of instructions through a metaphoric eye of a needle. Coupling of mathematical induction and (serial) ISE abstraction was engineered to provide this threading, as discussed throughout the article. See especially the description of how variable X is used in the pseudo-code of the serial iterative algorithm in the exchange problem; also the first-in-first-out queue data structure in the serial breadth-first search; and the serial merging algorithm in which two elements are compared at a time, one from each of two sorted input arrays. As eye-of-a-needle threading is already second nature for many programmers, it has come to be associated with ease of programming.

Threading through the eye of a needle is an aphorism for extreme difficulty, even impossibility, in the broader culture, including in the texts of three major religions. The XMT extension to the von Neumann apparatus (noted in "The XMT Processor" sidebar) exploits today's relative abundance of hardware resources to free computing from the constraint of threading through the original apparatus. Coupling mathematical induction and the ICE abstraction explored here is engineered to capitalize on this freedom for ease of parallel programming and improved machine and application performance.

XMT teachability (see Torbert et al.[23]). Highlights include evidence of 100X speedups on general-purpose applications on a simulator of 1,000 on-chip processors[13] and speedups ranging from 15X to 22X for irregular problems (such as Quicksort, breadth-first search on graphs, finding the longest path in a directed acyclic graph), and speedups of 35X–45X for regular programs (such as matrix multiplication and convolution) on the 64-processor XMT prototype versus the best serial code on XMT.[28]

In 2009, Caragea et al.[8] demonstrated nearly 10X average performance improvement potential relative to Intel Core 2 Duo for a 64-processor XMT chip using the same silicon area as a single core. In 2010, Caragea et al.[7] demonstrated that, using the same silicon area as a modern graphics processing unit (GPU), the XMT design achieves an average speedup of 6X relative to the GPU for irregular applications and falls only slightly behind on regular ones. All GPU code was written and optimized by researchers and programmers unrelated to the XMT project.

With few exceptions, parallel programming approaches that dominated parallel computing prior to many-cores are still favored by vendors, as well as high-performance users. The steps they require include decomposition, assignments, orchestration, and mapping.[10] Indeed, parallel programming difficulties have failed all general-purpose parallel systems to date by limiting their use. In contrast, XMT frees its programmers from doing all the steps, in line with the ICE/PRAM abstraction.

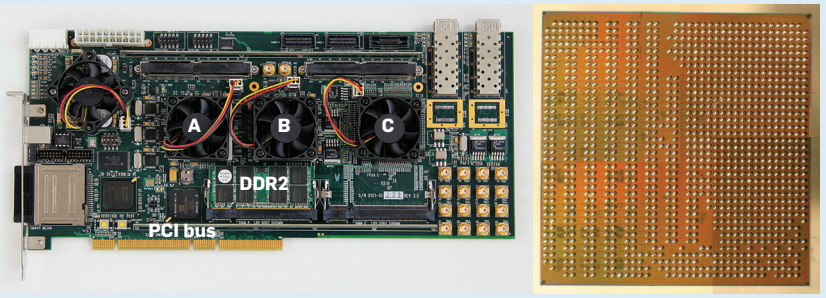
The XMT software environment release (a 2010 release of the XMTC compiler and cycle-accurate simulator of XMT) is available by free download from the XMT home page and from sourceforge.net, along with extensive documentation, and can be downloaded to any standard desktop computing platform. Teaching materials covering a class-tested programming methodology in which students are taught only parallel algorithms are also available from the XMT Web pages.

Most CS programs today graduate students to a job market certain to be dominated by parallelism but without the preparation they need. The level of awareness of parallelism required by the ICE/PRAM abstraction is so basic it is necessary for all other current approaches. As XMT is also buildable, the XMT approach is sufficient for programming a real machine. I therefore propose basing the introduction of the new generation of CS students to parallelism on the workflow presented here, at least until CS generally converges on a many-core platform.

Related efforts. Related efforts toward parallelism come in several flavors; for example, Valiant's Multi-BSP bridging model for multi-core computing[24] appears closest to the XMT focus on abstraction. The main difference, however, is that XMT aims to preempt known shortcomings in existing machines by showing how to build machines differently, while the modeling in Valiant[24] aims to improve understanding of existing machines.

These prescriptive versus descriptive objectives are not the only difference. Valiant[24] modeled relatively low-level parameters of certain multi-core architectures, making them closer to Vishkin et al.[27] than to this article. Unlike both sources, simplicity drives the "one-liner" ICE abstraction. Parallel languages (such as CUDA, MPI, and OpenMP) tend to be different from computational models, as they often do not involve performance modeling. They require a level of detail that distances them farther from simple

Figure 4. Left side. FPGA board (size of a car license plate) with three FPGA chips (generously donated by Xilinx): A, B: Virtex-4LX200; C: Virtex-4FX100. Right side. 10mm X 10mm chip using IBM Flip-Chip technology.



