Using Simple Abstraction to Guide the Reinvention of Computing for Parallelism

Uzi Vishkin
The University of Maryland Institute for Advanced Computer Studies (UMIACS) and Electrical and Computer Engineering Department
vishkin@umd.edu

ABSTRACT

The sudden shift from single-processor computer systems to many-processor parallel computing systems requires reinventing much of Computer Science (CS): how to actually build and program the new parallel systems. CS urgently requires **convergence** *to a robust parallel general-purpose platform* that provides good performance and is easy to program. Unfortunately, this same objective has eluded decades of parallel computing research. Now, continued delays and uncertainty could start affecting important sectors of the economy. This paper advocates a minimalist stepping-stone: settle first on a simple *abstraction* that encapsulates the new interface between programmers, on one hand, and system builders, on the other hand.

This paper also makes several concrete suggestions: (i) the Immediate Concurrent Execution (ICE) abstraction as a candidate for the new abstraction, and (ii) the Explicit Multi-Threaded (XMT) general-purpose parallel platform, under development at the University of Maryland, as a possible embodiment of ICE. ICE and XMT build on a formidable body of knowledge, known as PRAM (for parallel random-access machine, or model) algorithmics, and a latent, though not widespread, familiarity with it. Ease-of-programming, strong speedups and other attractive properties of the approach suggest that we may be much better prepared for the challenges ahead than many realize.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel architectures; C.5.3 [**Computer System Implementation**]: Microcomputers; D.1.3 [**Programming Techniques**]: Parallel programming; F.1.2 [**Computation by Abstract Devices**]: Parallelism; I.1.2 [**Computing Methodologies**]: Algorithms

General Terms

Algorithms, Design, Performance, Economics, Human Factors, Languages, Theory

Keywords

Parallel computing, Parallel algorithms, Abstraction

.

1. INTRODUCTION

Until 2004, standard (desktop) computers comprised a single processor core. Since 2005 we appear to be on track with a prediction [7] of 8-core computers in 2008 and 64-core ones by 2012. Transition from serial (single core) computing to parallel (many-core) computing mandates the reinvention of the very heart of computer science (CS) as these highly parallel computers need to be built and programmed differently from the single-core machines that dominated standard computer systems since the inception of the field. The first computers in the 1940s were controlled by a clock that ticked at a rate of about 5,000 times per second (5KHz). In 2003, the clock of a high end desktop processor ticked at a rate approaching 4,000,000,000 times per second (4GHz). This nearly million-fold improvement in clock rate over less than six decades occurred through a more or less regular doubling of the clock rate every three years. At that rate we should have had processors approaching 16GHz in 2009, but clock rates of processors have hardly improved since 2003. The bad news that prompted this change is that the industry did not find a way to continue improving clock rates within acceptable power budgets [7]. Computers are built using digital logic. Fortunately, due to miniaturization and other hardware technology improvements, the amount of logic that a computer chip can contain continues to grow, doubling every 18 to 24 months. This means that CS has entered a new era in which performance growth will have to rely on increased parallelism in processing. Computers with an increasing number of cores are expected without significant improvements in clock rates. *Exploiting these cores in parallel for faster completion of a computing task* will be the only way to improve performance of single tasks from one generation of computers to the next, since the number of cores will be the main difference between successive generations.

Unfortunately, commercial multi-core computers struggle to harness even the few cores they currently have for faster completion of a computational task. A perception of near despair in the community was reflected in the last posting of a special series on the problems posed by many-core processors on the Computing Community Consortium blog [17]. The problem is not new. Many parallel computer architectures have been proposed and built over the last 40 years, but with limited success. The new frontier of exploiting their parallelism is exactly what has often eluded their users. Observing that nowadays language researchers are locked into mechanisms supported by commodity hardware and hardware researchers are locked into fully supporting any current software, [17] calls on all involved communities to *collaboratively* start with a *clean slate*.

A clean slate can better accommodate a quest to reproduce salient tenets of the now-derailed serial paradigm for the many-core era. The presentation is guided by three tenets: (i) the high-level (macro) view in the form of the "software spiral" that sets a context for restating the main challenge, (ii) the single nail that holds everything together (micro view) in the form of a very simple and concise abstraction, and (iii) the engine of algorithmic thinking and knowhow; serial algorithms will have to be generalized to parallel algorithms. The paper concludes with an example of a comprehensive platform. Its computer programming, computer system specifications, architecture and implementation and their future evolution address these tenets.

1.1 The Software Spiral

The term software spiral reflects the cyclic process of hardware improvements leading to software improvements, which lead back to hardware improvements and so on. What facilitated the software spiral is a stable application-software base that could be reused effectively and enhanced from one hardware generation to the next. Better performance was assured with each generation if only the hardware could be improved. The *scalability parameter* for the serial software spiral had been the time for executing (any) serial code, where each hardware generation executed serial code faster. Andy Grove (Intel) noted that the software spiral had been an engine of sustained growth for Information Technology for many decades. It offered unique advantages. The software spiral provided a joint platform bringing into the fold many players, some with conflicting interests, agendas and backgrounds. They included users and builders of general-purpose computers, novices, casual contributors, and veteran (applications software, system software and hardware) developers alike. Due to the software spiral they all contributed to the same scalability parameter.

