Easy PRAM-based high-performance parallel programming with ICE[*]

Fady Ghanim[1], Uzi Vishkin[1,2], and Rajeev Barua[1]

[1]Electrical and Computer Engineering Department
[2]University of Maryland Institute for Advance Computer Studies
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; (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 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. Our main result is perhaps surprising: The run-time was comparable to XMTC with a 0.48% average gain for ICE across all benchmarks. Also, as an indication of ease of programming, we observed a reduction in code size in 7 out of 11 benchmarks vs. XMTC. For these programs, the average reduction in number of lines of code was 35.5% when compared to hand optimized XMTC The remaining 4 benchmarks had the same code size.

[*]This work was partially supported by NSF award 1161857.
A poster version of this paper appears in PACT 2016. [17]

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. This PRAM computation model 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 speedups 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 was the rich theory of parallel algorithms, known as PRAM (for parallel random-access machine or model) developed in the 1980s and early 1990s. XMT made big strides toward overcoming claims by many that it would be impossible in practice to support effectively PRAM algorithms [e.g., [11]]. Its premise (in prior work) has been that it must be the programmer who will produce a multi-threaded program: [32] 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 algorithm as-is; this elevates XMT from supporting PRAM-like programs to supporting PRAM programs. Note that the new result surprised even ourselves, exceeding our own expectations at the beginning of the XMT project: we 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 our 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 instructions 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 immediately 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 XMTC program. 3. The end result is that we achieve comparable performance to a hand-written XMTC 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

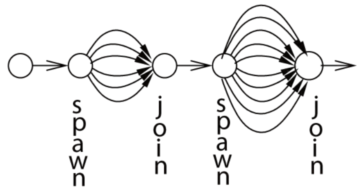
We present in this section a very brief review of some basic concepts of the XMT framework to make this paper as self contained as possible. As space limitations prevent us from presenting a comprehensive discussion, we refer the reader to [2], [26], [36], [37].

To understand the XMT architecture, we first look at how it is programmed. The **XMTC high-level language** is an extension of standard C detailed in [2]. 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 [18] that increments the base and return its original value. Figure 1(a) demonstrates its power by showing its usage to assign a unique index in array B when compacting an array A. Similar to PRAM algorithms, the XMT framework uses an arbitrary CRCW (concurrent read concurrent write) SPMD (single program multiple data) programming model. Concurrent writes to the same memory location result in an arbitrary one committing. An algorithm doesn't need to make assumptions about who will succeed, thus allowing threads to progress at their own pace independently from the others. See Figure 1.

The **XMT processor**, shown in Figure 2a, implements the above programming model efficiently. It includes many components but most relevant to this work are the master thread control unit (MTCU), processing clusters (C0...Cn) each comprising several thread control units (TCUs), and the prefix-sum unit. The MTCU has a standard private data cache, used only in serial mode, and a standard instruction cache. It shares the memory modules (MM0 .. MMm) with all the TCUs. The prefix sum unit executes `ps` operation very efficiently. Its hardware implementation [34][35] allows for an execution time independent from the number of requesting TCUs, thus allowing efficient and scalable inter-thread ordering and synchronization.

```
int A[N],B[N], base=0;
spawn(0,N-1) {
  int inc=1;
  if (A[$]!=0) {
    ps(inc,base);
    B[inc]=A[$];
  } join
}
```

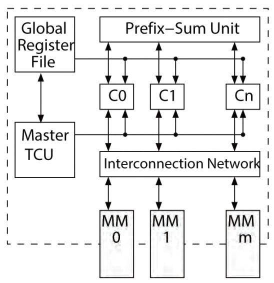
(a)



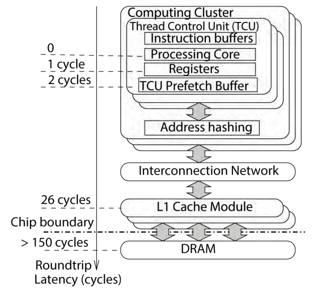
(b)

Figure 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.

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** [35], 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 [7].



(a) Block diagram.



(b) Memory Hierarchy in parallel mode.

Figure 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 × 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.

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 [12].
- **Graph Biconnectivity** 1024-core XMT achieves speedups up to 33X, while GPU/CPU hybrid achieved only a 4X speedup [12].
- **Graph Triconnectivity** 1024-core XMT got a speedup of 129X against serial on a core i7 920 processor [13].
- **Finding maximum flow** The best speed up for this algorithm on a hybrid NVedia Fermi GPU/CPU was 2.5X [20]. In contrast, a speedup of 108X was attained on a 1000-core XMT that uses the same silicon area as the GPU [6].
- **Burrows-Wheeler transform - BZIP2** XMT reaches up to 13X/25X Speedup for de/compression [14]. 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 **STBV09**
- **Gate-level Simulation Benchmark Suite** XMT obtained 100X speedups versus serial for [19].

The XMT processor manages the creation, termination, and scheduling of threads dynamically and cheaply with no involvement of an operating systems (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.

Problem:

Given a linked list with n elements, find for every elements its distance from the last element.

Input:

- *Array S(1...n): S(i)* contains the index of the successor of element i. The successor of the last element is the element itself.
- W(1…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 this last element.*

(a) Problem specification

```
pardo (unsigned i = 0; n-1;1) {
    while (S[i] != S[S[i]]) {
        W[i] = W[i] + W[S[i]];
        S[i] = S[S[i]];
    }
}
```

(b) ICE program

```
psBaseReg flag;  // number of threads that require
                    another loop iteration
void pointer_jump (int S[n], int W[n], int n) {
    int W_tmp[n];
    int S_tmp[n];
    do {
        spawn (0, n-1) {
            if (S[$] != S[S[$]]) {
                W_tmp[$] = W[$] + W[S[$]];
                S_tmp[$] = S[S[$]];
            } else {
                W_tmp[$] = W[$];
                S_tmp[$] = S[$];
            }
        }
        flag = 0;
        spawn (0, n-1) {
            if (S_tmp[$] != S_tmp[S_tmp[$]]) {
                int i = 1;
                ps(i, flag);
                W[$] = W_tmp[$] + W_tmp[S_tmp[$]];
                S[$] = S_tmp[S_tmp[$]];
            } else {
                W[$] = W_tmp[$];
                S[$] = S_tmp[$];
            }
        }
    } while (flag != 0);
}
```

