Easy PRAM-based high-performance parallel programming with ICE

Fady Ghanim  Uzi Vishkin  Rajeev Barua
Electrical and Computer Engineering Department
University of Maryland - College Park
MD, 20742, USA
{fghanim,barua,vishkin}@umd.edu

Abstract—Parallel machines have become more widely used. Unfortunately parallel programming technologies have advanced at a much slower pace except for regular programs. For irregular programs, this advancement is inhibited by high synchronization costs, non-loop parallelism, non-array data structures, recursively expressed parallelism and parallelism that is too fine-grained to be exploitable. We present ICE, a new parallel programming language that is easy-to-program, since: (i) ICE is a synchronous, lock-step language so there is no need for programmer-specified synchronization; (ii) for a PRAM algorithm its ICE program amounts to directly transcribing it; and (iii) the PRAM algorithmic theory offers unique wealth of parallel algorithms and techniques. We propose ICE to be a part of an ecosystem consisting of the XMT architecture, the PRAM algorithmic model, and ICE itself, that together deliver on the twin goal of easy programming and efficient parallelization of irregular programs. The XMT architecture, developed at UMD, can exploit fine-grained parallelism in irregular programs. We have built the ICE compiler which translates the ICE language into the multithreaded XMTC language; the significance of this is that multi-threading is a feature shared by practically all current scalable parallel programming languages thus providing a method to compile ICE code. As one indication of ease of programming, we observed a reduction in code size in 11 out of 16 benchmarks as compared to hand-optimized XMTC. For these programs, the average reduction in number of lines of code was 35.5%. The remaining 5 benchmarks had almost the same code size for both ICE and hand-optimized XMTC. Our main result is perhaps surprising: The run-time was comparable to XMTC with a 0.53% average gain for ICE across all benchmarks.

Index Terms—ICE, Ease of programming, irregular programs, PRAM, fine-grained parallelism, XMT, Nested ICE, Nested parallelism.

1 INTRODUCTION

Since 2005, practically all computers have become (multi-core) parallel machines. The field of parallel computing has made tremendous strides in exploiting parallelism for performance. However, it is also increasingly recognized that its trajectory is short of its general-purpose potential.

Parallel machines require partitioning the task at hand into subtasks (threads) to be run concurrently for minimizing: (i) memory accesses beyond local (cache) memories, and (ii) communication and synchronization among subtasks. Other programmers responsibilities include locking, which can be tricky for fine-grained multi-threading needed for scaling, work distribution and scheduling and handling concurrent access to data structures. While parallel programming languages and parallel machines differ on how much of the partitioning is the programmers responsibility, they all expect a significant effort from the programmer for producing an efficient multi-threaded program. Establishing correctness of these programs is yet another challenge, as asynchrony may increase the number of reachable states exponentially.

The theory of general-purpose parallel algorithms assumes an abstract computation model (known as PRAM for parallel random-access machine, or model) that stands in sharp contrast to these hardships; each time step involves a plurality of operations, all operation performed synchronously in unit time and may include access to a large shared memory. The PRAM computation model was developed in the 1980s and early 1990s, and abstracts away opportunities for using local memories, and minimizing computation or synchronization, locking, work distribution, scheduling and, in fact, any concept of threads. Also, for PRAM practically every problem has a parallel algorithm. This makes it both desirable and much easier to specify PRAM parallel algorithms, and the question that started out our work has been: but, at what performance penalty? As explained next, our surprising result is that it is feasible to avoid any performance penalty.

Coupled with prior work, our paper establishes the following result: (i) it is feasible to get competitive speed ups while essentially using PRAM algorithms as-is for programming a parallel computer system; furthermore (ii) these speedups are on par with multi-threaded code optimized to minimize non-local memory accesses, communication and synchronization. Establishing feasibility of using such abstract (and much simpler) PRAM programming whose performance is on par with the best manually optimized programs is a specific new contribution of the current paper.

Our prior work anticipated the above hardships. To preempt as many of them as we deemed feasible, our starting point for the design of a many-core architecture framework called XMT (short for eXplicit Multi-Threading) was PRAM. XMT made big strides toward overcoming claims by many that it would be impossible in practice to support effectively PRAM algorithms [e.g., [1]]. Its premise (in prior work) has been that it must be the programmer who will produce a multi-threaded program: [2] outlines a programmers workflow for advancing from a PRAM algorithm to an XMT multi-threaded program. Namely, the programmer is still responsible for producing a multi-threaded program with improved locality and reduced communication and synchronization. Hardware support that
XMT provides made this effort easier than for commercial machines, which paid off. This workflow allowed better speedups and demonstrated easier learning of parallel programming. Since our prior work remained wedded to programmer-provided multi-threading, it characterized XMT programming as PRAM-like, as opposed to just PRAM.

Our new work is fundamentally different. It shows for the first time that the threading-free synchronous parallel algorithms taught in PRAM textbooks can be used as-is for programming without performance penalty. Namely, it is feasible to reduce multi-threading to a compiler target, altogether freeing the cognition of the programmer from multithreading. In fact, we show that the programmer can essentially use the pseudo-code used in textbooks for describing synchronous parallel algorithms as-is; this elevates XMT from supporting PRAM-like programs to supporting PRAM programs. At the beginning of the XMT project: it was expected that the programmer will need to make an extra effort for explicating PRAM parallelism as multi-threaded parallelism; indeed, the name of XMT, explicit multi-threading, reflects the original expectation. As can be seen from the example, XMT gets us part of the way to fine-grained multi-threading, but not to lock-step PRAM programming.

ICE allows the same intuitive abstraction that made it easy to reason and program in serial. Namely, any instruction available for execution can execute immediately. In serial a program provides the instruction to be executed in the next time step. This made serial programs behave as rudimentary inductive steps from start of program to its final result. Similarly, ICE describes time-steps of serial or concurrent parallel instructions that execute in lock-step immediately at each time-step (inductively), while falling back to serial execution for serial portion of the code. In unifying serial and parallel code, ICE can be thought of as the natural extension of the serial model.

In this work we make the following contributions: 1. We enable the programmer to express the ICE abstraction directly using the new ICE programming model. 2. To enable this much higher-level programming, we propose a new compiler component that automatically translates the ICE program into an efficient XMT program. 3. The result is that we achieve comparable performance to a hand-written XMT program from an easy-to-program PRAM algorithm.