Alas, *the software spiral is now broken* (e.g., as implied by [18]): (i) Nobody builds hardware that provides improved performance on the old serial software base. (ii) There is no broad parallel computing application software base for which hardware vendors are committed to improve performance. (iii) No agreed-upon architecture currently allows application programmers to build such software base for the future.

Foremost among current challenges is **the many-core convergence challenge:** *Seek timely convergence to a robust many-core platform coupled with a new many-core software spiral* that will serve the world of computing for many years to come.

1.2 Case for Abstraction

[TEXT BOX BEGINS] One of the dictionary definitions of *abstract* is *difficult to understand*, or *abstruse*. In CS, however, abstraction has become synonymous with the quest for simplicity. Interestingly, the word *abstraction* in Hebrew shares the same root with *simple* (as well as *undress* and *expansion*). [TEXT BOX ENDS]

Many-core convergence is, of course, not the first time that CS is facing a complex system problem requiring a solution that involves many different players and should be robust withstanding future system upgrades. Addressing such problems by figuring out a simple abstraction that acts as a single nail holding everything together would be a characteristic CS intellectual success story. Abstractions that present the user with a *virtual machine* that is easier to understand and program than the underlying hardware, while allowing for the effective use of the hardware, have indeed facilitated significant CS accomplishments. Broad consensus built around these simple abstractions was crucial. Some formative abstractions were: (i) *that any single instruction available for execution in a serial program executes immediately*, henceforth called *immediate serial execution (ISE).* Since an instruction may apply to any location in memory, ISE actually extended another formative abstraction that we call *immediate memory access (IMA): that any particular word of an indefinitely large memory is immediately available.* And (ii) *that a computer is serving the task that the user is currently working on* **exclusively**, henceforth *exclusive computer availability (ECA).* The IMA abstraction abstracts away a hierarchy of memories, each with greater capacity, but slower access time, than the preceding one, and the ISE abstraction extends it to immediate execution of any operation. The left side of Figure 1 depicts the execution of a serial algorithm as implied by the ISE abstraction, where unit time instructions execute one at a time. The ECA abstraction abstracts away virtual file systems that can be implemented in local storage or a local or global network, access to the Internet, and other tasks that may be concurrently using the same computer system resources. These abstractions have improved the productivity of programmers and other users, and contributed towards broadening participation in computing.

[TEXT BOX BEGINS] `Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible, [8]`. Representing one of the most formative efforts in the history of the field, this quote reflects tension between a desired abstraction and physical realization. Six decades later, the verdict on how this tension was resolved is clear. As imperfect as this abstraction is, mainstream CS holds that the abstraction won. A prevailing working assumption for nearly every computer scientist is the IMA abstraction (as well as the more general ISE abstraction). In fact, many computer system and compiler professionals have labored to mitigate the gaps between the memory hierarchy and the IMA abstraction allowing programmers to incorporate the abstraction in their programming model, improving their productivity. The only exception to all those who conformed with the IMA abstraction is the relatively few who seek to get the most out of the memory hierarchy for their application, by avoiding the IMA abstraction. [TEXT BOX ENDS]

For the first writing on the clean slate sought in [17], we suggest a simple and robust abstraction. Since we will then need to introduce a software spiral for parallel computing, we first reflect on the issues and then present our abstraction desiderata. (1) *Educating users to co-lead the discussion, and embolden them to lead it if a platform is buildable.* Achieving convergence to a many-core platform requires active collaboration from numerous programmers, users and educators, and is quite a challenge: (a) the world is yet to see a parallel platform that merits such collaboration. A National Science Foundation panel [6] reports: *to many users, programming existing parallel computers is as intimidating and time-consuming as programming in assembly language*; (b) even when a systems vendor introduces a new system