(c) XMTC program

```
void pointer_jump (int S[n], int W[n], int n) {
    int W_tmp[n];
    int S_tmp[n];
    int *W_rd = W, *W_wt = W_tmp;
    int *S_rd = S, *S_wt = S_tmp;
    int *tmp_ptr;
    int crs_size = n/P + ((n%P) > 0);
    int flag = 1;
    while (flag != 0) {
        flag = 0;
        #pragma omp parallel num_threads(P) {
            #pragma omp parallel for reduction(+,flag) schedule(static, crs_size)
            for (int i = 0; i < n; i++) {
                if (S[i] != S[S[i]]) {
                    int x = 1;
                    flag += x;
                    W_wt [i] = W_rd[i] + W_rd[S_rd[i]];
                    S_wt [i] = S_rd[S_rd[i]];
                } else {
                    W_wt[i] = W_rd[i];
                    S_wt[i] = S_rd[i];
                }
            }
        }

        tmp_ptr = W_rd;     W_rd = W_wt;     W_wt = tmp_ptr;
        tmp_ptr = S_rd;     S_rd = S_wt;     S_wt = tmp_ptr;
    }
}
```

(d) OpenMP Program

Figure 3: Pointer jumping example showing simplicity of ICE code.

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, and 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-place update of 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. [1] 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 [2]. The loop continues until all threads are done.

[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 XMT `ps` operation uses registers, avoiding the hot spot.

```
⋮
serial code
shared variables declaration
⋮
pardo (pid = low; high; step) {
   ⋮
   private variables declaration
   lockstep parallel code
   ⋮
}
```

Figure 4: ICE language Syntax.

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 imporant 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 the later is slightly shorter and easier to write. Of course, an important takeaway is that both the XMTC and OpenMP codes, in figures 4(c) and 4(d) respectively, are considerably longer and more complex than the proposed ICE code in figure 4(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

Table 1: Comparison of the pardo and spawn constructs.

	pardo (lock-step)	**spawn (threaded)**
Syntax	pardo (CID=LB;UB;ST)	spawn (LB, UB)
Contexts Num. N	$(UB - LB)/ST + 1$	$UB - LB + 1$
First—last IDs	LB — $LB + ST \times N$	LB — UB
Stride	ST	1
MYPID	CID (user defined)	$
Execution Model	Each instruction is executed over all parallel contexts before the next one is initiated.	Instructions within a thread progress at their own pace.
Synchronization	After every Instruction	`join` or (`ps`)

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 `pardo` region. Each parallel context created by the outer `pardo` create multiple parallel contexts as specified by the inner `pardo`. *All these child parallel contexts created are lock-stepped 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.

[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 $\times$ the hyper-threading factor for multi-cores, and equals the number of TCUs on XMT.

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. This section will focus on the main challenges in building the new ICE compiler.

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) 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, to 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.

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

(a) Ice code	(b) XMTC translation

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

Handling control flow across multiple spawns Splitting `pardo` regions may cause complications for the program's control flow. There are two cases when this can happen: (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.

```
pardo (int i = 0; n; 1) {
  A[i+1] = c[i]; \\A1
  c[i] = A[i] + 1; \\A2
  B[i-1] = d[i]; \\B1
  d[i] = B[i] + i; \\B2
}
```

(a) Code in ICE

```
spawn(0,n) {
    unsigned i = $;
    A[i+1] = c[i]; \\A1
}

spawn(0,n) {
    unsigned i = $;
    c[i] = A[i] + 1; \\A2
    B[i-1] = d[i]; \\B1
}

spawn(0,n) {
    unsigned i = $;
    d[i] = B[i] + i; \\B2
}
```

(b) Equivalent code in XMTC

```
spawn(0,n) {
    unsigned i = $;
    A[i+1] = c[i]; \\A1
    B[i-1] = d[i]; \\B1
}

spawn(0,n) {
    unsigned i = $;
    c[i] = A[i] + 1; \\A2
    d[i] = B[i] + i; \\B2
}
```

(c) Optimized XMTC

Figure 6: Rescheduling memory accesses.

4.2 Optimization of the translated code

Splitting a `pardo` into multiple `spawn`s can degrade performance, due to the overhead of creating and managing more threads. Also, using memory to communicate information between `spawn`s 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.

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 XMTC 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 `spawn`s, 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

```
$\boldsymbol{M}$: set of all memory accesses
$\boldsymbol{CL_i} = \{\boldsymbol{m} \in \boldsymbol{M} : \boldsymbol{m}$ is a member of cluster $\mathbf{i}\}$
$\boldsymbol{NM} = \{\boldsymbol{m} \in \boldsymbol{M} : \boldsymbol{m}$ is not a member of any cluster$\}$

For an $\boldsymbol{m} \in \boldsymbol{NM}$:
$\boldsymbol{L_m} = \{\boldsymbol{m_L} \in \boldsymbol{M} :$ loop carried dependence between $\boldsymbol{m_L}$ and $\boldsymbol{m}\}$
$\boldsymbol{F_m} = \{\boldsymbol{m_F} \in \boldsymbol{M} : \boldsymbol{m}$ is Data flow dependent on $\boldsymbol{m_F}\}$
$\boldsymbol{C_m} = \{\boldsymbol{m_C} \in \boldsymbol{M} : \boldsymbol{m}$ is control dependent on value of $\boldsymbol{m_C}\}$.
$\boldsymbol{LP_m} = \{\boldsymbol{m_{LP}} \in \boldsymbol{M} : \boldsymbol{m}$ exist in a different loop from $\boldsymbol{m_{LP}}\}$
$\boldsymbol{NL_m} = \boldsymbol{L_m} \cap \boldsymbol{NM}$
$\boldsymbol{NF_m} = \boldsymbol{F_m} \cap \boldsymbol{NM}$
$\boldsymbol{NC_m} = \boldsymbol{C_m} \cap \boldsymbol{NM}$
$\boldsymbol{NLP_m} = \boldsymbol{LP_m} \cap \boldsymbol{NM}$

Define Procedure ConflictsWith ( $m, CL$ ) :
    if $NL_m \neq \Phi$ then
        return true
    if $L_m \cap CL \neq \Phi$ then
        return true
    if $LP_m \cap CL \neq \Phi$ then
        return true
    for $m_F \in NF_m$ do
        if ConflictsWith ($\boldsymbol{m_F}$ , $CL$ ) then
            return true
    for $m_C \in NC_m$ do
        if ConflictsWith ($\boldsymbol{m_C}$ , $CL$ ) then
            return true
    return false

Define Procedure cluster:
    Def: integer i = 0
    While ($NM \neq \Phi$) do
        define new cluster $\boldsymbol{CL_i}$
        for $m \in NM$ do
            if ConflictsWith (m, $CL_i$) then
                skip m
            else
                Add m to $\boldsymbol{CL_i}$
        i = i + 1
```

Figure 7: The clustering algorithm.