This paper proceeds as follows. Section 2 presents background information on the XMT architecture. Section 3 discusses the ICE language. Section 4 discusses the ICE compiler’s structure and translation method. In section 5, we present and discuss the results of our experiments. A review of related work is provided in Section 6 and section 7 is our conclusion.

2 BACKGROUND ON XMT ARCHITECTURE

We present in this section a very brief review of some basic concepts of the XMT framework to make this paper as self contained as possible. As space limitations prevent us from presenting a comprehensive discussion, we refer the reader to [3, 4, 5, 6].

To understand the XMT architecture, we first look at how it is programmed. The XMT high-level language is an extension of standard C detailed in [6]. A parallel region is delineated by `spawn` statement which initiates a specified number of virtual threads, and `join` statement which terminates them, as shown in figure 1(a). The virtual threads share and execute the same parallel code, and each is assigned a unique thread ID, designated $\$. The threads proceed with independent control and synchronize at the `join` statement. Synchronization can be achieved by the prefix-sum (`ps`) operation. The `ps` operation is an atomic fetch-and-add operation [7] that increments the base and return its original value. Figure 1(a) demonstrates its power by showing its usage in the prefix-sum unit.

![Fig. 1. XMT Programming. (a) Array Compaction example. Array A's non-zero elements are copied into B. The order is not necessarily preserved. After executing `ps(inc,base)`, the base variable is increased by `inc` and the `inc` variable gets the original value of `base`, as an atomic operation. (b) The XMT execution model: switching between serial and parallel modes.](image)

![Fig. 2. The left side of (b) shows the estimated latency to each memory hierarchy level from the processing core for a 1024 TCU configuration (64 clusters x 16 TCUs). Some elements are omitted for simplicity, such as the Master TCU, which operates in serial mode, the global register file and the prefix-sum unit.](image)
The XMT programming model allows programmers to specify an arbitrary degree of parallelism in their code. Clearly, real hardware has finite execution resources, so in general all threads cannot execute simultaneously. A hardware scheduler [9], allocates the individual virtual threads to the physical thread control units (TCU). It relies heavily on hardware support and the prefix-sum unit. Figure 2b gives an overview of the XMT memory hierarchy while operating in parallel mode. XMT designers chose not to deploy private caches in TCUs/ clusters due to the implementation complexities and power non-efficiency. Several techniques have been designed to reduce this latency, most notably prefetching customized for XMT [10].

We test and evaluate ICE using the XMT platform. The main reason is that unlike most other platforms XMT was designed in the first place to support PRAM algorithms and demonstrated to achieve unique speedups for irregular programs. Some examples of that are listed below. All speedups below were achieved over the best serial implementation on the state-of-the-art vendor’s platform; hence they represent real improvements in processing time.

- Graph Connectivity 1024-core XMT processor achieves a speedup of 99.8X, while the NVidia GTX480 had a speedup of 27.1X for graph connectivity [11].
- Graph Biconnectivity 1024-core XMT achieves speedups up to 33X, while GPU/ CPU hybrid achieved only a 4X speedup [11].
- Graph Triconnectivity 1024-core XMT got a speedup of 129X against serial on a core i7 920 processor [12].
- Finding maximum flow The best speed up for this algorithm on a hybrid NVidia Fermi GPU/ CPU was 2.5X [13]. In contrast, a speedup of 108X was attained on a 1000-core XMT that uses the same silicon area as the GPU [14].
- Burrows-Wheeler transform - BZIP2 XMT reaches up to 13X/ 25X Speedup for de/ compression [15]. In comparison, there was a slowdown of 2.8 for compression and a speedup of 1.1 for decompression on GPU.
- 2-D FFT XMT reached 20.4X speed up, whereas a 16-core AMD opteron got less than 4X [16].
- Gate-level Simulation Benchmark Suite XMT obtained 100X speedups versus serial for [17].

The XMT processor manages the creation, termination, and scheduling of threads dynamically and cheaply with no involvement of an operating system (OS) or other software. The XMT processor is programmed using the XMTC language, a parallel programming language based on C with modest extensions to take advantage of the special features provided by XMT. XMTC follows the fork-join threaded execution-model and provides similar set of features as other threaded languages currently used on commodity platforms and architectures.

3 THE ICE PROGRAMMING LANGUAGE

To see the features and advantages of the ICE programming model, consider the example in figure 3(a) which shows the problem specification for pointer jumping, a well-known, useful and widely used task in tree and graph algorithms. The example shows a specific assignment of weights which will compute the distance to the root in the output; however, any input assignment of weights can be chosen.

ICE follows the lock-step execution model and is based on the PRAM algorithmic model. A parallel region in ICE is specified inside the pardo construct. The pardo statement specifies lock-stepped parallel code in the statement body. However, XMTC language follows the threaded model, and uses the spawn construct to specify a parallel region. Figure 3(b) shows an ICE code to solve the pointer jumping problem defined in figure 3(a). An XMTC threaded version is shown in figure 3(c), and an OpenMP version in figure 3(d). Figure 4 provides the ICE syntax in Backus-Naur form. An example of usage is shown in figure 3(b). A pardo statement is a regular C statement and requires 4 components: 1- An init-expr which is thread identifier and first thread ID (unsigned i = 0), 2- a termn-expr termination thread id (n), 3- a step-expr step between consecutive threads (1), 4- and list of statements to be executed in lock-step. concurrent-write-statement is used to specify an assignment is a concurrent write. Independent-access-statement is used to specify that all statements in statement-list do not have any conflicts with each other. Both of these statement can only be used within a pardo block. Integer-constant and integer-identifier are constants and identifiers of integer-type. Table 1 provides a comparison between the syntax of the lock-stepped pardo and the threaded spawn. ICE and XMTC follow the same convention of having context/ thread local variables declared inside the parallel region, while shared variables are declared in serial regions.

From figure 3, we see that the ICE code is much shorter and simpler than both the XMTC and OpenMP codes. This is because the ICE lock-step model simplifies the expression of the in-update of place S and W. Hence, \((W(i) + W(S(i)))\) in the first statement is read and computed on all contexts, before any write is made to W(i) by any context. However, the unpredictability of the parallel threads pace in XMTC prohibits in-place updates of arrays S and W in Figure 3(c). Thus we must use temporaries S_temp and W_temp. Temporaries are used as an alternate to the actual arrays writing in the first part, and reading in the second. The \((ps)\) construct is used to count incomplete threads in the flag variable. The loop continues until all threads are done.