to the market, application-software vendors will not rush to invest in the first generation of a platform betting that there will be future generations; (c) off to a bad start, the dynamics of parallel computing is not encouraging. J. Hennessy, a leading computer architect, said in a recent interview: *Many of the early ideas were motivated by observations of what was easy to implement in the hardware rather than what was easy to use.* These early systems led to the parallel programming languages that the [6] report is complaining about. Still they became the basis for language standards. Consequently, application benchmarks that emerged from a wrong-headed architecture direction are guiding systems researchers and developers. Hennessy's advice on pursuing what is easy to use is still second priority, at best; (d) if programming will come to be viewed as hacking around the way a machine was engineered, US presence in CS will greatly weaken, as this would repel many capable prospective students. We argue that there is a better way. If a platform is buildable, **users are best qualified to determine prospects for adoption by users**. This tautology-like recognition is still a radical departure from current practice. Too timid to offer opinion, users often act as followers. Emboldening users to play a more assertive role in seeking convergence to a many-core platform is one of our goals. Our quest for a simple abstraction provides an effective vehicle for that. The more agreeable a system is to programmers the more likely it is to remedy past parallel computing ills. System researchers and developers will be the first to benefit as approval by users renders merit to their work. (2) The *future parallel scalability parameter* will have to account for the following considerations. User of a 16-core or a 1024-core computer must be able to use the same program, or else performance code will have to be continuously rewritten. We should expect that different parallel computer programs will expose different amounts of parallelism and some will remain serial, and be aware that the amount of parallelism a program can exploit tends to grow with the input size for the problem. (3) *One-size-fits-all is the heart of general-purpose computing.* The adaptability of the serial general-purpose platform to numerous applications that could not have been predicted when it was conceived benefited from the simplicity of the ISE abstraction, and has been one of its most attractive features. (4) *Avoid encouraging the wrong platforms.* The more difficult-to-program a many-core system the more resourceful its programmers has to be; since even simple problems lead to publishable results, academic communities may grow around systems that have no future. Diverting the focus to a simple abstraction would sidestep this unintended pitfall. (5) *Quick path to resumption of healthy competition.* Timely agreement on a many-core abstraction will allow bringing again into the fold many players with possibly different interests and agendas to advance the same scalability parameter. Resuming healthy competition among various players has worked well in the past, and should be contrasted with ideas for much more elaborate collaborations (sometimes on open-ended research questions) among competitors with no record of collaboration or open-ended research.

The desired abstraction will: (i) be simple, hiding the details of the underlying hardware, (ii) be accessible to the broadest possible groups of users, (iii) allow strong speedups for applications, (iv) incorporate a *scalability parameter* that suggests the following: each many-core generation should provide better performance on programs whose parallelism was not fully exploited by the previous generation, and not fall behind on others, including serial programs; (v) extend, rather than replace, existing (successful) abstractions; in particular, when code provides no parallelism, the user will need to be able to fall back on the serial abstraction ISE; and last, but definitely not least, (vi) be buildable; we must be able to build an actual computer system that provides good performance for users that rely on the abstraction. The ECA abstraction does not require change.

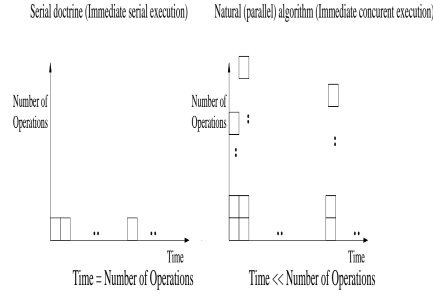

Figure 1: A serial algorithm based on the ISE abstraction versus a parallel algorithm based on the ICE abstraction.

2. THE ICE ABSTRACTION CANDIDATE

The candidate abstraction we propose is: *that indefinitely many instructions, which are available for concurrent execution, execute immediately*; we dub it *immediate concurrent execution (ICE).* A step-by-step explication of the instructions that are available next for concurrent execution is independent of the number of processors, and falls back on ISE, in case of one instruction per step. The right side of Figure 1 depicts the execution of a parallel algorithm as implied by the ICE abstraction, where at each time unit any number of unit time instructions that can execute concurrently do, followed by yet another time unit in which the same happens and so on. The discussion below first compares this abstract view of parallel algorithms to a similar view of serial algorithms. Later, it compares it to other parallel approaches. (i) Any (serial) algorithm on the left hand side is a special case (with concurrency of one at every step) of the right hand side. However, to make the relationship between a serial algorithm and the right side more interesting, consider the possibility where the serial algorithm has several other instructions that could execute concurrently with its first instruction. Assuming that all these instructions do execute concurrently, there will be now several other instructions that can execute next, and so on. Arguably, it would be more natural to have all instructions that can execute next do that, rather than artificially serialize them in some arbitrary order as the serial doctrine has taught us to do. Conversely, executing one after the other the operations that execute concurrently in a parallel algorithm on the right side of Figure 1 provides a serial algorithm for its left side. In the serial algorithm the number of time units (also called "depth") is the same as the total number of operations (or "work") of the algorithm, while in the parallel algorithm it is lower. Ideally, the work of a parallel algorithm will not much exceed that of its serial counterpart for the same problem, and its depth will be much lower than its work. (ii) *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]. (See also Figure 6 in Section 5.2.) The embodiment below reinforces simplicity and ease-of-programming, and

addresses speedups and implementation.

3. PRAM/XMT EMBODIMENT

In the remainder of this paper, we overview progress we have already made towards an *XMT/PRAM* embodiment of the ICE abstraction. We address all six desiderata above. Our presentation draws parallels to three *layers* in the vertical integration of the embodiment of the serial ISE abstraction. (i) The algorithms and data structures layer, as reflected, for example, in standard CS curriculum; (ii) the programming layer; and (iii) the actual hardware architecture and compiler of the computer system layer. Each section below refers to one layer.