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 the current cluster (lines 28 - 34). A node is 'ready-to-fire' if it satisfies the conditions in the lines (13 - 25). In simple terms, when we consider a memory access to be added to cluster *i*, it and all the unscheduled data flow and control memory accesses it depends on must not have a 'loop carried' dependence with any member of that cluster. The clustering algorithm has a complexity of O(nl), where n is the number of instructions that access memory, and l is the number of resulting clusters. Since it relies solely on the dependency graph, the clustering algorithm does not require any special changes to work with nested ICE code.

Reducing the number of temporaries We attempt to minimize the amount of intermediate information communicated across `pardo` splits, such as branch directions, loop states, and intermediate data. This information is stored to and retrieved from memory, which can cause performance degradation. So in order to achieve maximum performance, avoidable memory accesses must be eliminated or promoted to local variables inside the `spawn`s that resulted from the splitting where possible. Alternatively, communicated information must be aggregated such that it can be stored and retrieved in the least number of accesses possible. For that reason, 1. We take clustering a step further. Memory accesses scheduled to an earlier cluster are moved to a later clusters if these clusters contain members dependent on the memory accesses and it is legal to do so. For a move to be legal, a memory access must satisfy all the conditions in the lines (13 - 25) in figure 7 for the target cluster, and all clusters in between. 2. We use bit vectors to record the branch directions for split `pardo`s, where each branch decision along the tree gets a single bit.

Handling Control Flow after Clustering The clustering process will result in reordering memory accesses which can 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

[4]Even though the execution order within a `pardo` is different from that of a loop, we are using the term loop carried dependence to refer to the parallel contexts cross dependence between different memory access in the pardo block

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 XMTC code. Thereafter the XMTC code is compiled using the existing gcc-based XMTC compiler [2]. 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.

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 end of a `pardo` block when generating IR from source. Also, we outline each parallel section into its own function, giving it a different context from its surrounding code. Furthermore, we use only the following native LLVM transformations that are guaranteed to not modify the memory ordering. First, we use **memory to register promotion** pass which transforms the code into SSA (Static Single Assignment) making subsequent optimizations much easier. Then we attempt to remove all extra instructions to make the code more efficient, and reduce the amount of information communicated across `pardo` splits. To that end we use **instruction combine** pass to combine instructions into simpler forms whenever possible, and the **Global Value Numbering (GVN)** pass to find all redundant instructions and remove them.

At this stage, we do the clustering and scheduling of `pardo` block instructions, and take steps to reduce the information communicated across splits. After clustering is complete, we mark the synchronization points between clusters. Following this, we use the **Control Flow Graph Simplify (CFGSimplify)** pass to remove all the extra control flow edges, and empty basic blocks that may have resulted from the steps taken to handle the control flow after clustering was complete (*i.e.*, cloning the CFG).

Finally, we translate the LLVM IR to XMTC high level code using our XMTC backend. The XMTC backend is a modified version of LLVM native C Backend with added support to generate high-level XMTC code. Here we do the splitting of `pardo`s into `spawn`s at the marked synchronization points. Also, in this stage we split loops and conditionals as discussed earlier, create all arrays for communicating intermediate data, and any other steps required for generating correct XMTC code. Furthermore, there are various trivial optimizations relating to expression eliminations that were implemented by hand due to lack of time. These optimizations are not related to ICE, and rely on regular compiler data flow analysis. The optimizations are performed over the generated XMTC code, and are not required for the correct operation of ICE.

After the XMTC code is produced, we compile it with the existing gcc-based XMTC compiler [2] to produce binaries for the XMT FPGA and XMT cycle accurate simulator.

5 Results

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

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

Benchmark	**Problem Size**	**Abrv.**
Integer Sort*	1048576	INT
Merging*	1000000	MRG
Sample Sort*	131072	SMP
Breadth First Search*	32768 nodes, 65536 edges	BFS
Breadth First Search (nested)*	32768 nodes, 65536 edges	NBFS
Graph Connectivity*	32768 nodes, 65536 edges	CVTY
Maximum Finding	262144	MAX
Tree Contraction	32768 nodes	CTRC
Tree Rooting*	32768 nodes, 65536 edges	RANK
2D Jacobi Stencil Computation (flattened)	512x512	JAC
2D Jacobi Stencil Computation (nested)	512x512	NJAC
LU Factorization (flattened)	512x512	LU
LU Factorization (nested)	512x512	NLU
Cholesky Factorization (flattened)	512x512	CHO
Cholesky Factorization (nested)	512x512	CHO
Topological Sort (nested)	32768 nodes, 65536 edges	TOBO

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 [22], [24], [31] 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 to express parallelism: `pardo` 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 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 preprocessor directives.