A, B: Virtex-4LX200.
C: Virtex-4FX100.

10mm X 10mm chip using IBM Flip-Chip technology.

The XMT Processor

The XMT processor (see the figure here) includes a master thread control unit (MTCU), processing clusters, each comprising several thread-control units (TCUs), a high-bandwidth low-latency interconnection network[3] and its extension to a globally asynchronous locally synchronous, GALS-style, design incorporating asynchronous logic,[15,18] memory modules (MM), each comprising on-chip cache and off-chip memory, prefix-sum (PS) unit(s), and global registers. The shared-memory-modules block (bottom left of the figure) suppresses the sharing of a memory controller by several MMs. The processor alternates between serial mode (in which only the MTCU is active) and parallel mode. The MTCU has a standard private data cache (used in serial mode) and a standard instruction cache. The TCUs, which lack a write data cache, share the MMs with the MTCU.

The overall XMT design is guided by a general design ideal I call no-busy-wait finite-state-machines, or NBW FSM, meaning the FSMs, including processors, memories, functional units, and interconnection networks comprising the parallel machine, never cause one another to busy-wait. It is ideal because no parallel machine can operate that way. Nontrivial parallel processing demands the exchange of results among FSMs. The NBW FSM ideal represents my aspiration to minimize busy-waits among the various FSMs comprising a machine.

Here, I cite the example of how the MTCU orchestrates the TCUs to demonstrate the NBW FSM ideal. The MTCU is an advanced serial microprocessor that also executes XMT instructions (such as spawn and join). Typical program execution flow, as in Figure 3, can also be extended through nesting of sspawn commands. The MTCU uses the following XMT extension to the standard von Neumann apparatus of the program counters and stored program: Upon encountering a spawn command, the MTCU broadcasts the instructions in the parallel section starting with that spawn command and ending with a join command on a bus connecting to all TCU clusters.

The largest ID number of a thread the current spawn command must execute Y is also broadcast to all TCUs. The ID (index) of the largest executing threads is stored in a global register X. In parallel mode, a TCU executes one thread at a time. Executing a thread to completion (upon reaching a join command), the TCU does a prefix-sum using the PS unit to increment global register X. In response, the TCU gets the ID of the thread it could execute next; if the ID is ≤Y, the TCU executes a thread with this ID. Otherwise, the TCU reports to the MTCU that it finished executing. When all TCUs report they've finished, the MTCU continues in serial mode.

The broadcast operation is essential to the XMT ability to start all TCUs at once in the same time it takes to start one TCU. The PS unit allows allocation of new threads to the TCUs that just became available within the same time it takes to allocate one thread to one TCU. This dynamic allocation provides runtime load-balancing of threads coming from an XMTC program.

We are now ready to connect with the NBW FSM ideal. Consider an XMT program derived from the workflow. From the moment the MTCU starts executing a spawn command until each TCU terminates the threads allocated to it, no TCU can cause any other TCU to busy-wait for it. An unavoidable busy-wait ultimately occurs when a TCU terminates and begins waiting for the next spawn command.

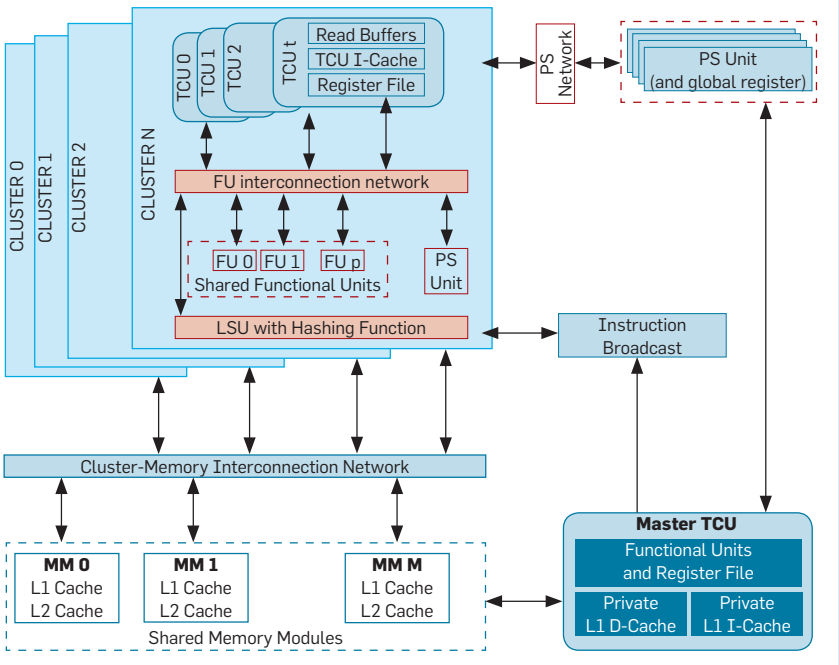
TCUs, with their own local registers, are simple in-order pipelines, including fetch, decode, execute/memory-access, and write-back stages. The FPGA computer has 64 TCUs in four clusters of 16 TCUs each. XMT designers and evangelists aspire to develop a machine with 1,024 TCUs in 64 clusters. A cluster includes functional units shared by several TCUs and one load/store port to the interconnection network shared by all its TCUs.