3.1 The 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. RAM, e.g. [9], is the basic serial model underlying standard programming languages assuming that any memory access or any (logic, or arithmetic) operation takes unit time (the ISE abstraction). The formal PRAM model assumes a certain number, say p, of processors, each can concurrently access any location of a shared memory within the same time as a single access. The PRAM has several sub-models differing by the assumed outcome of concurrent access to the same memory location for either read or write purposes. For brevity, we note here only one of these sub-models, the Arbitrary Concurrent-Read Concurrent-Write (CRCW) PRAM: concurrent accesses to the same memory location for reads or writes are allowed; reads complete before writes and an arbitrary write (to the same location) unknown in advance succeeds. PRAM algorithms are essentially prescribed as (a) a sequence of rounds, and (b) for each round, up to p processors can execute concurrently. The performance objective is minimizing the number of rounds. The PRAM parallel algorithmic approach is well-known and has never been seriously challenged by any other parallel algorithmic approach on ease of thinking, or wealth of knowledge-base. However, the PRAM model is a strict formal model. A PRAM algorithm must prescribe for each and every one of its p processors the instruction that the processor executes at each time unit in a detailed computer-program-like fashion, which can be quite demanding. The PRAM algorithms theory mitigates this using the work-depth (WD) methodology.

The WD methodology (due to [15]) suggests a simpler way: a parallel algorithm can be prescribed as (a) a sequence of rounds, and (b) 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 with the ICE abstraction). The first performance objective is reducing work. The immediate-second priority is reducing depth. The WD methodology allows significant flexibility for describing these concurrent operations, including, for example, implicit descriptions. Shiloach and Vishkin [15] conjectured in the early 1980s that deriving a full PRAM description from WD description would be possible and later become a matter of teachable skill. The first part of the conjecture was first validated by PRAM algorithms research in the 1980s. The methodology of restricting attention only to work and depth has, in fact, been used as the main framework for the presentation of PRAM algorithms in texts such as [13, 14], validating the second part; see also the class notes available through [1]. For concreteness, we demonstrate WD descriptions on two examples (see text boxes). Example 1 gives a flavor of parallelism in a very simple way. Example 2 demonstrates advantages of the WD methodology and requires some (truly) minimal CS background.