The OpenMP code in figure 3(d) essentially executes similarly to the XMTC version, and is equally long and complicated. However, there are two main differences. 1. The ps operation in XMTC version is replaced by a reduction operation in OpenMP. 2. Unlike the XMTC version, the loop was not unrolled in the OpenMP version. Instead, two sets of pointers were used to alternate the source and destination of copying between the original and temporary S and W arrays. It is important to understand that implementations in figures 3(c) and (d) are fully interchangeable between XMTC and OpenMP. Namely, the implementations will work very similarly regardless of the platform used. However, when implemented on a similar platform, the implementation in figure 3(c) will have a slight performance advantage over the implementation in figure 3(d), while

1. Although the code in figure 3 uses arrays to implement trees, pointer jumping can be implemented in ICE with structures and pointers just as easily. The code will be conceptually similar.

2. The ps operation could have been avoided by multiple writes of true to a boolean variable called threads-remaining in the loop, but that would create a hot-spot in memory. The XMTC ps operation uses registers, avoiding the hot spot.
Problem:
Given a linked list with n elements, find for every element its distance from the last element.

Input:
- Array S[0...n]: S(i) contains the index of the successor of element i. The successor of the last element is the element itself.
- W(0...n): W(i) contains the weight of element i. Initially W(i) = 0 for the last element in the list and W(i) = 1 for all other elements.

Output:
- S(i) is the index of the last element of the list.
- W(i) is the distance of element i from the last element.

(a) Problem specification

```c
void pointer_jump (int S[n], int W[n], int n) {  
pardo (unsigned i = 0; n; 1) {  
  while (S[i] != S[S[i]]) {  
    W[i] = W[S[i]] + W[S[S[i]]];  
    S[i] = S[S[i]];  
  }  
  S[i] = S[S[i]];  
  W[i] = W[S[i]] + W[S[S[i]]];  
}  
while (flag != 0) {  
  flag += x;  
  flag = 0;  
  x = 1;  
  for (int i = 0; i <= n; i++) {  
    W[i] = W[i] + W[i];  
  }  
  while (flag != 0) {  
    flag += x;  
  }  
  if (S[i] != S[S[i]]) {  
    W[i] = W[i] + W[S[i]];  
    S[i] = S[S[i]];  
  }  
  if (S[i] != S[S[i]]) {  
    W[i] = W[i] + W[S[i]];  
    S[i] = S[S[i]];  
  }  
}  

(b) ICE program
```

```
```

```c
void pointer_jump (int S[n], int W[n], int n) {  
  int S_temp[n], W_temp[n];  
  #pragma omp parallel num_threads(P) 
  ps(i, flag);  
  int W_rd[n], W_wt[n];  
  #pragma omp parallel for reduction(+ : flag) schedule(static, or size)  
  if (S[i] != S[S[i]]) {  
    int x = 1;  
    flag += x;  
    W_wt[i] = W_rd[i] + W_rd[S[S[i]]];  
    S_wt[i] = S_rd[S[S[i]]];  
  }  
  W_wt[i] = W_wt[i] + W_wt[S[S[i]]];  
  S_wt[i] = S_wt[i] + S[S[i]];  
}  
```

(c) XMTC program

```
```

(d) OpenMP Program

```
```

**Fig. 3. Pointer jumping example showing simplicity of ICE code.**

**Fig. 4. ICE language Syntax in Backus-Naur form.**

the later is slightly shorter and easier to write. Of course, an important takeaway is that both the XMTC and OpenMP codes, in figures 3(c) and 3(d) respectively, are considerably longer and more complex than the proposed ICE code in figure 3(b).

The above example in figure 3 shows many of the strengths of the ICE programming model, listed below:

- Easier translation from PRAM algorithms
  Unlike threaded model, PRAM algorithms readily fit into the ICE programming model. This is illustrated by the great difference between figures 3(b) and (c) - manually translating the first to the second can be a significant effort. Thus ICE makes parallel programming easier, fulfilling one of our primary goals.

- No need for thinking about synchronization or race conditions beyond what the PRAM algorithm specifies
  A programmer needs to decide when and where synchronization is required and what intermediate variable are needed to avoid race conditions, and be proactive in eliminating unintended race conditions. This task is a huge contributor to making parallel programming difficult, and requires special knowledge and experience. ICE assumes an implied barrier after every statement in a parallel region thus dealing with synchronization and make it impossible to have unintended race conditions. Thereafter the compiler manages race conditions and introduces any required intermediate temporaries to avoid them. ICE relieves the programmer from this heavy burden and makes parallel programming easier. This is demonstrated in figure 3(c), where the programmer has to decide the location of synchronization at the end of spawn blocks, introduce any needed ps operations in all the right places, and introduce the S_temp, W_temp, and flag intermediate variables to avoid race conditions resulting from the in-place update.

- No need to think about scheduling or coarsening
  While not the case in XMTC, several other threaded models in common use such as MPI and pthreads, require the programmer to manually schedule available parallelism.
into N threads and to coarsen if the available parallelism exceeds N^3. In contrast ICE is a declarative programming model where the programmer simply expresses all available parallelism without regard to the number of hardware contexts, or the scheduling of the code to those contexts. Scheduling and coarsening is performed automatically by the compiler and/or run-time system. This significantly reduces the burden on the programmer, and it also makes the code more portable across XMT computers with different numbers of hardware contexts.

Given the advantages above, we believe that ICE represents a significant leap in the ease of programming compared to threaded programming models. In addition, execution on hardware specialized in exploiting parallelism in irregular algorithms such as XMT, will deliver excellent speedups for irregular programs written in ICE.

Nested Parallelism in ICE. ICE allows programmers to specify nested parallelism in ICE by using the pardo keyword from within a spawn region. Each parallel context created by the outer pardo create multiple parallel contexts as specified by the inner pardo. All these child parallel contexts execute lock-step with one another across all parallel contexts of the same level. Variable locality for nested ICE follows the same principle that we discussed earlier, namely; variables declared inside an inner pardo are private, while variables declared outside are shared between the group of parallel contexts created by each individual context of the outer pardo. Nested ICE is translated into nested XMTC code by translating nested pardo regions into their equivalent nested spawn regions.