The global memory address space is evenly partitioned into the MMs through a form of hashing. The XMT design eliminates the cache-coherence problem, a challenge in terms of bandwidth and scalability. In principle, there are no local caches at the TCUs. Within each MM, the order of operations to the same memory location is preserved.

For performance enhancements (such as data prefetch) incorporated into the XMT hardware, along with more on the architecture, see Wen and Vishkin[28]; for more on compiler and runtime scheduling methods for nested parallelism, see Tzannes et al.,[23] and for prefetching methods, see Caragea et al.[6]

Patents supporting the XMT hardware were granted from 2002 to 2010, appearing in Nuzman and Vishkin[18] and Vishkin.[25]

Block diagram of the XMT architecture.



abstractions.

Several research centers[1,2] are actively exploring the general problems discussed here. The University of California, Berkeley, Parallel Computing Lab and Stanford University's Pervasive Parallelism Laboratory advocate an application-driven approach to reinventing computing for parallelism.

Conclusion

The vertical integration of a parallel-processing system, compiler, programming, and algorithms proposed here through the XMT framework with the ICE/PRAM abstraction as its front-end is notable for its relative simplicity. ICE is a newly defined feature that has not appeared in prior research, including my own, and is more rudimentary than prior parallel computing concepts. Rudimentary concepts are the basis for the fundamental development of any field. ICE can be viewed as an axiom that builds on mathematical induction, one of the more rudimentary concepts in mathematics. The suggestion here of using a simple abstraction as the guiding principle for reinventing computing for parallelism also appears to be new. Considerable evidence suggests it can be done (see the sidebar "Eye-of-a-Needle Aphorism").

The following comparison with a chapter on multithreading algorithms in the 2009 textbook *Introduction to Algorithms* by Cormen et al.[9] helps clarify some of the article's contributions. The 1990 first edition of Cormen et al.[9] included a chapter on PRAM algorithms emphasizing the role of work-depth design and analysis; the 2009 chapter[9] likewise emphasized work-depth analysis. However, to match current commercial hardware, the 2009 chapter turned to a variant of dynamic multithreading (in lieu of work-depth design) in which the main primitive was similar to the XMT sspawn command (discussed here). One thread was able to generate only one more thread at a time; these two threads would then generate one more thread each, and so on, instead of freeing the programmer to directly design for the work-depth analysis that follows (per the same 2009 chapter).

Cormen et al.'s[9] dynamic multithreading should encourage hardware

The XMT on-chip general-purpose computer architecture is aimed at the classic goal of reducing single-task completion time.

enhancement to allow simultaneously starting many threads in the same time required to start one thread. A step ahead of available hardware, XMT includes a spawn command that spawns any number of threads upon transition to parallel mode. Moreover, the ICE abstraction incorporates work-depth early in the design workflow, similar to Cormen et al.'s 1990 first edition.[9]

The $O(\log n)$ depth parallel merging algorithm versus the $O(\log^2 n)$ depth one in Cormen et al.[9] demonstrated an XMT advantage over current hardware, as XMT allows a parallel algorithm for the same problem that is both faster and simpler. The XMT hardware scheduling brought the hardware performance model much closer to work-depth and allowed the XMT workflow to streamline the design with the analysis from the start.

Several features of the serial paradigm made it a success, including a simple abstraction at the heart of the "contract" between programmers and builders, the software spiral, ease of programming, ease of teaching, and backward compatibility on serial code and application programming. The only feature that XMT, as in other multi-core approaches, does not provide is speedups for serial code. The ICE/PRAM/XMT workflow and architecture provide a viable option for the many-core era. My XMT solution should challenge and inspire others to come up with competing abstraction proposals or alternative architectures for ICE/PRAM. Consensus around an abstraction will move CS closer to convergence toward a many-core platform and putting the software spiral back on track.

