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 ones 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 enough to program by at least all CS majors. Unfortunately, lesser ease-of-programming objectives have eluded decades of parallel computing research. The idea of starting with an established easy parallel programming model and build an architecture for it has been treated as radical by vendors. This article advocates a more radical idea. Start with a minimalist stepping-stone: a simple *abstraction* that encapsulates the *desired* interface between programmers and system builders.

An *Immediate Concurrent Execution (ICE)* abstraction proposal is followed by two specific contributions: (i) A general-purpose many-core *Explicit Multi-Threaded (XMT)* computer architecture. XMT was designed from the ground up to capitalize on the huge on-chip resources becoming available in order to support the formidable body of knowledge, known as PRAM (for parallel random-access machine, or model) algorithmics, and the latent, though not widespread, familiarity with it. (ii) A *programmer's workflow* that links: ICE, PRAM algorithmics and XMT programming. The synchronous PRAM provides ease of algorithm design, and ease of reasoning about correctness and complexity. Multi-threaded programming relaxes this synchrony for implementation. Directly reasoning about soundness and performance of multi-threaded code is generally known to be error prone. To circumvent that, the workflow incorporates multiple levels of abstraction: the programmer must only establish that the multi-threaded program behavior matches the 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 we may be much better prepared for the challenges ahead than many realize.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel architectures; D.1.3 [**Programming Techniques**]: Parallel programming; I.1.2 [**Computing Methodologies**]: Algorithms

General Terms

Algorithms, Design, Performance

Keywords

Parallel computing, Parallel algorithms, Abstraction

1. ABSTRACTING PARALLELISM

The following rudimentary abstraction made serial computing simple: *that any single instruction available for execution in a serial program executes immediately.* Abstracting away a hierarchy of memories, each with greater capacity, but slower access time than the preceding one, and different execution time for different operations, this abstraction has been used by programmers to conceptualize serial computing and supported by hardware and compilers. A program provides the instruction to be executed next at each step (inductively). The left side of Figure 1 depicts serial execution as implied by this serial abstraction, where unit-time instructions execute one at a time.

The rudimentary parallel abstraction we propose is: *that indefinitely many instructions, which are available for concurrent execution, execute immediately*, and dub the abstraction *immediate concurrent execution (ICE).* A consequence of ICE is a step-by-step (inductive) explication of the instructions that are available next for concurrent execution. The number of instructions in each step is independent of the number of processors. In fact, processors are not even mentioned. The explication falls back on the serial abstraction in case of one instruction per step. The right side of Figure 1 depicts parallel execution as implied by the ICE abstraction. 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.

How can parallelism be advantageous.

The PRAM answer is that in a serial program 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 program it can be much lower. The objective for a parallel program is that its work will not much exceed that of its serial counterpart for the same problem, and its depth will be much lower than its work. Section 3.2 notes the straightforward connection between ICE and the rich PRAM algorithmic theory; and ICE is nothing more than a subset of the work-depth model. But, how to go about

.

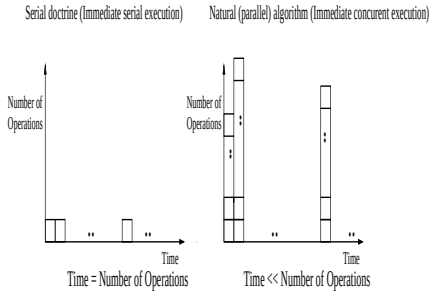

Figure 1: Serial execution based on the serial abstraction versus parallel execution based on the ICE abstraction.

building a computer system that realizes the promise of ease-of-programming and strong performance?

This article offers a comprehensive solution to this profound question. Section 3.1 discusses basic tensions between the PRAM abstraction and hardware implementation. Section 3.3 then describes a workflow that goes through ICE and PRAM-related abstractions for programming effectively the explicit multi-threaded (XMT) computer architecture.

Some many-core architecture is expected to become mainstream. To become mainstream, architecture will have to be easy enough to program by every CS major. We are not aware of other many-core architectures whose abstraction is PRAM-like. Allowing the programmer to view a computer as a PRAM makes it easy to program [11]. Hence, this article could interest all such majors.