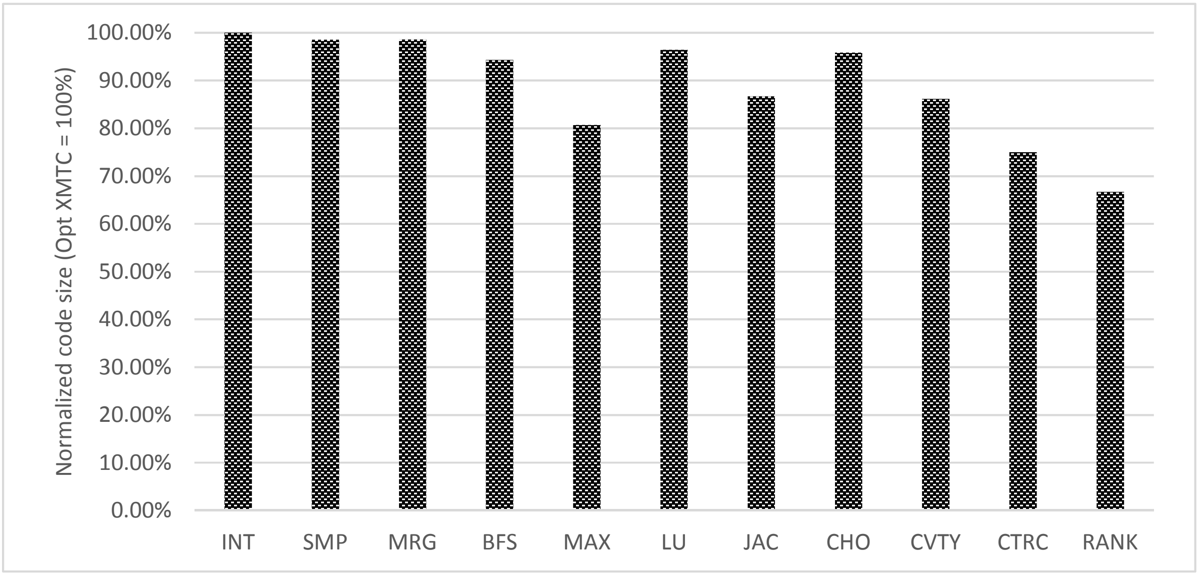


Figure 8: Code size for the entire program normalized to XMTC.

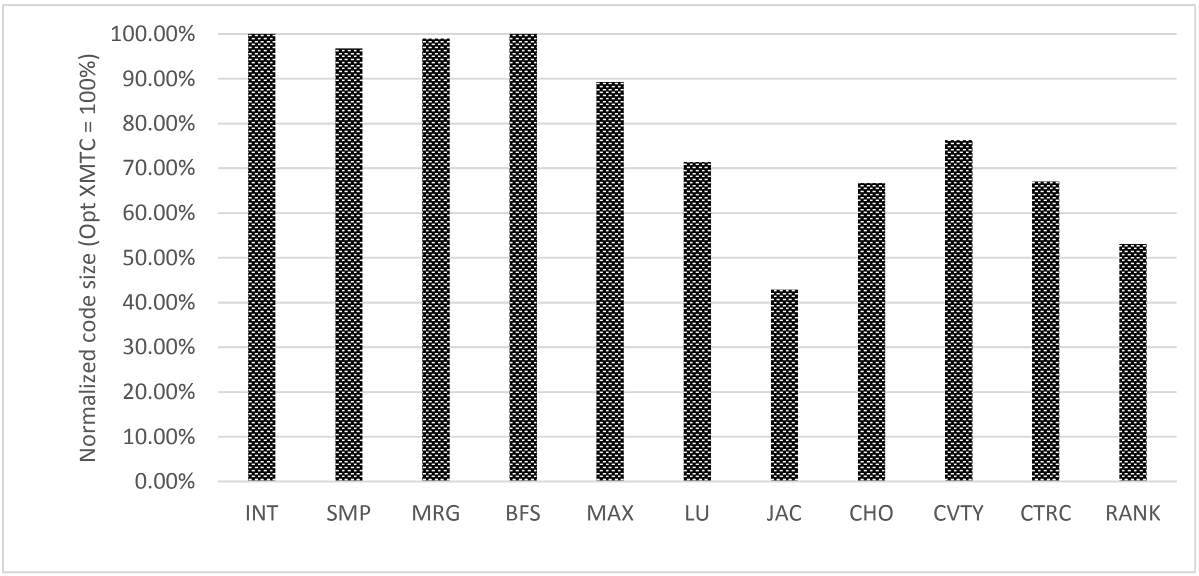


Figure 9: Code size of the algorithm's parallel sections normalized XMTC.

Now we look at figure 8 where we see a comparison of the reduction in the entire program code 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 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 9 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

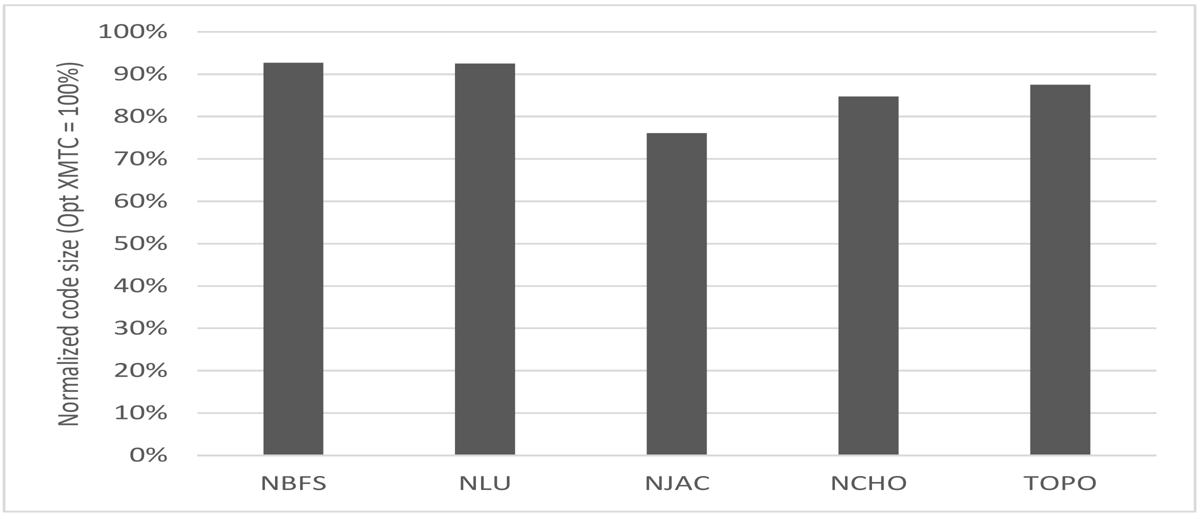


Figure 10: Code size for the entire program normalized to XMTC for nested benchmarks.