4 ICE TRANSLATION AND IMPLEMENTATION

In this work we translate programs written in ICE to the XMTC high level language. This requires maintaining correctness of the lock-step ICE program when translated to a threaded model. In this section we will discuss the challenges of such translation. We will also discuss our effort to deal with those challenges to ensure correctness. After that we will discuss the optimizations we made to maintain comparable performance to a highly-optimized hand-written XMTC code. Later, we will discuss the structure of our ICE compiler.

4.1 Translation

In this work we translate ICE programs to threaded XMTC programs using a new ICE compiler that we built. The output XMTC code is compiled using the existing relatively mature and well-studied XMTC compiler to executable XMT binary code. Due to the independence of threads, preserving correctness of lock-step code requires explicit synchronization between threads by the ICE compiler. In this subsection, we will discuss the steps we have taken to ensure the correct translation of ICE programs.

To translate ICE programs to XMTC programs, we split the pardo region into multiple spawn regions. Replacing every pardo with spawn will not work since the former requires lockstep execution, but the latter (regular multi-threading)

3. where N is the number of hardware contexts available on the target hardware. The number of hardware contexts is the number of threads that the hardware can actually run at any one instant. This equals the number of cores × the hyper-threading factor for multi-cores, and equals the number of TCUs on XMT.

| pardo (i = 0; n; 1) { if (i < 50) \ A[i+1] = c[i]; \ n = A[i] + 1; \ } | char cond[n=1]; 
| spawn(0,n) { unsigned i = $; \ if (i < 50) \ A[i+1] = c[i]; \ } | spawn(0,n) { unsigned i = $; \ if (cond[i]) \ c[i] = A[i] + 1; |

Fig. 5. (a) A pardo with a conditional branch. (b) Its XMTC translation.

does not ensure it. We saw this in figure 3. Splitting occurs at points where a barrier is required. In XMT there is no way to implement barriers except by using join. We introduce a join by terminating a spawn region and starting a new one, effectively splitting the pardo. This solution ensures that there will be no violation of the data dependencies (true or anti-dependence) between the memory accesses within the pardo region. This method's downside is that the parallelism granularity is reduced, but its degree is maintained.

To ensure correctness, the order of reads and writes must be maintained. Thus when translating ICE to XMTC, we need to split a pardo into multiple spawn blocks wherever the pardo contains both a read and a write to a data object accessed by at least two different parallel contexts. This ensures that a memory access is completed by all parallel contexts, before any context starts with the next memory access. This splitting is performed by introducing a barrier between the read and the write. Two cases are possible: anti-dependence where a write to a data object are done after a read (e.g. W and S in figure 3(b)), and true dependence where a read is performed after a write. Both cases require splitting the pardo region into two successive spawn regions. However, in the anti-dependence case, we also need to introduce a (compiler-inserted) temporary, which we perform the writes instead in the first spawn region, and copy them back in the second.

Correct translation of nested ICE code into nested XMTC code is similar to non nested ICE code in that it requires splitting the pardo region into multiple spawn regions. However, splitting an inner pardo region requires that we split all outer pardo regions containing it as well. Translating nested ICE code by only splitting the inner pardo region without splitting any of the outer pardo regions will create multiple spawn regions contained within one parent spawn block. Each parent thread created by the outer spawn will in turn execute its instance of the inner spawn calls at its own pace. So, a parent thread may potentially complete the execution of multiple inner spawn calls before any is executed by other parent threads. Thus, the parallel contexts created by a nested pardo will not synchronize with other nested parallel contexts on same level of nesting, thus breaking the lock-step execution semantics of ICE. Hence when an inner pardo region is split, the outer pardo containing it is split as well. Preserving and restoring the parallel context at pardo splits. Splitting a pardo region causes discontinuity in the execution of a pardo iteration. During execution, a pardo
split will result into terminating a group of threads as they reach the synchronization point, and spawning a new group of threads that will continue the execution starting anew after the synchronization point. This will cause an interruption to the data and control flow of the iteration resulting in an incorrect execution. So if the ICE program is to execute correctly, the new thread corresponding to a pardo iteration must be able to continue execution from where the old thread corresponding to the same iteration left off. As such a mechanism is needed to maintain the parallel context for each pardo iteration before and after a split.

Splitting pardo regions may cause complications for the program’s data and control flow. For data, it is sufficient to keep a copy of the registers holding the intermediate data still needed. However the situation for preserving the control flow is slightly more complicated. There are two cases when a split to a pardo region requires special attention: (1) When a pardo region contains a conditional branch where one of its directions requires a barrier as in figure 5. (2) When a pardo region contains a serial loop within which a barrier is needed. This causes a problem when expressing the continue and break statements, and the serial loop’s back edge as in figure 3(b).

To maintain correctness, a parallel context must preserve its intended control flow, which is not easily possible in these cases since XMT disallows branching between spawn blocks.

To maintain control flow, we communicate branch decisions across splits by recording the branch state for each context into memory, and retrieve it when needed. Hence, for the first case when a branch condition is evaluated as in figure 5(b), we record the result to memory (temporary array cond) and retrieve it in any later spawn that is on either branch direction. A similar solution is used for the second case where the serial loop is taken outside the parallel region and is executed by the MTCU, the loop condition becomes a flag indicative of the existence of threads that are not done executing yet, and the original loop termination condition becomes a normal branch and is treated as in the branch case. An example of this is the do-while loop in figure 3(c) where the serial loop is taken outside the spawn block, the terminating condition now is (flag = 0) instead of (S(i) == S(S(i))). flag is incremented by threads which still have work to do, using the ps operation (explained earlier in figure 1).

We use temporary arrays to record when a context executes a continue or break (no example shown). Resultant spawn blocks from such a loop split will check the temporary arrays to see if the context have executed either a continue or break, and will similarly execute a continue or break. In case of splitting nested pardo regions, temporaries will need to be created to communicate the control direction for each level of nesting, since a split within the inner pardo requires that we split its parent pardo regions as well.

4.2 Optimization of the translated code

Splitting a pardo into multiple spawns can degrade performance, due to the overhead of creating and managing more threads. Also, using memory to communicate information between spawns increases the degradation even further. This is exacerbated when the number of splits is high, or a split happens in a deeply nested pardo region. Hence it is crucial to avoid splitting whenever possible, and to mitigate the effects of the unavoidable splits.