[TEXT BOX BEGINS] *Example 1*: Given are two variables A and B, each containing some value. The *Exchange problem* is to exchange their values; e.g., 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 3 steps: 1. X:=A. 2. A:=B. 3. B:=X. Namely, in order not to overwrite A and lose its content, the content of A is first stored in X, then B is copied to A, and finally the original content of A is copied from X to B. The work in this algorithm is 3, the depth is 3, and the space requirement (beyond the input and output) is 1. Next, consider a generalization of the Exchange problem, called *Array Exchange*. Given two arrays A[0..n-1] and B[0..n-1], each of size n, exchange their content, so that 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. Its pseudo-code follows.

```
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 1. 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 above 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 the serial algorithm. Its depth has improved from 3n to 3. If n is 1,000 this would constitute *speedup by a factor of 1,000* relative to the serial algorithm. The increase in space to n demonstrates a cost of parallelism. [TEXT BOX ENDS]

[TEXT BOX BEGINS] *Example 2*: Consider the directed graph whose nodes are all the commercial airports in the world. There is an edge from node u to node v if there is a non-stop flight from airport u to airport v. s is one of these airports. The problem is to find the smallest number of non-stop flights from s to any other airport. The WD algorithm works as follows. Suppose that: (a) following step i we found the smallest number of non-stop flight from s to all airports that can be reached from s in at most I flights, and (b) all other airports are marked "unvisited". Step i+1 will: (a) concurrently find the destination of every outgoing flight from any airport to which the smallest number of flights from s is exactly i, and (b) for every such destination that is marked "unvisited", mark it as requiring i+1 flights from s. Figure 2 depicts the execution of the algorithm. Starting at the node marked s all the nodes marked 1 are found. Exploring from the outgoing edges from all the nodes marked 1 the nodes marked 2 are found. Some nodes marked 2 have more than one incoming edge. In such cases the Arbitrary CRCW convention implies that one of the attempting writes succeeds. While we don't know which one succeeds we do know that they would all enter the number 2 (in general, however, Arbitrary CRCW allows also different values). Finally, exploring the outgoing edges from all the nodes marked 2 the nodes marked 3 are found, and then the algorithm terminates in Step 4 as no new nodes are

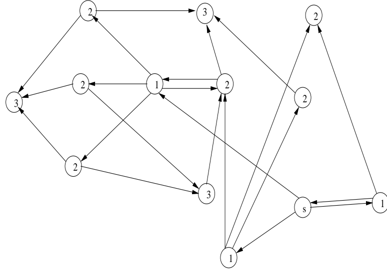

Figure 2: Finding the smallest number of non-stop flights from s to all other airports.

found. The standard serial algorithm for this problem [9] is known as breadth-first search (BFS), and the parallel algorithm above is basically BFS with one difference. Step i+1 above allows concurrent-writes. In the serial version, BFS 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 (FIFO) queue data structure. Two observations are in order: (i) this serial order obstructs the parallelism that BFS offers naturally; the freedom to process in any order nodes for whom the shortest path from s has the same length is lost, and (ii) students trained to incorporate such serial data structures into their program acquire bad serial habits that are difficult to uproot; it may be better to preempt the problem by teaching parallel programming and parallel algorithms early. To demonstrate the advantage of the parallel algorithm over the serial one, assume that the number of edges in the graph is 600,000 (the number non-stop flight links) and the smallest number of flights from airport s to any other airport no more than 6. While the serial algorithm requires 600,000 basic steps, the parallel one requires only 6. While each of the 6 steps may require longer wall clock time than each of the 600,000 steps, the factor 600,000/6 provides much leeway for speedups by a proper architecture. [TEXT BOX ENDS]

Embodiment of the ICE abstraction. By way of the Work-Depth methodology, the vast PRAM algorithmic knowledge base provides a direct embodiment of the ICE abstraction. This is a key point for this paper.

Our broad eXplicit Multi-Threaded (XMT) framework builds on the WD description methodology in two related ways: 1. *The design and analysis of algorithms direction*: The importance of this direction is that it puts the design and analysis of parallel algorithms and data structures on par with their serial counterpart. Given a PRAM with p processors, the approach teaches how to analyze the run-time of an algorithm in a way that resembles the run-time analysis of serial algorithms, as taught by the theory of the field. This direction preceded XMT. PRAM algorithms courses have been taught since the mid-1980s. 2. *The practical programming direction* and its teaching are reviewed next.

3.2 The XMT programming model

The programming model underlying the XMT framework is an arbitrary CRCW SPMD (single program multiple data) programming model that has two executing modes: serial and parallel. The two instructions, `spawn` and `join`, specify the beginning and end of a parallel section (executed in parallel), respectively. See Fig. 3. An arbitrary number of virtual threads, initiated by a spawn and terminated by a join, share the same code. The spawn command extends the embodiment of 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 one committing. No assumption needs to be made beforehand about which will succeed. An algorithm designed with this property in mind permits each thread to progress at its own speed from its initiating spawn to its terminating join, without ever having to wait for other threads; that is, no thread busy-waits for another thread. The implied “independence of order semantics” (IOS) allows XMT to have a shared memory with a relatively weak coherence model. An advantage of using this easier-to-implement SPMD model is that it is also an extension of the classical PRAM model, with its formidable body of parallel algorithms knowledge. The programming model also incorporates the prefix-sum statement. The prefix-sum operates 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 the return value is the initial value of B (such a result is called atomic, and is similar to fetch-and-increment in [11]). The primitive is especially useful when several threads simultaneously perform a prefix-sum 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 will receive a different return value. This way, the parallel prefix-sum command can be used for implementing efficient and scalable inter-thread synchronization, by arbitrating an ordering between the threads. The XMTC high-level language is an extension of standard C. The extensions are described individually in the programmer's manual included in the software release [1]. A parallel region is delineated by spawn and join statements. Synchronization is achieved through the prefix-sum and join commands. Every thread executing 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.

[TEXT BOX BEGINS] <u>*Code examples*</u>: Consider the following example of a small XMTC program for the parallel exchange algorithm of the previous section:

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

The program simply spawns a concurrent thread for each of the depth-3 serial exchange iterations. Our second code example assumes an array of n integers A. We wish 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;
        if(A[$]) != 0){
                ps(e,x);
                B[e] = A[$] }
}
```

The code above declares a variable x as the base value to be used in a prefix-sum command (ps in XMTC), and initializes 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

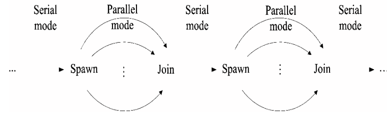

Figure 3: Serial and parallel execution modes.

Figure 4: Left side: FPGA board (the size of a car license plate) comprising three FPGA chips (generously donated by Xilinx). A, B: Virtex-4LX200. C: Virtex-4FX100. Right side: 10mmX10mm chip using IBM Flip-Chip technology.

to get a unique index into B where it can place its value. [TEXT BOX ENDS]

Teaching XMT programming: The author's graduate parallel algorithms course at UMD was upgraded in 2007 to include 6 programming assignments. The in-class instruction hardly discussed XMTC programming. Instead, the students self-studied the programming from the documentation (manual and tutorial) of XMTC. Their feedback regarding ease of programming has been very positive. Consequently, *8* complementary pilots were designed to examine *feasibility of teaching developmentally appropriate parts* of the course to undergraduate seniors, undergraduate freshmen that major or who do not major in CS, high school students at various levels and even middle school students. By Summer 2009, 120 students in grades K-12 will have programmed XMT. 2008/9 courses have been mostly offered by high-school teachers relying on a recent software release [1] and advised by K-12 School of Education learning experts. Although data collection for all pilots has just begun, observed teacher development and students' high-level engagement clearly *indicate the teachability* of parallel algorithmic thinking (PAT) at high school and even middle school levels. The UMD graduate course taught the abstract side first: how to mathematically analyze algorithms in the PRAM model of computation for performance and correctness. Programming came later. In contrast, the college freshmen and K-12 students advance directly from algorithms to programming, similar to teaching of serial programming at these levels, and learning methods in the field of Mathematics education. There, an activity-effect relationship model of learning and teaching (e.g., [16]) builds on proper sequencing of activities (i.e., programming) and learning.