2. INTRODUCTION

Until 2004, standard (desktop) computers comprised a single processor core. Since 2005 we appear to be on track with a prediction [6] 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 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. By 2003, the clock rate of a high end desktop processor reached 4GHz, but clock rates of processors have hardly improved ever since. The industry did not find a way to continue improving clock rates within acceptable power budgets [6]. Fortunately, silicon technology improvements such as miniaturization allow the amount of logic that a computer chip can contain to continue growing, doubling every 18 to 24 months. 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* is now the only way to improve performance of single tasks from one generation of computers to the next.

Unfortunately, chipmakers are busy designing multicore processors that most programmers can't handle [20]. Andy Grove (Intel) noted that the *software spiral* (the cyclic process of hardware improvements leading to software improvements, which lead back to hardware improvements and so on) had been an engine of sustained growth for Information Technology for many decades. A stable application-software base that could be reused and enhanced from one hardware generation to the next was available. Better performance was assured with each generation if only the hardware could run serial code faster. Alas, *the software spiral is now broken* (cf. [22]): (a) There is no broad parallel computing application software base for which hardware vendors are committed to improve performance. (b) No agreed-upon parallel architecture currently allows application programmers to build such software base for the future. Instating a new software spiral could be a "killer application" for general-purpose many-core computing; application software developers will put it to good use for specific application, and more people will want to buy new machines. This leads to the following.

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. A software spiral is basically an *infrastructure* for the economy. Since advancing infrastructures generally merits government funding, designating software spiral reinstatement as a killer application allows also funding agencies to support the work.

3. PROGRAMMER'S WORKFLOW AND THE XMT ARCHITECTURE

3.1 Motivation and Preview

[TEXT BOX BEGINS] One of the English 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 *expand*). [TEXT BOX ENDS]

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 [11]. In CS theory, the speedup provided by parallelism is measured as work divided by depth, but reducing the advantage of ICE/PRAM to practice is a whole different matter. Our reduction to practice relies on the programmer's workflow, depicted in the right columns of Figure 2 and reviewed later in this section, and the XMT architecture. Section 3.2 briefly recalls the parallel algorithms stage, as developed through the mid-1990s. The step-by-step explication of PRAM (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 power constraints). Consider two examples: (i) Memories based on long tightly-synchronous pipelines of the type seen in the Cray vector machines have been long out of favor. (ii) Processing memory requests takes anywhere from one to 400 clocks; to be effective hardware must be made as flexible as possible to advance without unnecessary waiting for concurrent memory requests.

Section 3.3 is on the programming stage of the workflow. It is followed up by discussion on performance tuning, and a limited review of the computer system comprising computer architecture, actual hardware, and compiler. This latter material also connects to the overall effort to *relax to the extent possible the tightly synchronous base of PRAM algorithms.* To underscore the importance of the bridge that our approach builds from the tightly synchronous PRAM to a relaxed synchrony implementation we note some known issues with the power consumption of current multi-core architectures:: (i) the high power consumption of the wide communication buses needed to implement cache coherence; (ii) the basic nm complexity of cache coherence traffic (given n cores and m invalidations) and its implied huge toll on inter-core bandwidth; and (iii) the high power consumption needed for a *tightly synchronous implementation* in silicon used in these

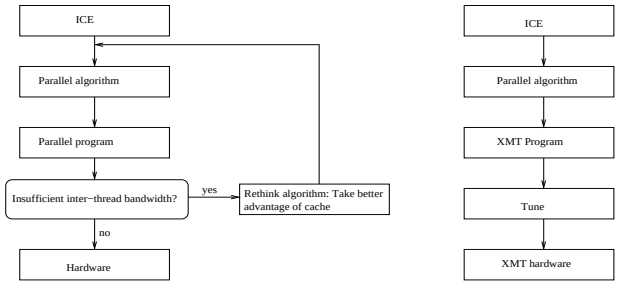

Figure 2: The right column depicts a workflow from an ICE abstraction of an algorithm to implementation, while the left column may never terminate.

designs. Our approach addresses these three issues by avoiding hardware-supported cache-coherence altogether and by significantly relaxing synchrony.