**Fig. 6. Rescheduling memory accesses.**

Splitting a pardo can be avoided if we can prove that a memory location is exclusively accessed by a certain parallel context only. In this case, the splitting becomes unnecessary and a direct conversion from a pardo to a spawn will work. One example of this is when a parallel context with ID ‘i’ always reads and writes to A[i]; hence we know that no two contexts access the same memory location. This means that no race conditions are possible; hence no splitting is needed. Optimization for anti-dependence case within loops in pardo When the anti-dependence is within a loop in a pardo (as in figure 3 example), we can get better performance by unrolling the pardo once, and then transforming the two loops, that result so that the first loop updates temporary data structures that are clones of the original data structures, and the second loop does the opposite. An example of this is seen in figure 3(c). Thereafter the pardo is split to place the two loops in different spawn blocks in the XMT output. Other elements in the figure such as ps operation and ‘flag’ will be discussed in detail shortly.

Clustering In an optimization for unavoidable splits, we rearrange memory accesses within a pardo into clusters to minimize the number of splits needed. Each cluster represents a spawn block. These clusters consist of a group of memory accesses that are independent from one another across the different parallel contexts. When a pardo region is split into multiple spawns, often there are more splits than necessary. We see an example of this in figure 6(a), where there is a dependence between statements A1 and A2, and another between B1 and B2, but none exist between the A and B statements. Without optimization we will end up with three spawns after the splitting as in figure 6(b). However, by rearranging and grouping independent memory accesses as in figure 6(c) and only then doing the splitting, we end up with two spawns. We call this rescheduling scheme clustering.

The clustering algorithm is a list scheduling algorithm. Figure 7 shows the algorithm used. We build a dependence graph in which we capture all data (flow or ‘loop-carried’4) and control dependencies between all the memory accesses. Then we start building one cluster at a time by scheduling all ‘ready-to-fire’ nodes in greating order (lines 28 - 34). A node is ‘ready-to-fire’ if it satisfies the conditions in the lines...
potentially distribute instructions of a basic block across two or more clusters. This reordering causes a major problem when splitting serial loops, since it prevents the transformation of a serial loop within a pardo region, discussed in subsection 4.1 above, in which a split serial loop within a pardo block is replaced by a serial loop outside the resulting spawn blocks. This is because after clustering, the instructions belonging to that serial loop are likely to get mixed with instructions from other basic blocks that are not part of the serial loop.

We solve this problem by creating an empty replica of the Control Flow Graph (CFG) of the pardo region in all spawn blocks that were generated from it. As such, every basic block inside the pardo will have a copy of it inside every resulting spawn blocks. This allows us to maintain the correctness of the control flow more easily, and allows a direct and uncomplicated placement of the memory accesses in their respective spawn blocks. Basically, a memory access is simply moved from the original parent basic block inside the pardo block, to the parent block’s replica inside the spawn block where it belongs. Furthermore, we can still use memory to communicate control direction as discussed in subsection 4.1 above; however it now must be performed in every spawn block.

There are two exceptions where a basic block is not replicated: 1. If the basic block is a target of a conditional branch whose condition cannot be calculated at that stage yet because it depends on a memory access(es) that occur at a later spawn block. As long as the basic block replicated belongs to such a spawn block, the conditional branch will be replaced with a direct branch to the common immediate post-dominator basic block of the conditional branch’s targets. 2. If the basic block belongs to a serial loop inside a pardo block. Since, as was discussed in subsection 4.1, we achieve the back edge of the loop by creating a serial loop outside the spawn blocks and replace the loop with branches inside of it, the basic blocks from the loop cannot exist along basic blocks from outside it, since that means that these other basic blocks will execute every time the loop is executed. Instead, during clustering we make sure that a cluster is not shared between multiple loops (lines 17 - 18 of figure 7). As such, a split serial loop will be clustered into a set of consecutive spawn blocks.

4.3 The ICE compiler structure

The ICE compiler uses a modified clang frontend and the LLVM compiler infrastructure to perform source-to-source translation of ICE code into XMT code. Thereafter the XMT code is compiled using the existing gcc-based XMT compiler [6]. We modified Clang by adding the ‘pardo’ keyword, and implemented the parsing of the pardo and the relevant IR code generation. We have also implemented multiple LLVM passes to accomplish all the various steps required to convert the lock-step semantics into threaded code. Figure 8 outlines the algorithm for translating from ICE to XMT.

The LLVM compiler stack is designed for serial threaded code executed by a single processor, making it incompatible with lock-stepped parallel code. Since the available compiler transformations do not take into account many of the properties of parallel code (e.g. differentiating between shared vs local variables or serial vs parallel contexts), we took certain steps to maintain the correctness of the ICE code when using native LLVM passes. For example, we mark the beginning and
In this section we present the results of our experiments comparing ICE language to XMTC. We first look at the difference in ease of programming between ICE and XMTC by showing a comparison of the number of code lines needed to write the same algorithms. Then, we look at the translation accuracy, by comparing the ICE to XMTC translation, to the hand-optimized XMTC in terms of the number of spawn blocks and temporaries used. Finally, we provide performance comparison results between XMTC and ICE for our benchmarks.

Since ICE is a new language with no standardized benchmarks, we developed a suite of 16 benchmarks based on common PRAM algorithms to use for our experiments. This benchmark suite contains benchmarks for both nested and non-nested algorithms. For each benchmark, a pseudo-code was written, then based on that pseudo-code we implemented two versions: an XMTC version that is manually optimized for best performance, and the ICE version. We compile the ICE versions using our ICE compiler, then the automatic output XMTC code is compiled using the XMTC compiler. We use the same XMTC compiler for compiling both the XMTC code and the automatically generated XMTC code from ICE. We include a list of our benchmarks in Table 2. Due to space constraints, we refer the reader to [18, 19, 20] for a detailed description of each of these algorithms.

5.1 Ease of use and Code size

In this section, we will look at the code sizes of all benchmarks ICE and XMTC implementations. We use code size as a measure of ease of programming. This is fair because ICE and XMTC are extensions of the C language, each featuring an extra keyword for lock-step in ICE, and spawn for threads in XMTC. Both languages are identical otherwise. This means that for the same pseudo-code of an algorithm with same inputs and outputs, the increase in code size indicates more elaboration was needed to ensure correctness and/or higher performance as can be seen in the example in Figure 3. Thus, we believe comparing lines of code to approximate ease of programming is a valid approach to demonstrate the ease of programming of ICE compared to XMTC.