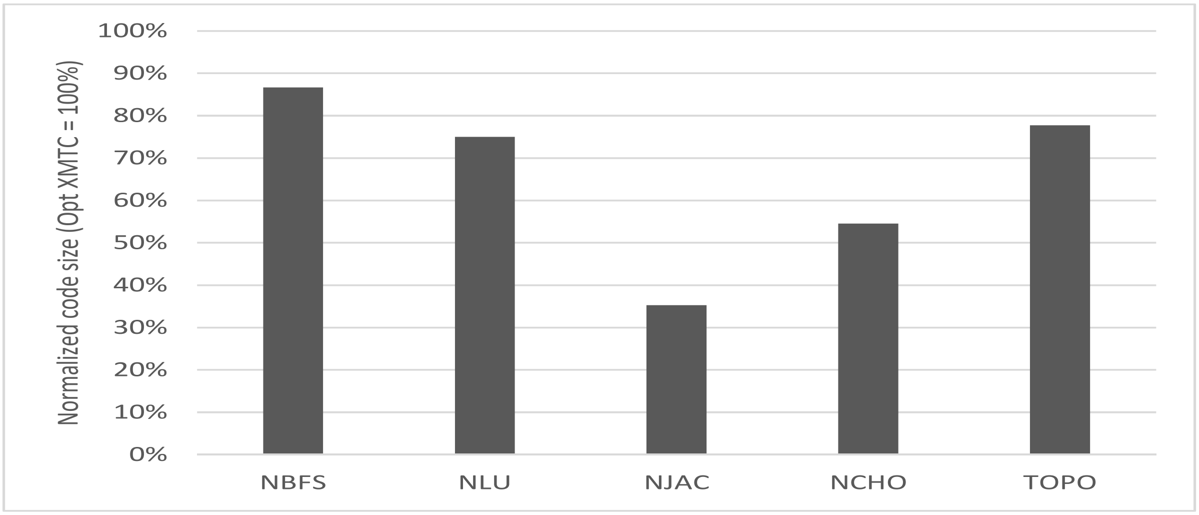


Figure 11: Code size of the algorithm's parallel sections normalized to XMTC for nested benchmarks.

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 10 and 11. Figure 10 provides a comparison of the reduction in size of the entire program code for nested ICE normalized to optimized nested XMTC, while figure 11 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 13.28% for the entire program, and 34.14% for the parallel algorithm portion of the code. We also notice in figure 11 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[5] 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 `spawn` blocks and temporaries has on the runtime performance of the translated XMTC code as discussed in section 4.2

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 fifteen out of the sixteen benchmarks had the same number of spawns and temporaries in both XMTC versions. For the single benchmark where the auto-generated XMTC had more spawns and temporaries compared to hand-written XMTC. This benchmark had multiple independent indirect memory references that cannot be

[5]Each temporary was used to store only one value that may be read multiple times.

Table 3: Number of spawn blocks and temporaries in both XMTC programs.

Benchmark	**Hand-written XMTC**		**Generated XMTC**	
	Spawns	Temp.	Spawns	Temp.
Integer Sort	3	0	3	0
Merging	4	0	4	0
Sample Sort	8	0	8	0
Breadth First Search	3	0	3	0
Breadth First Search (nested)	3	0	3	0
Graph Connectivity	12	2	13	3
Maximum Finding	4	0	4	0
Tree Contraction	7	4	7	4
Tree Rooting	5	2	5	2
Jacobi	2	1	2	1
Jacobi (nested)	4	1	4	1
LU Factorization	1	0	1	0
LU Factorization (nested)	2	0	2	0
Cholesky Factorization	2	0	2	0
Cholesky Factorization (nested)	3	0	3	0
Topological Sort	5	0	5	0

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 quality 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.

5.3 Performance

XMT is excellent at exploiting parallelism in irregular algorithms and we list examples of published work that shows XMT's speedups against commodity superscalar architectures in section 2.

In this section, we will focus on the performance comparison between ICE and XMTC. We use the XMT FPGA which has 64 TCUs to measure the performance for both the XMTC and ICE versions of the same algorithm pseudo-code. Figures 12 and 13 provides the speedup of

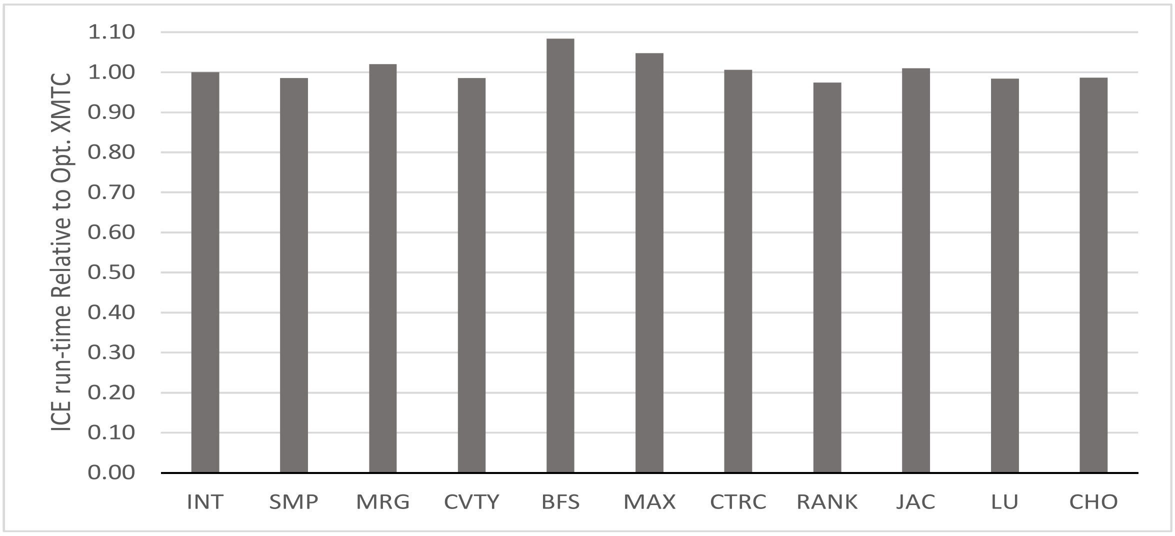


Figure 12: 64 TCU XMT processor speedup comparison

ICE normalized to hand-optimized XMTC for non-nested and nested programs, respectively. Figures 14 and 15 shows the net run-time improvement of ICE relative to hand-optimized XMTC, normalized to hand-optimized XMTC for both non-nested and nested programs, respectively. Run-time measurements are taken when the XMT binaries are run on the XMT FPGA. We provide the performance results for the ICE code normalized to hand-optimized XMTC programs.