Preview of the Workflow and XMT.

Workflows are important as they guide the human-to-machine process of programming. Figure 2 depicts two attempts at workflows. The non-XMT hardware implementation in the left column may require that the algorithm is revisited and changed to fit bandwidth constraints among threads of the computation, a programming process that sometimes may never lead to an acceptable outcome. However, the XMT hardware allows a workflow that *requires only tuning for performance* (right column of Figure 2); revisiting and possibly changing the algorithm is generally not needed. In fact, an optimizing compiler should be able to do the tuning without intervention from the programmer, similar to serial computing.

Most of the programming effort involved in traditional parallel programming (domain partitioning, load balancing), can be of lesser importance for exploiting on-chip parallelism, where parallelism overhead can be made low and processor-to-memory bandwidth high. This observation drove the development of the XMT programming model and its implementation. XMT is intended to provide: 1) a simpler parallel programming model that 2) efficiently exploits on-chip parallelism. These two goals are achieved by a number of 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 (whose count can reach the thousands). Specifically, XMT can: (i) forward *(at once)* program instructions to all processors within the same time required to forward the instructions (for one thread) just to one processor; and (ii) reallocate any number of processors that complete their jobs at the same time to new jobs (along with their instructions) within the same time required to reallocate one processor. The high-bandwidth low-latency interconnection network and the low-overhead creation of many threads allow efficient support of fine-grained parallelism. This fine granularity is used to hide memory latencies and allows a programming model for which locality is less of an issue. The above mechanisms support dynamic load balancing, relieving the programmers of the task of assigning work to processors. The programming model is simplified further by seeking to let threads run to completion without synchronization (no busy-waits), and synchronizing accesses 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 wider range of applications.

[TEXT BOX BEGINS] *Postscript: Not a monolithic outlook* The reinvention of computing for parallelism requires pulling together quite a few communities. The recent paper [27] seeks to build a bridge to other architectures by casting the abstraction-centric vision of this article as a possible module in them. [27] identifies a limited number of capabilities that the module provides, and suggests a preferred embodiment of these capabilities using concrete "hardware hooks". If it is possible to augment a computer architecture with these capabilities (using these hardware hooks, or by other means), the ICE abstraction and the programmer's workflow, in line with this article, can be supported. [TEXT BOX ENDS]

3.2 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 [10]. RAM, the basic serial model underlying standard programming languages, assumes that any memory access or any (logic, or arithmetic) operation takes unit-time (the serial 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, PRAM 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 [21]) 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 methodology of restricting attention only to work and depth has been used as the main framework for the presentation of PRAM algorithms in texts such as [17, 18]; see also the class notes available through [1]. It is easy to derive a full PRAM description from a WD description. 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.
[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 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 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 2n (for array X and n concurrent values of i) 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 flights 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. Note that some "unvisited" nodes may 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 i+1 (in general, however, Arbitrary CRCW allows also different values). The standard serial algorithm for this problem [10] 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 of non-stop flight links) and the smallest number of flights from airport s to any other airport is no more than 5. 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]
The programmer's workflow starts with the easy to think ICE abstraction and ends with the XMT system, providing a practical implementation of the vast PRAM algorithmic knowledge base.

3.3 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 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 one committing. No assumption needs to be made beforehand about which one succeeds. 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 PRAM-like. 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 [13]). 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 among the threads.

The XMTC high-level language implements the programming model. XMTC is an extension of standard C. XMTC augments C with a small number of commands, such as spawn, join and prefix-sum. A parallel region is delineated

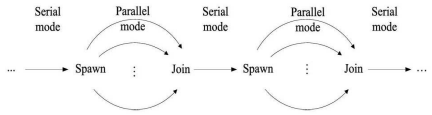

Figure 3: Serial and parallel execution modes.