We provide the two different measurement of code size: a measurement for the entire program, and a measurement for the parallel algorithmic part. For both measures, we declared each variable on a separate line. For the algorithmic parallel portion of the code, we measure only the benchmark’s declared each variable on a separate line. For the algorithmic parallel portion of the code, we measure only the benchmark’s code size for parallel sections only, excluding all shared variable declarations and non-recurring initializations, all serial algorithms used as part of the main parallel algorithm (i.e., serial sorting or summation, etc.), the main function, and all preprocess directives.

Now we look at Figure 9 where we see a comparison of the reduction in the entire program size for non-nested ICE programs normalized to optimized XMTC. This graph shows that ICE has a smaller code size when compared to XMTC for seven out of our eleven benchmarks. The other four
### TABLE 2

Benchmarks List. For benchmarks marked with an *, we used The pseudo and optimized XMTC codes that were predeveloped by the XMT/XMTC platform designers. We only implemented the ICE version.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>XMTC Code Size</th>
<th>Problem Size</th>
<th>Abrev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Sort*</td>
<td>56</td>
<td>1048576</td>
<td>INT</td>
</tr>
<tr>
<td>Merging*</td>
<td>122</td>
<td>1000000</td>
<td>MRG</td>
</tr>
<tr>
<td>Sample Sort*</td>
<td>140</td>
<td>131072</td>
<td>SMP</td>
</tr>
<tr>
<td>Breadth First Search*</td>
<td>53</td>
<td>32768 nodes, 65536 edges</td>
<td>BFS</td>
</tr>
<tr>
<td>Breadth First Search (nested)*</td>
<td>55</td>
<td>32768 nodes, 65536 edges</td>
<td>NBFS</td>
</tr>
<tr>
<td>Graph Connectivity*</td>
<td>108</td>
<td>32768 nodes, 65536 edges</td>
<td>CVTY</td>
</tr>
<tr>
<td>Maximum Finding</td>
<td>57</td>
<td>262144</td>
<td>MAX</td>
</tr>
<tr>
<td>Tree Contraction</td>
<td>112</td>
<td>32768 nodes</td>
<td>CTRC</td>
</tr>
<tr>
<td>Tree Rooting*</td>
<td>90</td>
<td>32768 nodes, 65536 edges</td>
<td>RANK</td>
</tr>
<tr>
<td>2D Jacobi Stencil Computation (flattened)</td>
<td>60</td>
<td>512x512</td>
<td>JAC</td>
</tr>
<tr>
<td>2D Jacobi Stencil Computation (nested)</td>
<td>49</td>
<td>512x512</td>
<td>NJAC</td>
</tr>
<tr>
<td>LU Factorization (flattened)</td>
<td>56</td>
<td>512x512</td>
<td>LU</td>
</tr>
<tr>
<td>LU Factorization (nested)</td>
<td>42</td>
<td>512x512</td>
<td>NLU</td>
</tr>
<tr>
<td>Cholesky Factorization (flattened)</td>
<td>72</td>
<td>512x512</td>
<td>CHO</td>
</tr>
<tr>
<td>Cholesky Factorization (nested)</td>
<td>60</td>
<td>512x512</td>
<td>CHO</td>
</tr>
<tr>
<td>Topological Sort (nested)</td>
<td>64</td>
<td>32768 nodes, 65536 edges</td>
<td>TOBO</td>
</tr>
</tbody>
</table>

benchmarks saw no reduction in code size, since they contain none of the cases that ICE can help programmers with. These benchmarks were included only as a base-line case. ICE provides an average code size reduction of 11.01% for the entire set, and 16.08% for the benchmarks that showed an improvement.

Figure 10 shows the percentage of code size reduction for the parallel algorithm part of the benchmark for non-nested ICE programs when normalized to the XMTC version. We notice that here as well, ICE provides the largest reduction in code size when compared to XMTC with reduction of up to 57.14% in some cases. ICE provides an average reduction of 21.61% for the entire set, and 33.35% for benchmarks that showed an improvement. This shows the potential of ICE to reduce code size (and therefore programming effort) compared to XMTC, which is a more traditional threaded language.

We also notice that the ease of programming benefit of ICE extends to nested ICE as well, as can be seen in figures 11 and 12. Figure 11 provides a comparison of the reduction in size of the entire program code for nested ICE normalized to optimized nested XMTC, while figure 12 shows the percentage of code size reduction for the parallel algorithm part of the benchmark for nested ICE when normalized to the nested XMTC version. We notice in both figures that ICE provides an average reduction in code size of 12.28% for the entire program, and 34.14% for the parallel algorithm portion of the code. We also notice in figure 12 that the maximum reduction in code size for the algorithm portion of the code was 64.71%. For all 16 benchmarks taken together, ICE provides an average reduction in code size of 11.72% for the entire program, and 25.53% for the parallel algorithm portion of the code.

#### 5.2 Accuracy

In this section we take a look at the ICE compiler’s accuracy and effectiveness in translating to XMTC. We look at the number of spawn blocks and temporaries used to implement our benchmarks. We believe that this will help demonstrate the ICE compiler’s effectiveness in producing high performance XMTC programs, due to the effect of spawn blocks and temporaries on the runtime performance of the translated XMTC code as discussed in section 4.2.

5. Each temporary was used to store only one value that may be read multiple times.
In this section, we will focus on the performance comparison between ICE and XMT. We use the XMT FPGA which has 64 TCUs to measure the performance for both the XMT and ICE versions of the same algorithm pseudo-code. Figures 13 and 14 provides the speedup of ICE normalized to hand-optimized XMT for non-nested and nested programs, respectively. The speedup is 0.91% on average for non-nested benchmarks, 0.76% on average for nested benchmarks, and has negligible effect on the validity of our comparison. We use the XMT FPGA taken when the XMT binaries are run on the XMT FPGA. We provide the performance results for the ICE code normalized to hand-optimized XMT programs.

We have taken steps to ensure that ICE is being compared to the fastest hand-optimized XMT. Since memory accesses are the biggest source of overhead in XMT, we did not use temporaries in XMT programs unless it was necessary. This is shown in table 3 where eleven benchmarks use no temporaries and fifteen use two temporaries or less. The other lesser source of overhead comes from the creation and termination of threads. This overhead is very small in XMT and have negligible effect on the validity of our comparison.