3.3 XMT architecture and hardware

The *eXplicit Multi-Threading (XMT[1])* on-chip general-purpose computer architecture is aimed at the classic goal of reducing single task completion time. The WD methodology equips the algorithm designer with the ability to express all the parallelism that he/she observes. XMTC programming further permits expressing this virtual parallelism by "dreaming up" as many concurrent threads as the programmer wishes. To complete the embodiment of the ICE abstraction, the XMT processor must provide an effective way for mapping 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 very short, the XMT hardware must directly manage XMT threads. In particular, an XMT program looks like a single thread to the operating system (OS). The text box "the XMT processor" reviews the XMT hardware and provides further links for more information.

Commitments to silicon of XMT include a 64-processor, 75MHz computer[2] based on field-programmable gate array (FPGA) technology [21], and 64-processor ASIC 10mmX10mm chip using IBM' s 90nm technology, pictured in Figure 4. A basic yet stable compiler has also been developed.

[TEXT BOX BEGINS] The XMT processor (see Fig 5) includes a master thread control unit (MTCU), processing clusters each comprising several thread-control units (TCUs), a high-bandwidth low-latency interconnection network, memory modules (MM) each comprising on-chip cache and off-chip memory, a global register file (GRF) and a prefix-sum unit. Fig. 5 suppresses the sharing of a memory controller by several MMs. The processor alternates between serial mode, where only the MTCU is active, and parallel mode. The MTCU has a standard private data cache used only in serial mode and a standard instruction cache. The TCUs do not have a write data cache. They and the MTCU all share the MMs. Due to space limitation we need to refer the reader to [21] and prior XMT papers for a description of the way in which: (i) the XMT apparatus of the program counters and stored program extends the standard von-Neumann serial apparatus, (ii) virtual threads coming from an XMTC program are allocated dynamically at run time, for load balancing, to TCUs, (iii) hardware implementation of the PS operation and its coupling with a global register file (GRF), (iv) a more general design ideal, called no-busy-wait finite-state-machines (NBW FSM), guides the overall design of XMT. In principle, the MTCU is an advanced serial microprocessor that can also execute XMT instructions such as spawn and join. Typical program execution flow was shown in Fig. 3. The MTCU broadcasts the instructions in a parallel section, which starts with a spawn command and ends with a join command, on a bus connecting to all TCU clusters. In parallel mode a TCU can execute one thread at a time. TCUs have their own local registers and they are simple in-order pipelines including fetch, decode, execute/memory-access and write back stages. The FPGA computer has 64 TCUs in 4 clusters of 16 TCUs each. (We aspire to have 1024 TCUs in 64 clusters in the future). A cluster has functional units shared by several TCUs and one load/store (LS) port to the interconnection network, shared by all its TCUs. The global memory address space is evenly partitioned into the MMs using a form of hashing. In particular, the cache-coherence problem, a challenge for scalability, is eliminated: in principle, there are no local caches at the TCUs. Within each MM, order of operations to the same memory location is preserved; a store operation is acknowledged once the cache module accepts the request, regardless if it is a cache hit or miss. A ready-to-run version of an XMT program seeks to optimize: (i) the length of the (longest) sequence of round trips to memory (LSRTM), (ii) queuing delay to the same shared memory location (known as queue-read queue-write, QRQW), and (iii) work and depth

[1] The University System of Maryland (USM) has selected an XMT-based proposal for a Maryland Research Center of Excellence for Reinvention of Computing for Parallelism in a competition across all fields among 5 out of 50 proposals.

[2] A naming contest for this XMT computer by UMD received nearly 6000 submissions. The name Paraleap was selected.

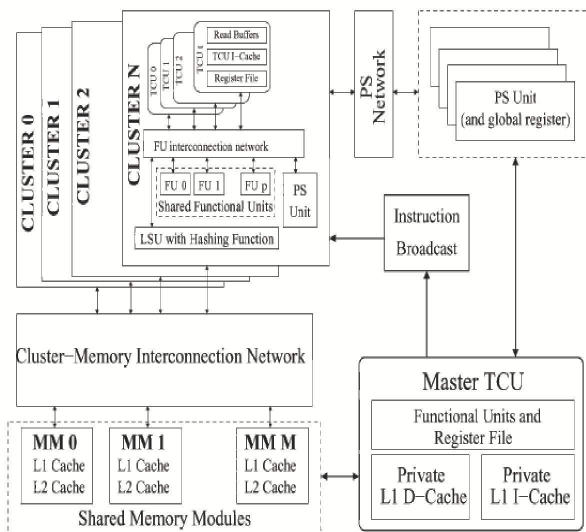

Figure 5: A block diagram of the XMT architecture.