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. For the hardware implementation noted later, XMTC threads can be as short as providing 8-10 machine instructions, which is not difficult to get from PRAM algorithms. Programmers are pleasantly surprised by the flexibility of translating PRAM algorithms to XMTC multi-threaded programs. Being able to code the whole merging algorithm (see below) using a single spawn-join pair is one such surprise.
[TEXT BOX BEGINS] *Two simple code examples*: Consider the following example of a small XMTC program for the parallel exchange algorithm of the previous section:

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

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

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

[TEXT BOX BEGINS] *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 needs to be mapped into an array $C = C[1 \ldots 2n]$ which is also sorted. We first review Shiloach-Vishkin's 2-step PRAM algorithm for merging and then discuss its XMTC programming. The two steps are: (i) *Partitioning*. This step first selects some number x of elements from A at equal distances. In the example of Figure 4, suppose that the four elements 4,16,20 and 27 are selected. Each of these elements is then 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, and 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 needs to be completed into $2x$ "strips" (Step 2 in Figure 4 shows 8 strips). (ii) *Actual work*. For each of these strips the remaining merging job is to merge a subarrary of A with a subarray of B, mapping their elements into a subarray of C. Since these $2x$ merging jobs are mutually independent, each can concurrently apply the standard linear-time serial merging algorithm. The complexity analysis of this algorithm follows. Since each strip can have at most n/x elements from A and n/x elements from B, the depth (or parallel time) of step (ii) is $O(n/x)$. If $x \leq n/\log n$, Step 1 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 will require as many spawn (and respective join) commands as the number of PRAM steps. The reason for presenting this example in this article is that there is a way to use only *a single spawn (and a single join) command to represent the whole merging algorithm. Merging in XMTC:* An XMTC program would spawn $2x$ concurrent threads, one for each of the selected elements in array A or B. Using binary search, each thread will first rank its array element relative to the other array. It will then *proceed directly (without a join operation)* to merging the elements in its strip, terminating just before setting the merging result of another selected element. The reason is that this merging result is computed by another thread. *Example:* Consider the thread of element 20. Starting with binary search on array B it 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); it then compares 23 to 22 to rank 22, and compares 23 to 24 to rank 23; it then compares 24 to 25, but terminates since the thread of 24 will rank 24, concluding the example. Our general experience has been that often, with little effort, XMT-type threading requires fewer synchronizations than literally implied by the original PRAM algorithm. The XMTC merging example demonstrates that sometimes the reduction in synchronizations can be big. [TEXT BOX ENDS]

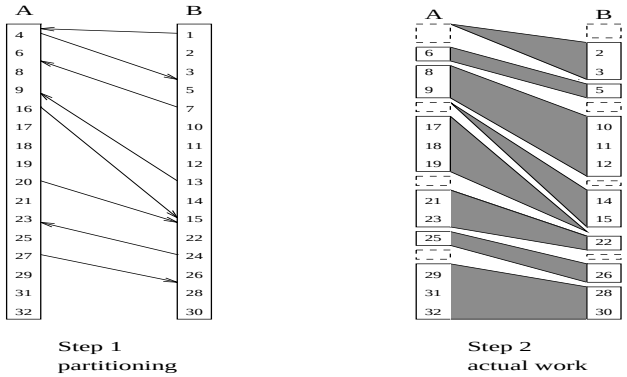

Figure 4: Main steps of the ranking/merging algorithm

Other XMTC commands. Prefix-sum-to-memory (psm) is another prefix-sum command whose base can be 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 can be nested. A nested spawn command in XMTC code needs to be replaced (by the programmer, or compiler) by sspawn commands. The

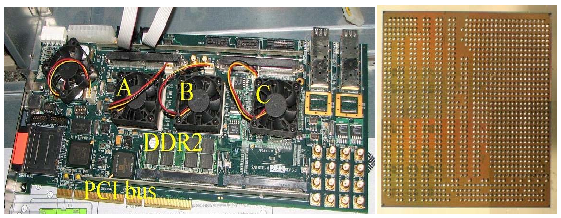

Figure 5: Left side: FPGA board (the size of a car license plate) comprising three FPGA chips (generously donated by Xilinx). A, B: Virtex-4LX200. C: Virtex-4FX100. Right side: 10mm X 10mm chip using IBM Flip-Chip technology.