ICE achieves comparable performance to hand-optimized XMT, which takes considerably more programming effort to write than ICE. We see in figure 15 that ICE has a 0.76% speedup on average for non-nested benchmarks, with maximum slowdown of 2.5% when compared to the performance of optimized XMT. Figure 16, shows that ICE for nested programs has the same run-time on average as hand-optimized XMT, with a maximum slowdown of 0.91%. We believe such negligible performance penalties for

---

**Table 3**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Hand-written XMT</th>
<th>Generated XMT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>spawns</td>
<td>Temp.</td>
</tr>
<tr>
<td>Integer Sort</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Merging</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Sample Sort</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Breadth First Search</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Breadth First Search (nested)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Graph Connectivity</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Finding</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Tree Contraction</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Tree Rooting</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Jacobi (nested)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>LU Factorization</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>LU Factorization (nested)</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky Factorization</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Cholesky Factorization (nested)</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Topological Sort</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

We look at table 3 to see the number of spawn blocks and temporaries used by the programmer and the ICE compiler. This table shows that 15 out of the 16 benchmarks had the same number of spawns and temporaries in both XMT versions. For the single benchmark where the auto-generated XMT had more spawns and temporaries compared to hand-written XMT. This benchmark had multiple independent indirect memory references that cannot be detected by compilers. However, the programmer for the hand-written version was able to avoid the extra splits and temporaries.

The ability of the ICE compiler to generate high performance code is strongly dependent on the performance of the alias analysis used to determine the dependencies between memory accesses. These dependency relationships are used during the clustering step to determine the number of resultant spawn blocks as was discussed in section 4.2. Whenever uncertain about a dependency, the compiler conservatively assumes a dependence exists anyway. This means that whenever alias analysis provide definitive no-alias answers about memory references, the clustering algorithm makes better clustering decisions. Alias analysis is a large field of compiler theory research and any advancements in it will benefit ICE. However, it is outside the scope of this work and we will not discuss it any further.

---

**Fig. 12.** Code size of the algorithm's parallel sections normalized to XMT for nested benchmarks.

**Fig. 13.** 64 TCU XMT processor speedup comparison.
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2758375, IEEE Transactions on Parallel and Distributed Systems

**Fig. 14. 64 TCU XMT processor speedup comparison for nested ICE**

![Image of graph comparing speedup for nested and non-nested ICE]  

**Fig. 15. 64 TCU XMT net speedup of ICE normalized to optimized XMTC with a 95% confidence interval**

![Image of graph showing net speedup comparison]  

**Fig. 16. 64 TCU XMT net speedup of ICE normalized to optimized XMTC for the nested benchmarks with a 95% confidence interval**

![Image of graph showing net speedup for nested benchmarks]  

**Fig. 17. 64 TCU XMT net speedup comparison between nested and non-nested ICE normalized to optimized XMTC**

![Image of graph comparing nested and non-nested ICE speedup]  

ups over XMTC for expert programmers, since intuitively hand-optimized parallel code should always be faster. Upon investigating, we found that there are multiple factors contributing to the observed speedups. For some benchmarks (MRG, MAX, JAC), the ICE code was accurately translated to its equivalent XMTC code (i.e., it has the same number of spawn blocks and temporaries). However, the program layout of both versions is different. This suggests that the performance gain is a result of factors unrelated to the translation such as data location in the read-only cache, instruction scheduling, the data pre-fetched, or the optimizations recognized by the XMTC compiler. For another benchmark subset (BFS, CTRC), the performance gain was in part a result of the used LLVM compiler’s native optimizations which are more recent than the GCC compiler used in XMTC implementation. This is combined with the ICE compiler specific optimizations that we implemented. When a PRAM algorithm requires multiple synchronization points within a deep nested if-else block, the condition needs to be re-evaluated after each point. The ICE compiler use of bit vectors to record the evaluation results for multiple branches means a single memory read per a spawn block will be sufficient as was discussed in section 4.2. Since a programmer is very unlikely to use bit vectors to record results of multiple branches, multiple reads per spawn block are needed for condition evaluation.

To compare the performance of ICE for both the nested and non-nested cases of same algorithms, we look at figure 17 where we see a comparison of both nested and the non-nested net speedups as compared to hand-optimized XMTC. We notice that for three of four benchmarks, the nested version achieved slightly better speedups compared to the non-nested version, whereas for the fourth benchmark, the nested version achieved significantly lower performance when compared to its non-nested counterpart. We believe that this was mainly due to the minor changes made to the algorithm to be able to write a non-nested version of it. We do not think that we can make a conclusion on which method is better based on such a small subset.

We validated the scalability of our results by running on a subset of our benchmarks on the XMT cycle accurate simulator [21] using a 1024 TCU configuration. As can be seen in figure 18 the results are quite similar to the 64-TCU FPGA results.
The ease of programming of ICE allowed us to write programs directly from a parallel (PRAM) algorithm with effort less than that of non-optimized XMTC, and gain performance comparable to hand optimized XMTC through automating the process of optimizing the code.

6 RELATED WORK

There are hundreds of parallel languages – Michael Wrinn from Intel listed over 225 parallel languages in his SIGCS 2010 keynote address, and it is impractical to discuss them all here. We will focus on languages that are most closely related, either for having an algorithmic foundation, such as PRAM, or have an ICE-like lock-step execution model, or are meant for XMT like hardware suited for irregular programs. In summary, we have not found any related work that has the entire ICE ecosystem of easy to program language, based on a rich algorithmic theory (i.e., PRAM), a capable compiler mapping to threaded programs, and a hardware capable of exploiting fine-grained irregular parallel programs.

Our goal here is to allow programmers to use - as freely as possible - an extended form of lock-step programming similar to the way parallel algorithms are expressed in the PRAM literature. We call this extended form ICE programming. Additionally we show how to map the ICE lock-step semantics onto multithreaded semantics such as XMT’s while achieving the best performance we can. This performance objective entails reducing the lock-step specification synchrony automatically.

So far, XMT programming of PRAM algorithms was done using the modest XMTC extension to C [22] suggests a “programmer’s workflow” guiding the programmer on advancing an algorithm ICE abstraction to an XMT program and fine tuning its performance. The XMT hardware achieves strong speedups over serial algorithm for many parallel algorithms implemented using this workflow [22]. This work seeks to significantly reduce the algorithm-to-computer-program effort by the programmer. A programmer will encode an algorithm specification in ICE instead of programming in XMT. The ICE implementation should be “on par” in performance with hand-optimized XMTC code.