The XMT workflow also gives programmers a productivity advantage. For example, I have traced several errors in student-developed XMTC programs to shortcuts the students took around the ICE algorithms. Overall, improved understanding of programmer productivity, a traditionally difficult issue in parallel computing, must be a top priority for architecture research. To the extent possible, evaluation of productivity should be on par with performance and power. For starters, productivity benchmarks must be developed.

Ease of programming, or programmability, is a necessary condition for the success of any many-core platform, and teachability is a necessary condition for programmability and in turn for productivity. The teachability of the XMT approach has been demonstrated extensively; for example, since 2007 more than 100 students in grades K–12 have learned to program XMT, including in two magnet programs: Montgomery Blair High School, Silver Spring, MD, and Thomas Jefferson High School for Science and Technology, Alexandria, VA.[22] Others are Baltimore Polytechnic High School, where 70% of the students are African American, and a summer workshop for middle-school students from underrepresented groups in Montgomery County, MD, public schools.

In the fall of 2010, I jointly conducted another experiment, this one via video teleconferencing with Professor David Padua of the University of Illinois, Urbana-Champaign using Open MP and XMTC, with XMTC programming assignments run on the XMT 64-processor FPGA machine. Our hope was to produce a meaningful comparison of programming development time from the 30 participating Illinois students. The topics and problems covered in the PRAM/XMT part of the course were significantly more advanced than Open MP alone. Having sought to demonstrate the importance of teachability from middle school on up, I strongly recommend that it becomes a standard benchmark for evaluating many-core hardware platforms.

Blake et al.[4] reported that after analyzing current desktop/laptop applications for which the goal was better performance, the applications tend to comprise many threads, though few of them are used concurrently; consequently, the applications fail to translate the increasing thread-level parallelism in hardware to performance gains. This problem is not surprising given that most programmers can't handle multi-core microprocessors. In contrast, guided by the simple ICE abstraction and by the rich PRAM knowledgebase to find parallelism, XMT programmers are able to represent that parallelism using a type of threading the XMT hardware is engineered to exploit for performance.

For more, see the XMT home page at the University of Maryland http://www.umiacs.umd.edu/users/vishkin/XMT/. The XMT software environment release is available by free download there and from sourceforge.net at http://sourceforge.net/projects/xmtc/, along with extensive documentation. A 2010 release of the XMTC compiler and cycle-accurate simulator of XMT can also be downloaded to any standard desktop computing platform. Teaching materials covering a University of Maryland class-tested programming methodology in which even college freshmen and high school students are taught only parallel algorithms are also available from the XMT Web pages.

Acknowledgment

This work is supported by the National Science Foundation under grant 0325393.