XMTC commands are described in the programmer's manual included in the software release [1].

3.4 Tuning XMT programs for performance

Our discussion of performance tuning requires an overview 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 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. The XMT processor must now 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. The main thing that a performance programmer needs to know in order to tune the performance of their XMT program is reviewed next. 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 [12]), and (iii) work and depth (as above). *Optimizing these ingredients is a responsibility shared in a subtle way between the architecture, the compiler, and the programmer/algorithm designer*. See "Tuning example" for how accounting for LSRTM can improve performance.

[TEXT BOX BEGINS] *Tuning example*. Given n numbers in an array $A[1..n]$, consider the problem of computing their sum $A(1) + A(2) + .. + A(n)$. The standard parallel algorithm for this summation problem is guided by a balanced binary tree. Assuming for simplicity that n is a power of 2, the algorithms works in $\log_2 n$ parallel steps. The first step comprises $n/2$ pairwise additions: $A(1) + A(2)$, $A(3) + A(4)$, .., $A(n-1) + A(n)$. The second step has $n/4$ additions, producing: $A(1) + A(2) + A(3) + A(4)$, .., $A(n-3) + A(n-2) + A(n-1) + A(n)$, and so on till step $\log_2 n$ that produces the sum $A(1) + A(2) + .. + A(n)$ in a single addition. It can be readily seen that this algorithm requires $O(n)$ work and $O(\log n)$ time, and that it can be cast using $\log_2 n$ spawn-join pairs. However, a closer looks suggests some performance improvements: (i) With p processors, have first each processor sum serially a separate group of n/p elements, reducing the original problem to summation of p elements. (ii) The summation of p elements may be done faster if guided by a balanced k-ary tree, where each node of the tree has k children (a balanced binary tree is a k-ary tree for $k = 2$). Namely, do the summation in $\log_k p$ rounds. In the first round compute p/k sums: $A(1)+..+A(k)$, $A(k+1)..A(2k)$,.. In the second round, compute p/k^2 sums, and so on. But, how to chose the optimal k value? For XMT, the dominant parameter that guides the selection of the optimal k value, and navigate among the different implementation options is LSRTM [28]. The number of time units needed for a round-trip from the TCU clusters in Figure 6 across the cluster-memory interconnection network depends on the XMT hardware. We then need to analyze the code in order to figure out the extent to which the round-trip required by different commands can overlap in time (e.g., using prefetching into prefetch buffers at the TCUs) and therefore be pipelined. LSRTM captures the number of round-trips that cannot overlap. We refer the reader to [28] for more information. [TEXT BOX ENDS]

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, this could be done by clustering virtual threads off-line or on-line, and prioritization of execution of nested spawns using known heuristics based on a mix of depth-first and breadth-first searches.

Commitments to silicon of XMT include a 64-processor, 75MHz computer based on field-programmable gate array (FPGA) technology [29], and 64-processor ASIC 10mm X 10mm chip using IBM' s 90nm technology, pictured in Figure 5. A basic yet stable compiler has also been developed.

[TEXT BOX BEGINS] The XMT processor (see Fig 6) includes a master thread control unit (MTCU), processing clusters each comprising several thread-control units (TCUs), a high-bandwidth low-latency interconnection network (see [4] and its extension to Globally-Asynchronous Locally-Synchronous, known as GALS-style, design incorporating asynchronous logic in [19, 16]), memory modules (MM) each comprising on-chip cache and off-chip memory, prefix-sum (PS) unit(s) and global registers. Fig. 6 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 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.

The overall design of XMT is guided by a general design ideal, called no-busy-wait finite-state-machines (NBW FSM). The NBW FSM ideal is that the FSMs (processors, memories, functional units, interconnection networks, etc) comprising the parallel machine will never cause one another to busy-wait. We use the term ideal because it would be untenable for a parallel machine to operate that way. Non-trivial parallel processing demands exchange of results among FSMs. The NBW FSM ideal represents an aspiration to reduce to the possible minimum busy-waits among the various FSMs that comprise the machine. Below, we use the example of how the MTCU orchestrates the TCUs in order to demonstrate the NBW FSM ideal.