(as above). *Optimizing these ingredients is a responsibility shared in a subtle way between the architecture, the compiler, and the programmer/algorithm designer*. See [20] for more information.

Representative performance enhancements include: (i) Broadcast: if most threads in a spawn-join section need to read a variable, it is broadcasted through the instruction broadcasting bus to TCUs rather than reading the variable serially from the shared memory. (ii) Software prefetch mechanism with hardware support to alleviate the interconnection network round trip delay. A prefetch instruction brings the data to a prefetch buffer at the TCUs. Cost-effectiveness of nesting spawn commands and current and future ways for expressing nested spawns in a program is also beyond the scope of this paper (see [20]). [TEXT BOX ENDS]

4. RELATED EFFORTS

Related efforts come in several flavors. Valiant's Multi-BSP bridging model for multi-core computing [19] appears closest to our focus on abstraction, but since [19] models relatively low-level parameters of certain multi-core architectures, it is closer to [20] than to the current paper. In contrast to both these papers, simplicity drives the "one-liner" ICE abstraction.

Parallel languages, such as CUDA, MPI, or OpenMP tend to be different than computation models, as they often do not involve performance modeling. Languages require a level-of-detail that distances them further from simple abstractions; however, [3] is pursuing an interesting approach for mitigating that.

Several industry funded research centers consider the general problems discussed in this paper. Their main output so far have been agendas, informative reviews and opinion pieces, such as [4, 5, 17]. The UC-Berkeley Parallel Computing Lab and Stanford's Pervasive Parallelism Laboratory tend to be application-driven. For example, UC-Berkeley advocates advancing from applications to a many core platform through quite a few design patterns, each fitting a group of applications. Such domain-specific abstractions, have also been already incorporated by XMT (see Figure 6), but are quite different than the (general-purpose) one-size-fits-all ISE-like ideal, that ICE seeks.

An alternative PRAM-inspired many-core effort is noted in [2].

5. CONCLUSION

Features of the serial paradigm that made it such a success include: a simple abstraction at the heart of the "contract" between programmers and builder, the software spiral, ease-of-programming and ease-of-teaching, and backwards compatibility on serial code and on application programming. We have presented an approach that reproduces many of these features for the impending transition to the many-core era. The only exception is that, like everybody else, we do not provide speedups for serial code. The approach is promising as the field can modernize itself, while holding on to these attractive features.

Besides proposing focus on abstraction, we presented the ICE abstraction and its XMT/PRAM embodiment.

5.1 The ICE abstraction

Abstraction has facilitated computer-related breakthroughs before. The memory hierarchy had been a challenge for serial computing. The IMA abstraction addressed that. Hardware needs more time to execute some operations than others. ISE extended IMA by abstracting that, but is insufficient to address parallel processing. Abstractions have played an important role in parallel programming models. They should play an even greater role for many-cores, due to the broad access sought. This paper presents such a candidate abstraction along with a comprehensive embodiment for it. *Our main point is that the ICE abstraction, coupled with an XMT/PRAM platform provide a viable option for the many-core era*. Our solution will hopefully inspire others to come up with competing abstraction proposals, or alternative embodiments for ICE. Consensus built around an abstraction will move us closer to convergence to a many-core platform and to putting the software spiral back on track. Note that the paper concludes with a paragraph explaining why the ICE abstraction reconnects the training of CS students with the future needs of the field regardless of whether it becomes the abstraction of choice for many-cores. Interestingly, *teaching has become a CS research issue*:

Teachability as a benchmark: Ease-of-programming (programmability) is a necessary condition for the success of a many-core platform and teachability is a necessary condition for programmability. We have been using teachability at various K-20 levels as a benchmark for ICE and XMT and suggest that others do that as well.

5.2 XMT

The feature of XMT that received most attention in this paper is its practical implementation of the ICE abstraction. However, **XMT is a comprehensive and coherent solution for the many-core era in its own right**. XMT accounts for: application programming (VHDL/Verilog, OpenGL, MATLAB, etc), parallel algorithms, parallel programming, compiling, architecture, power, deep-submicron implementation, and backwards compatibility on serial code. We mentioned a 64-processor, 75MHz XMT FPGA-based computer prototype [21], 90nm ASIC tape-outs, and a basic compiler. XMT is easy to build. A single graduate student, with no prior design experience, completed the XMT hardware description (in Verilog) in just over 2 years. XMT is also silicon-efficient. Our ASIC design indicates that a 64-processor XMT needs the same silicon area as a (single) current commodity core. The approach goes after any type of application parallelism regardless of its amount, reg-