DARPA launched the HIGH Productivity Computing Systems (HPCS) program with the purpose of building systems that can be programmed productively. It resulted into three languages; Cray’s CHAPEL [23], SUN’s Fortress [24], and IBM’s X10 [25]. Although all these languages have ease of programming and high productivity as a goal, none is suited for the lock-step model of PRAM algorithms. Further all these languages require manual specification of synchrony and concurrency, whereas the ICE compiler automates the process. Finally, these languages are intended to be mapped to traditional coarse-grained hardware; hence they perform poorly on irregular programs when compared to XMT.

APL is an early example of high-level programming that allows for lock-step parallelism. A series of papers that appears to have culminated with [26] sought execution of compiler-extracted parallelism from APL programs on the IBM RP3. However, APL did not provide sufficient support for the PRAM parallel algorithms literature. The V-RAM [27] appears to be the first lock-step programming model aimed at implementing this literature. However, it was a lock-step model targeting vector hardware. NESL that followed was not lock-step, but, still appears to have targeted machine models for which synchronization was relatively easy; see, e.g., [28]. In any case, we are unaware of speedup results for these approaches (APL, V-RAM, NESL, etc.) that approach XMT results, especially for irregular applications.

The case for (lock-step, nested) ICE programming Blelloch [27][29] examined parallel algorithms and found that nearly all are parallel operations over collections of values, called data-parallelism by Hillis and Steele [30]. The languages based on it are referred to as data-parallel languages (e.g. [31, 32, 33, 34]). Also, Blelloch contrasted flat data-parallel languages with nested data-parallel languages. Blelloch claimed that the ability to nest parallel calls is critical for expressing algorithms in a way that matches our high-level intuition of how they work. We concur.

As the multi-threaded architectures gained popularity, the need for nesting, encouraged by Blelloch’s work, gained momentum. Cilk [35] is a good example of such general multi-threaded programming. Multi-threaded architectures allowed greater implementation flexibility than flat real (vector-like) machines. Cilk contributed important compiler and run-time techniques such as work-stealing for implementation of nested parallelism. [36] further optimized work stealing to an improvement called Lazy Binary Splitting (LBS). Cilk++ [37] has incorporated a concept of reducers that can be supported by their scheduler without incurring significant overhead.

Unlike Cilk, ICE avoids the synchrony and concurrency problems that hindered the productivity in general multi-threaded programming. ICE also directly connects with parallel algorithms literature solving the original problem that nested data-parallelism addressed, and helps reduce programmer effort much further than both of XMTC and consequently Cilk. Further ICE equips programmers with more freedom for designing for WD performance, as evident from the comparison of the multi-threaded algorithms section in [38], to parallel algorithms texts [18, 19, 20] and demonstrated by the merging algorithm in [2]. However, Cilk

6. called high-level work-depth (HLWD)
is more accommodating to programmers than its immediate competition and has an important advantage of being supported by commodity hardware, but which cannot exploit irregular parallelism as effectively as XMT.

Our central question is: How should the programming of parallel machines be? We believe there is a considerable intellectual and practical merit in advancing programming specification that unleashes the wealth of parallel algorithms in the literature. This merit is suggested by the fact that while the technology and parallel architectures changed over time, these algorithms remained resilient to change in spite of the vigorous attempts by numerous researchers. Hence, we believe that this programming specification should be simple to produce, as close to the original parallel algorithm as possible, and is efficiently implementable on some architecture platform. This will guide future parallel architectures through benchmarks implemented based on these specifications. However, the success of XMT on ease of programming suggests that support of parallel algorithms theory, and its concept of parallel algorithmic thinking is as important to parallel systems designs as any set of specific applications or features. This is also the biggest departure from standard computer architecture practice.

7 CONCLUSION

We present ICE, a new lock-step easy-to-program parallel programming language based on the PRAM algorithmic model. We present the ICE compiler that we developed which translates the lock-step ICE programs into a traditional threaded XMTC programs. We demonstrate that the ICE compiler can provide comparable performance to highly-optimized XMTC programs while requiring much less effort from the programmer. We show how ICE easiness-to-program works in synergy with XMT’s efficient parallelization of irregular programs to strike the ever-sought balance between the compiler and the programmer roles in producing parallel programs, where the programmer needs only to specify parallelism and rely on the compiler to do the rest. Finally, given the relatively slow progress in parallel programming language technologies for irregular programs, our works suggests new opportunities for benchmarking parallel machines by their efficient support of high-level parallel algorithmic languages.

We conclude with a broader perspective on the significance of our contribution. It should be clear that ICE (or work-depth) parallelism exists in every serial algorithm. The only effort needed when we wish to use parallelism inherent in a serial algorithm is to express it, which in our experience is just a matter of skill, with no creativity involved. In contrast, practically all commercial approaches to parallel programming are based on partitioning the work to be done among processors or threads. There is no clear path for deriving that from a serial algorithm, and, when doable, requires significant creativity; in fact, in many cases it either cannot be done or cannot be done beyond very limited levels of parallelism. This extra level of creativity raises the bar on the skill and effort of programmers, and has greatly limited the adoption of many cores among programmers and application software vendors. Our paper, along with prior XMT work, establishes that there is a way to avert the above practice, which arguably amounts to throwing the parallel programmer under the bus, through proper hardware and software design choices.

REFERENCES

This work was supported in part by the National Science Foundation grant CNS1161857.

REFERENCES


Dr. Uzi Vishkin (SM’98) received the B.S. and M.S. degrees in mathematics from Hebrew University, Jerusalem, Israel, and the D.Sc. degree in computer science from Technion, Israel Institute of Technology, Haifa, Israel, in 1981. He is a permanent member of the University of Maryland Institute for Advanced Computer Studies, College Park, and has been a Professor of electrical and computer engineering with the University of Maryland, College Park, since 1988. He was a Professor of computer science with Technion from 2000 to 2002, and a Professor of computer science at the University of Maryland, College Park, since 1988. Prior to that, he was a Research Faculty Member with the Courant Institute, New York University, New York, NY, and a Post-Doctoral Fellow with the IBM T. J. Watson Research Center, Yorktown Heights, NY. He has authored or co-authored 190 publications, including nine patents. One of the founders of the field of parallel algorithms and the inventor of the explicit multithreading architecture, his primary area of interest has been parallel computing in general and parallel algorithms in particular. Dr. Vishkin is a Fellow of ACM.