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, but it 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 that starts with that spawn command and ends with a join command on a bus connecting to all TCU clusters. The largest ID number of a thread that the current spawn command needs to 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 can execute 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 needs to execute next; if the ID is less than or equal Y, the TCU executes a thread with this ID. Otherwise, the TCU reports to the MTCU that it finished. Once all TCUs report that they finished, the MTCU continues in serial mode. The broadcast operation is essential to the XMT ability to start all TCUs at once within the same time it takes to start one TCU. The use of the PS unit allows allocation of new threads to the TCUs that just became available within the same time of allocating one thread to one TCU. This dynamic allocation provides run time load-balancing of threads coming from an XMTC program. We are ready to make a connection with the NBW FSM ideal. Consider an XMT program derived from the workflow. From the moment the MTCU starts executing a spawn command till the time each TCUs terminates the threads allocated to it, no TCU can cause any other TCU to busy-wait for it. The busy-wait of course happens once a TCU terminates (and begins waiting for the next spawn command).

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 bandwidth and 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.

For performance enhancements incorporated in the XMT hardware such as data prefetch and more information about the architecture, see [29]. Compiler and run-time scheduling methods for nested parallelism are discussed in [24] and prefetching methods in [9]. Patents supporting the XMT hardware appeared in [26, 19]. [TEXT BOX ENDS]

3.5 Other Information and Comments

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, regularity, 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 (e.g., [15]) and teachability (e.g., [23]). Highlights include: evidence of 100X speedups on general-purpose applications on a simulator of 1000 on-chip processors [14],

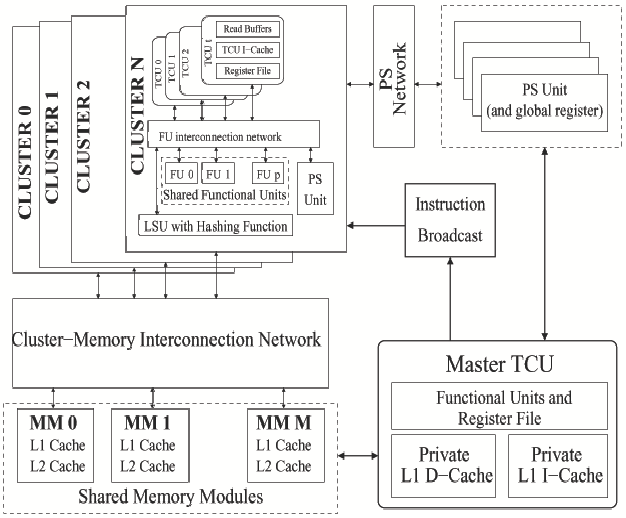

Figure 6: A block diagram of the XMT architecture.

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 [29]. The paper [8] demonstrates nearly 10X average performance improvement potential relative to Intel Core 2 Duo for a 64-processor XMT chip that uses the same silicon area as a single core. The recent paper [7] demonstrates that, using the same silicon area as a modern graphics processing unit (GPU), our design achieves an average speedup of 6X relative to the GPU for irregular applications and falls only slightly behind on regular ones. All the GPU code was written and optimized by others.

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 [11]. 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 these, in line with the ICE/PRAM abstraction.
Software release: The XMT environment is available for immediate adoption. A recent release of the XMTC compiler and a cycle-accurate simulator of XMT 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.
For teaching parallelism: Most CS programs graduate students to a job market certain to be dominated by parallelism without needed preparation. The *level of cognition of parallelism required by the ICE/PRAM abstraction is so basic that it is* **necessary** *for all other current approaches*. We propose to base the introduction of the new generation of CS students to parallelism on the presented workflow, at least until convergence to a many-core platform is achieved.

Note that since XMT is buildable the XMT approach is also "*sufficient*".