References

1. Adve, S. et al. *Parallel Computing Research at Illinois: The UPCRC Agenda.* White Paper. University of Illinois, Champaign-Urbana, IL,2008; http://www.upcrc.illinois.edu/UPCRC_Whitepaper.pdf
2. Asanovic, K. et al. *The Landscape of Parallel Computing Research: A View from Berkeley.* Technical Report UCB/EECS-2006-183. University of California, Berkeley, 2006; http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
3. Balkan, A., Horak, M., Qu, G., and Vishkin, U. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Proceedings of the 15th Annual IEEE Symposium on High Performance Interconnects* (Stanford, CA, Aug. 22–24). IEEE Press, Los Alamitos, CA, 2007.
4. Blake, G., Dreslinski, R., Flautner, K., and Mudge, T. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France, June 19–23). ACM Press, New York, 2010, 302–313.
5. Borkar, S. et al. *Platform 2015: Intel Processor and Platform Evolution for the Next Decade. White Paper.* Intel, Santa Clara, CA, 2005; http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_borkar_2015.pdf
6. Caragea, G., Tzannes, A., Keceli, F., Barua, R., and Vishkin, U. Resource-aware compiler prefetching for many-cores. In *Proceedings of the Ninth International Symposium on Parallel and Distributed Computing* (Istanbul, Turkey, July 7–9). IEEE Press, Los Alamitos, CA, 2010, 133–140.
7. Caragea, G., Keceli, F., Tzannes, A., and Vishkin, U. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *Proceedings of the Second Usenix Workshop on Hot Topics in Parallelism* (University of California, Berkeley, June 14–15). Usenix, Berkeley, CA, 2010.
8. Caragea, G., Saybasili, B., Wen, X., and Vishkin, U. Performance potential of an easy-to-program PRAM-on-chip prototype versus state-of-the-art processor. In *Proceedings of the 21st ACM SPAA Symposium on Parallelism in Algorithms and Architectures* (Calgary, Canada, Aug. 11–13). ACM Press, New York, 2009, 163–165.
9. Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, Third Edition.* MIT Press, Cambridge, MA, 2009.
10. Culler, D. and Singh, J. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan-Kaufmann, San Francisco, CA, 1999.
11. Gibbons, P., Matias, Y., and Ramachandran, V. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science 196*, 1–2 (Apr. 1998), 3–29.
12. Gottlieb, A. et al. The NYU ultracomputer designing an MIMD shared-memory parallel computer. *IEEE Transactions on Computers 32*, 2 (Feb. 1983), 175–189.
13. Gu, P. and Vishkin, U. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor. *Journal of Embedded Computing 2*, 2 (Apr. 2006), 181–190.
14. Hochstein, L., Basili, V., Vishkin, U., and Gilbert, J. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software 81*, 11 (Nov. 2008), 1920–1930.
15. Horak, M., Nowick, S., Carlberg, M., and Vishkin, U. A low-overhead asynchronous interconnection network for gals chip multiprocessor. In *Proceedings of the Fourth ACM/IEEE International Symposium on Networks-on-Chip* (Grenoble, France, May 3–6). IEEE Computer Society, Washington D.C., 2010, 43–50.
16. JaJa, J. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Reading, MA, 1992.
17. Keller, J., Kessler, C., and Traeff, J. *Practical PRAM Programming.* Wiley-Interscience, New York, 2001.
18. Nuzman, J. and Vishkin, U. *Circuit Architecture for Reduced-Synchrony-On-Chip Interconnect.* U.S. Patent 6,768,336, 2004; http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&p=1&u=%2Fnetahtml%2FPTO%2Fsearch-bool.html&r=1&f=G&l=50&co1=AND&d=PTXT&s1=6768336.PN.&OS=PN/6768336&RS=PN/6768336
19. Patterson, D. The trouble with multi-core: Chipmakers are busy designing microprocessors that most programmers can't handle. *IEEE Spectrum* (July 2010).
20. Shiloach, Y. and Vishkin, U. An $O(n^2 \log n)$ parallel max-flow algorithm. *Journal of Algorithms 3*, 2 (Feb.1982), 128–146.
21. Sutter, H. The free lunch is over: A fundamental shift towards concurrency in software. *Dr. Dobb's Journal 30*, 3 (Mar. 2005).
22. Torbert, S., Vishkin, U., Tzur, R., and Ellison, D. Is teaching parallel algorithmic thinking to high school students possible? One teacher's experience. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Milwaukee, WI, Mar. 10–13). ACM Press, New York, 2010, 290–294.
23. Tzannes, A., Caragea, G., Barua, R., and Vishkin, U. Lazy binary splitting: A run-time adaptive dynamic works-stealing scheduler. In *Proceedings of the15th ACM Symposium on Principles and Practice of Parallel Programming* (Bangalore, India, Jan. 9–14). ACM Press, New York, 2010, 179–189.
24. Valiant, L. A bridging model for multi-core computing. In *Proceedings of the European Symposium on Algorithms* (Karlruhe, Germany, Sept. 15–17). Lecture Notes in Computer Science 5193. Springer, Berlin, 2008, 13–28.
25. Vishkin, U. U.S. Patents 6,463,527; 6,542,918; 7,505,822; 7,523,293; 7,707,388, 2002–2010; http://patft.uspto.gov/
26. Vishkin, U. Algorithmic approach to designing an easy-to-program system: Can it lead to a hardware-enhanced programmer's workflow add-on? In *Proceedings of the 27th International Conference on Computer Design* (Lake Tahoe, CA, Oct. 4–7). IEEE Computer Society, Washington D.C., 2009, 60–63.
27. Vishkin, U., Caragea, G., and Lee, B. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-on-chip platform. In *Handbook on Parallel Computing*, S. Rajasekaran and J. Reif, Eds. Chapman and Hall/CRC Press, Boca Raton, FL, 2008, 5.1-60.
28. Wen, X. and Vishkin, U. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the Fifth ACM Conference on Computing Frontiers* (Ischia, Italy, May 5–7). ACM Press, New York, 2008, 55–66.

Uzi Vishkin (vishkin@umd.edu) is a professor in the University of Maryland Institute for Advanced Computer Studies (http://www.umiacs.umd.edu/~vishkin) and Electrical and Computer Engineering Department, College Park, MD.

Fonte: Communications of the ACM, v. 54, n. 1, p. 75-85, 2011. [Base de Dados]. Disponível em: <http://web.ebscohost.com>. Acesso em: 11 fev. 2011.