ularity, or grain size and is amenable to standard multiprogramming (i.e., where the hardware supports several concurrent OS threads). We also demonstrated good performance, programmability and teachability. Highlights include: evidence of 100X speedups on general-purpose applications on a simulator of 1000 on-chip processors [12], and speedups ranging between 15X to 22X for irregular problems such as Quicksort, breadth-first search (BFS) on graphs, finding the longest path in a directed acyclic graph (DAG), and speedups in the range 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 [21]. With few exceptions, parallel programming approaches that dominated parallel computing prior to many-cores are still favored by vendors and high-performance computing user communities. These approaches require steps such as: decomposition, assignments, orchestration and mapping, from the programmer [10]. Indeed, parallel programming difficulties have failed all general-purpose parallel systems to date by limiting their use. In contrast, XMT frees its programmer from doing any of that, in line with the ICE abstraction. The left side of Figure 6 compares serial programming (the path that ends with 1), parallel programming along the line of [10] (path 2) and XMT parallel programming (path 4), for performance programming in languages such as C for serial programming, MPI or OpenMP for standard parallel programming, and XMTC for XMT. Path 3 represents the very limited success in extracting parallelism from serial code using compiler, in spite of significant efforts over four decades. The right side compares application programming, where it is envisioned that the same application languages used today with some minor restrictions (such as not using serial programming style code in MATLAB) will be directly compiled to run on XMT.

<ins>*Software release*</ins>: The XMT environment is available for immediate adoption in the form of a just released XMTC compiler and cycle-accurate simulator of XMT that can be downloaded to any standard desktop computing platform. This software release is available through the XMT home page, or sourceforge.net [1] along with extensive documentation. Teaching materials comprising a class-tested programming methodology, where college freshmen and even high-school students are taught only parallel algorithms and then self-study XMT programming, are also provided.

<ins>*For teaching parallelism*</ins>: Most CS programs graduate students to a job market certain to be dominated by parallelism without needed preparation. We propose to base the introduction of the new generation of CS students to parallelism on the XMT platform, at least until convergence to a many-core platform is achieved. The *level of cognition of parallelism required by the ICE abstraction is so basic that it is* **necessary** *for all other current approaches.* XMT is buildable, which makes the XMT approach also "***sufficient***".

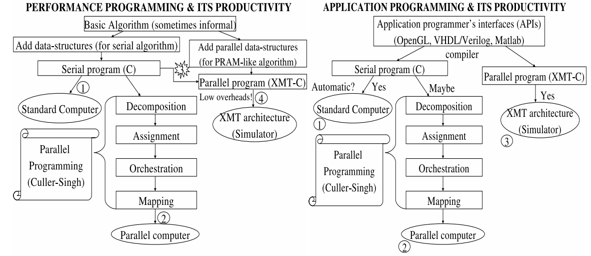

Figure 6: *Left side*: Productivity of performance programming.

6. REFERENCES

[1] *Explicit Multi-Threading (XMT): home page http://www.umiacs.umd.edu/users/vishkin/XMT/ and software release http://sourceforge.net/projects/xmtc/.*

[2] *http://www.plurality.com/PR081106.pdf.*

[3] *White paper: Parallel computing without parallel programming, www.interactivesupercomputing.com/.*

[4] S. Adve and et al. Parallel computing research at Illinois - the UPCRC agenda, U. Illinois. 2008.

[5] K. Asanovic and et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, UC Berkeley, 2006.

[6] D. Atkins. *Revolutionizing Science and Engineering Through Cyberinfrastructure: NSF Blue-Ribbon Advisory Panel on Cyberinfrastructure*, 2003.

[7] S. Borkar and et al. Platform 2015: Intel processor and platform evolution for the next decade. Intel. 2005.

[8] A. Burks, H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computer instrument. 1946.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2001.

[10] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan-Kaufmann, 1999.

[11] A. Gottlieb and et al. The NYU ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Trans. Computers*, 32,2:175–189, 1983.

[12] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocesor. *J. Embedded Comp.*, 2:181–190, 2006.

[13] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[14] J. Keller, C. Kessler, and J. Traeff. *Practical PRAM Programming*. Wiley-Interscience, 2001.

[15] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[16] M. Simon, R. Tzur, K. Heinz, and M. Kinzel. Explicating a mechanism for conceptual learning: Elaborating the construct of reflective abstraction. *J. Research in Mathematics Education*, 35:305–329, 2004.

[17] M. Snir. Multi-core and parallel programming: Is the sky falling? The Computing Community Consortium Blog, http://www.cccblog.org/2008/11/17/multi-core-and-parallel-programming-is-the-sky-falling/.

[18] H. Sutter. The free lunch is over - a fundamental shift towards concurrency in software. *Dr. Dobbs Journal*, 30, March 2005.

[19] L. Valiant. A bridging model for multi-core computing. In *Proc. European Symp. on Algorithms*. LNCS, Vol 5193, Springer-Verlag, 2008.

[20] U. Vishkin, G. Caragea, and B. Lee. *Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform. In Handbook on Parallel Computing (Eds S. Rajasekaran, J. Reif)*. Chapman and Hall/CRC Press, 2008.

[21] X. Wen and U. Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proc. ACM Computing Frontiers*, Ischia, Italy, May 2008.