We have taken steps to ensure that ICE is being compared to the fastest hand-optimized XMTC. Since memory accesses are the biggest source of overhead in XMT, we did not use temporaries in XMTC 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 XMTC, which takes considerably more programming effort to write than ICE. We see in figure 14 that ICE has a 0.76% speedup

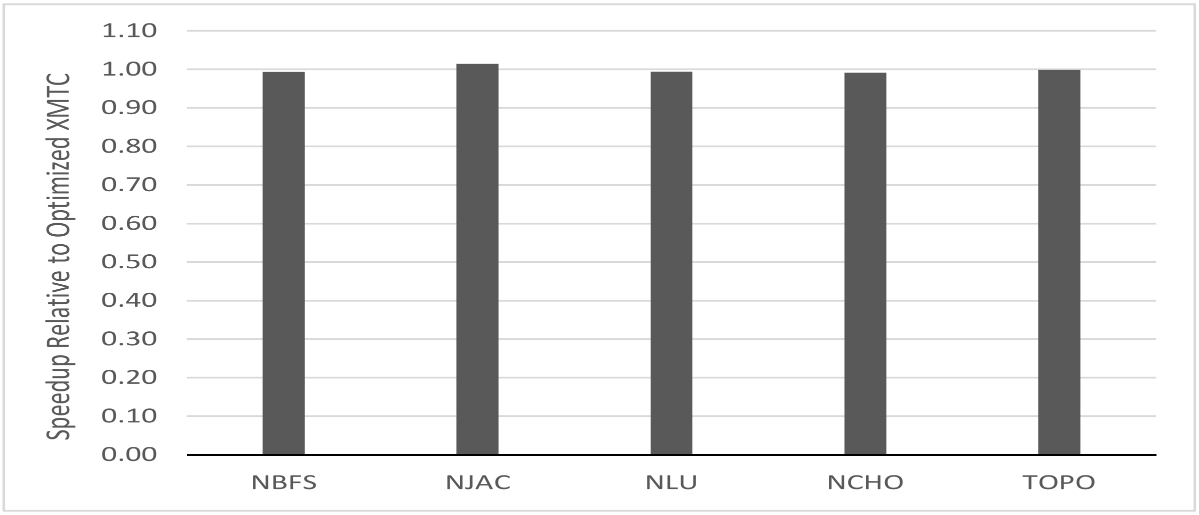


Figure 13: 64 TCU XMT processor speedup comparison for nested ICE

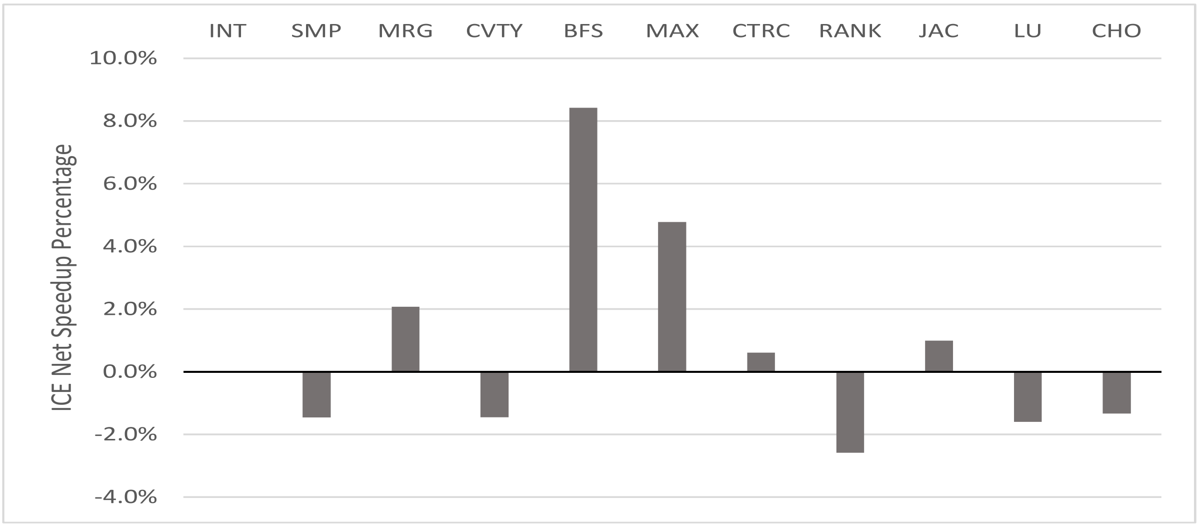


Figure 14: 64 TCU XMT net speedup of ICE normalized to optimized XMTC

on average for non-nested benchmarks, with maximum slowdown of 2.5% when compared to the performance of optimized XMTC. Figure 15, shows that ICE for nested programs has the same run-time on average as hand-optimized XMTC, with a maximum slowdown of 0.91%. We believe such negligible performance penalties for a much easier programming effort is an obvious good choice for programmers. For non-performance-expert programmers who cannot write highly optimized XMTC code, ICE might even provide a speedup.

We also notice that for some benchmarks, ICE has achieved a speed up when compared to hand-optimized XMTC. In this work, we do not claim that ICE can provide speed 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 speed ups. 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 version 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,