4. RELATED EFFORTS

Related efforts come in several flavors. Valiant's Multi-BSP bridging model for multi-core computing [25] appears closest to our focus on abstraction. The main difference is in the intentions: our modeling seeks to guide builders of new machines to incorporate desired features, as opposed to capturing existing features. These *prescriptive versus descriptive* objectives are not the only difference. [25] models relatively low-level parameters of certain multi-core architectures, which makes it closer to [28] than to this article. 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.

Several research centers consider the general problems discussed in this article [2, 3]. The UC-Berkeley Parallel Computing Lab and Stanford's Pervasive Parallelism Laboratory advocate an application-driven approach to reinventing computing for parallelism.

5. CONCLUSION

The vertical integration offered by the XMT framework with the ICE/PRAM abstraction as its front-end is quite unique. ICE is a newly separated feature that did not appear in prior papers, *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 only on mathematical induction, itself one of the more rudimentary concepts of Mathematics. The suggestion to have a simple abstraction become the lead guide of the discussion on the reinvention of computing for parallelism appears to also be new. This article provides evidence that this can be done.

[TEXT BOX BEGINS] Eye-of-a-needle aphorism. The XMT processor text box recalls the von-Neumann apparatus of stored program and program counter. Introduced at a time of extreme hardware scarcity, this apparatus forced threading of instructions through a metaphoric eye-of-a-needle. The coupling of mathematical induction and the ISE abstraction was engineered to provide this threading. This eye-of-a-needle threading is evident in serial examples of this article. See: (i) Example 1 of Section 3.2, in the use of variable X in the pseudo-code of the serial iterative algorithm for the exchange problem; (ii) Example 2 of Section 3.2, in the FIFO queue data structure in the serial BFS; and (iii) the serial merging algorithm, noted indirectly in Section 3.3 in which two elements are compared at a time, one from each of the two sorted input arrays. Having become a second nature for many programmers, eye-of-a-needle threading is often associated with ease of programming. Interestingly, threading through an eye-of-a-needle is considered an aphorism for extreme difficulty, or even impossibility in the broader culture, including texts of three major religions. The XMT extension to the von-Neumann apparatus noted in the XMT processor text box uses today's greater hardware resources to free computing from the constraint of threading through the original apparatus. The coupling of mathematical induction and the ICE abstraction of this article is engineered to capitalize on this freedom for ease of parallel programming and improved performance. [TEXT BOX ENDS]

The following comparison with a new multithreading algorithms chapter in the 2009 third edition of [10] can help elucidate some contributions of this article. The 1990 first edition included a chapter on PRAM algorithms with a prominent role for work-depth design and analysis. The 2009 chapter retains work-depth analysis. However, to match current hardware, the new chapter resorts to a variant of dynamic multithreading (in-lieu of work-depth design) whose main primitive is similar to the XMT sspawn command (Section 3.3) that starts one additional thread a time. One thread can only generate one more thread; these two threads can generate one more thread each and so on, instead of directly designing for the work-depth analysis that follows. The [10] dynamic multithreading direction should encourage hardware enhancement that will allow starting at once many threads within the same time required to start one thread. A step ahead of available hardware, XMT has already demonstrated 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, more similar to the first edition of [10]. The $O(\log n)$ depth parallel merging algorithm above versus the $O(\log^2 n)$ depth one in [10], demonstrates an XMT advantage over current hardware. *In summary*, the XMT hardware scheduling brought the hardware performance model much closer to work-depth and allowed our workflow to streamline the design with the analysis from the start.

Features of the serial paradigm that made it such a success include: a simple abstraction at the heart of the "contract" between programmers and builders, the software spiral, ease-of-programming and ease-of-teaching, and backwards compatibility on serial code and on application programming. The only feature that we, like everybody else, do not provide is speedups for serial code. *One of our main points is that the ICE/PRAM/XMT workflow and architecture provide a viable option for the many-core era*. Our solution should also inspire others to come up with competing abstraction proposals, or alternative architectures for ICE/PRAM. 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.

The workflow provides a productivity advantage to programmers. For example, we have traced several errors in XMTC programs of students to shortcuts they have taken around the ICE/algorithms stages. Overall, improved understanding of programmer's productivity, traditionally one of the hardest nuts for parallel computing, must become a top priority for architecture research. To the extent possible, evaluation of productivity should be on par with that of performance and power. For start, productivity benchmarks need to be developed. A suggestion for making teachability a useful benchmark follows.