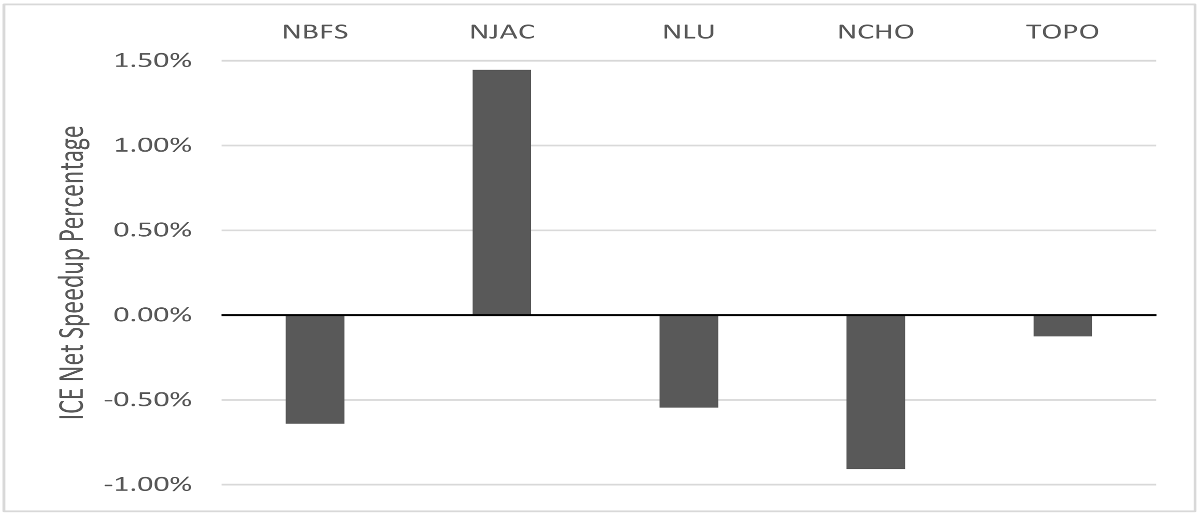


Figure 15: 64 TCU XMT net speedup of ICE normalized to optimized XMTC for the nested benchmarks

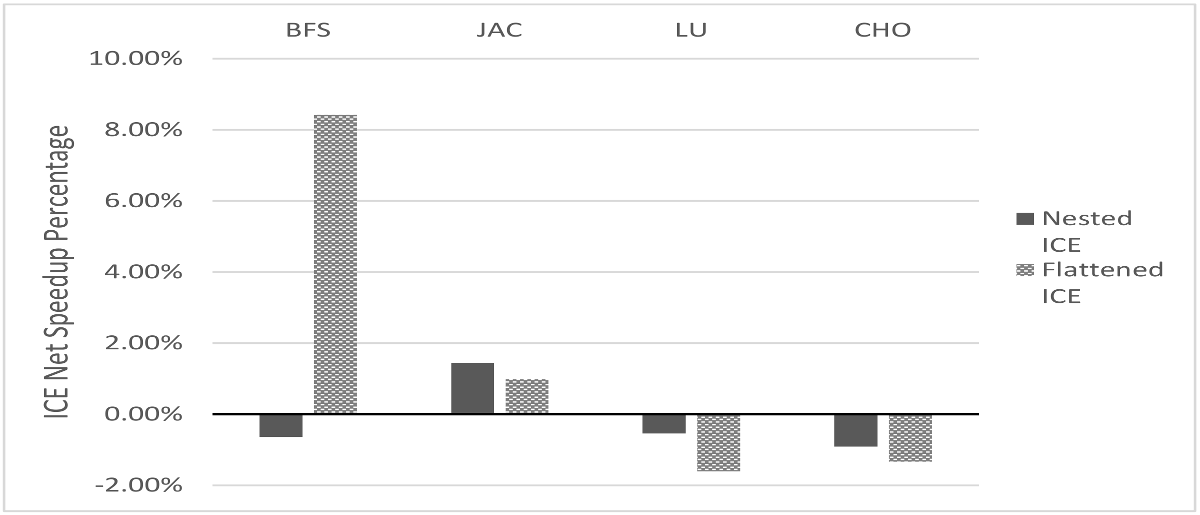


Figure 16: 64 TCU XMT net speedup comparison between nested and non-nested ICE normalized to optimized XMTC

instruction scheduling, the data pre-fetched, or the optimizations recognized by the XMTC compiler. For another benchmark subset (BFS, CTRC), the performance gain was a result of the LLVM compiler's native optimizations which is more recent than the GCC compiler used in XMTC implementation. This is combined with the ICE compiler specific optimization 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 16 where we see a comparison of both nested and the non-

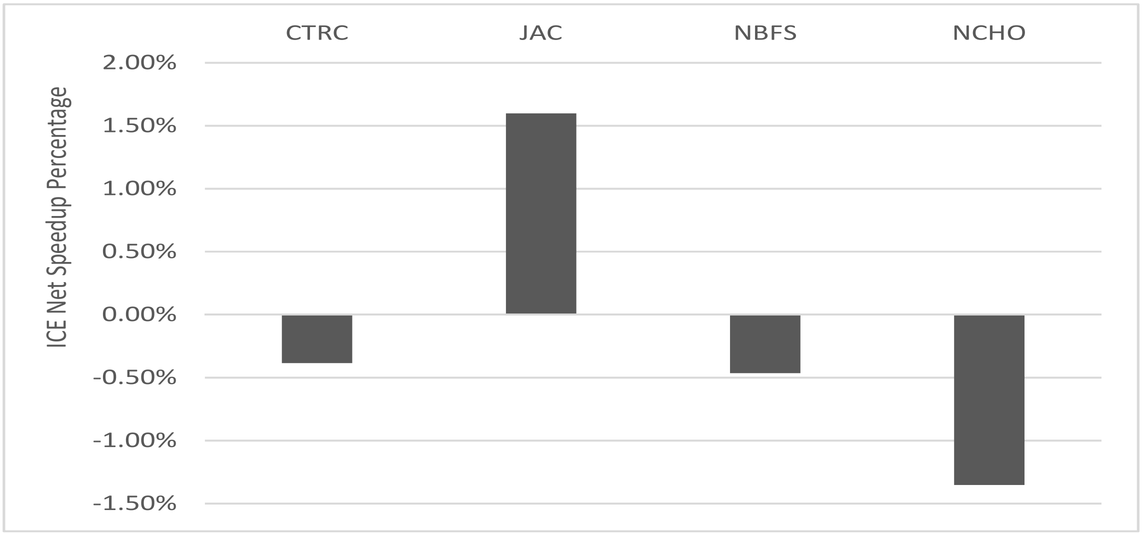


Figure 17: 1024 TCU cycle accruate XMT simulator net speedups for nested and non-nested ICE normalized to optimized XMTC