Ease-of-programming (programmability) is a necessary condition for the success of a many-core platform and teachability is a necessary condition for programmability and in turn for productivity. The teachability of our approach has been extensively demonstrated. Over 100 students in grades K-12 have already programmed XMT and it even entered the regular syllabus of the year-long parallel computing course at Thomas Jefferson High-School for Science and Technology, Alexandria, VA [23]. Having gone through the effort of demonstrating teachability from middle-school and up, we

suggest that teachability at various levels becomes a standard benchmark for any many-core approach.

The recent paper [5] observed an interesting problem with current chipmakers' microprocessors, after analyzing current desktop/laptop applications for which better performance was desired. These applications tend to comprise many threads, but only very few of these threads 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 the inability of most programmers to handle current (and near-future) multicore microprocessors noted earlier. In contrast, guided by the simple ICE abstraction and the rich PRAM knowledge base to find parallelism, XMT programmers represent it using a type of threading that matches the XMT hardware.

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

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

[4] A. Balkan, M. Horak, G. Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Proc. Hot Interconnects*, Stanford, CA, 2007.

[5] G. Blake, R. Dreslinski, K. Flautner, and T. Mudge. Evolution of thread-level parallelism in desktop applications. In *Proc. ISCA*, 6 2010.

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

[7] G. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. gpu: Comparison of many-cores on irregular workloads. In *Proc. Usenix HotPar*, University of California, Berkeley, June 2010.

[8] G. Caragea, B. Saybasili, X. Wen, and U. Vishkin. Performance potential of an easy-to-program PRAM-On-Chip prototype versus state-of-the-art processor. In *Proc. ACM SPAA*, 2009.

[9] G. Caragea, A. Tzannes, F. Keceli, R. Barua, and U. Vishkin. Resource-aware compiler prefetching for many-cores. In *Proc. 9th Int. Symp. on Parallel and Distributed Computing (ISPDC)*, Istanbul, Turkey, July 2010.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Ed.* MIT Press, 2009.

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

[12] P. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous pram. *Theor. Comput. Sci.*, 196:3–29, 1998.

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

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

[15] L. Hochstein, V. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *J. Systems and Software*, 81, 2008.

[16] M. Horak, S. Nowick, M. Carlberg, and U. Vishkin. A low-overhead asynchronous interconnection network for gals chip multiprocessor. In *Proc. 4th ACM/IEEE International Symposium on Networks-on-Chip (NOCS2010)*, Grenoble, France, May 2010.

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

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

[19] J. Nuzman and U. Vishkin. Circuit architecture for reduced-synchrony on-chip interconnect. *U.S. Patent*, 6,768,336, 2004.

[20] D. Patterson. The trouble with multicore: Chipmakers are busy designing microprocessors that most programmers can't handle. *IEEE Spectrum*, July, 2010.

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

[22] H. Sutter. The free lunch is over - a fundamental shift towards concurrency in software. *Dr. Dobbs J.*, 2005.

[23] S. Torbert, U. Vishkin, R. Tzur, and D. Ellison. Is teaching parallel algorithmic thinking to high-school student possible? one teacher's experience. In *Proc. 41st ACM SIGCSE*, 2010.

[24] A. Tzannes, G. Caragea, R. Barua, and U. Vishkin. Lazy binary splitting: A run-time adaptive dynamic works-stealing scheduler. In *Proc. 15th ACM PPoPP*, 2010.

[25] L. Valiant. A bridging model for multi-core computing. In *Proc. Eur. Symp. Alg.*, 2008.

[26] U. Vishkin. Supporting patents. *U.S. Patents*, 6,463,527;6,542,918;7,505,822;7,523,293;7,707,388, 2002-2010.

[27] U. Vishkin. Algorithmic approach to designing an easy-to-program system: can it lead to a hw-enhanced programmer's workflow add-on? In *Proc. Int. Conf. Computer Design (ICCD)*, 2009.

[28] 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.

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