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 [23] using a 1024 TCU configuration. As can be seen in figure 17 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 SIGCSE 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. [33] suggests a "programmer's workflow" guiding the programmer on advancing an algorithm ICE abstraction[6] to an XMTC program and fine tuning its performance. The XMT hardware achieves strong speedups over serial algorithm for many parallel algorithms implemented using this workflow [33]. 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 XMTC. 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

[6]called high-level work-depth (HLWD)

languages; Cray's CHAPEL [28], SUN's Fortress [27], and IBM's X10 [38]. 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 [9] sought execution of compiler-extracted parallelism from APL programs on the IBM RP3[7]. However, APL did not provide sufficient support for the PRAM parallel algorithms literature. The V-RAM [4] 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., [5]. 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 [4][3] examined parallel algorithms and found that nearly all are parallel operations over collections of values, called data-parallelism by Hillis and Steele [21]. The languages based on it are referred to as data-parallel languages (e.g. [1], [8], [15], [25]). Also, Blelloch contrasted flat data-parallel languages[8] with nested data-parallel languages[9]. 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 [29] 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. [30] further optimized work stealing to an improvement called Lazy Binary Splitting (LBS). Cilk++ [16] 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 [10], to parallel algorithms texts [22], [24], [31] and demonstrated by the merging algorithm in [32]. However, Cilk 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.

[7]The IBM RP3 built on the NYU Ultracomputer project, which also inspired XMT.
[8]A sequential function can be applied in parallel over a set of values
[9]Any function - including parallel - can be applied in parallel over a set of values

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

[1] Arvind, R. S. Nikhil and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. on Programming Languages and Syst.*, vol. 11, no. 4, pp. 598–632, Oct. 1989.

[2] A. O. Balkan and U. Vishkin, "Programmer's manual for XMTC language, XMTC compiler and XMT simulator," University of Maryland Institute for Advanced Computer Studies (UMIACS), Tech. Rep. UMIACS-TR-2005-45, 2006. [Online]. Available: `http://www.umiacs.umd.edu/users/vishkin/XMT/manual4xmtc1out-of2.pdf`.

[3] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, pp. 85–97, 3 Mar. 1996, ISSN: 0001-0782.

[4] ——, *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[5] G. Blelloch and J. Greiner, "A provable time and space efficient implementation of nesl," in *ACM SIGPLAN Int. Conf. on Functional Programming*, 1996.

[6] G. Caragea and U. Vishkin, "Better speedups for parallel max-flow," in *Proc. 23rd ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, 2011.

[7] G. C. Caragea, A. Tzannes, F. Keceli, R. Barua and U. Vishkin, "Resource-aware compiler prefetching for many-cores," in *International Symposium on Parallel and Distributed Computing*, 2010.

[8] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[9] W. Ching and D. Ju, "Execution of automatically parallelized API programs on RP3," *IBM J. of research and Development*, vol. 35, pp. 767–778, 5/6 1991.

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

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, E. Santos, E. Santos, E. Santos, R. Subramonian and T. von Eicken, "Logp: Towards a realistic model of parallel computation," *SIGPLAN Not.*, vol. 28, no. 7, pp. 1–12, 1993.

[12] J. A. Edwards and U. Vishkin, "Better speedups using simpler parallel programming for graph connectivity and biconnectivity," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, ACM, 2012, pp. 103–114.

[13] J. A. Edwards and U. Vishkin, "Brief announcement: Truly parallel burrows-wheeler compression and decompression," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, ACM, 2013, pp. 93–96.

[14] ——, "Parallel algorithms for burrows-wheeler compression and decompression," *Theor. Comput. Sci.*, vol. 525, pp. 10–22, 2014.

[15] J. T. Feo, D. C. Cann and R. R. Oldehoeft, "A report on the Sisal language project," *J. of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, Dec. 1990.

[16] M. Frigo, P. Halpern, C. E. Leiserson and S. Lewin-Berlin, "Reducers and other cilk++ hyperobjects," in *Proc. 21st Annu. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[17] F. Ghanim, R. Barua and U. Vishkin, "Poster: Easy pram-based high-performance parallel programming with ice," in *The 25th International Conference on Parallel Architectures and Compilation Techniques*, 2016.

[18] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU ultracomputer: Designing a MIMD, shared-memory parallel machine (extended abstract)," in *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, IEEE Computer Society Press, 1982, pp. 27–42.

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

[20] Z. He and B. Hong, "Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms," in *Proc. 24th IEEE Int. Parallel and Distributed Processing Symp.*, 2010.

[21] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

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

[23] F. Keceli, A. Tzannes, G. C. Caragea, R. Barua and U. Vishkin, "Toolchain for programming, simulating and studying the xmt many-core architecture," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 1282–1291.

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

[25] P. Mills, L. S. Nyland, J. Prins, J. H. Reif and R. A. Wagner, "Prototyping parallel and distributed programs in proteus," in *Symp. Parallel and Distributed Processing 1991*, IEEE Comput. Soc.

[26] D. Naishlos, J. Nuzman, C.-W. Tseng and U. Vishkin, "Towards a first vertical prototyping of an extremely fine-grained parallel programming approach," in *Proc. 13th annu. ACM symp. on Parallel algorithms and architectures (SPAA)*, 2001.

[27] (). Project fortress, [Online]. Available: `http://projectfortress.java.net/`.

[28] (). The chapel parallel programming language, [Online]. Available: `http://chapel.cray.com/`.

[29] *The mit cilk home page: Http://supertech.csail.mit.edu/cilk/.*

[30] A. Tzannes, G. C. Caragea, R. Barua and U. Vishkin, "Lazy binary-splitting: A run-time adaptive work-stealing scheduler," in *Proc. 15th ACM SIGPLAN symp. on Principles and practice of parallel programming (PPOPP)*, 2010.

[31] U. Vishkin, "Thinking in parallel: Some basic data-parallel algorithms and techniques - course class notes," [Online]. Available: `http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf`.

[32] ——, "Using simple abstraction to guide the reinvention of computing for parallelism," *CACM*, vol. 54,1, pp. 75–85, 2011.

[33] 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 (editors: S. rajasekaran, j. reif)," in. Chapman and Hall/CRC Press, 2008.

[34] U. Vishkin, "Prefix sums & an application thereoff.," *U.S. Patent*, no. 6 542 918, 2003.

[35] ——, "Spawn-join instruction set architecture for providing explicit multi-threading (xmt)," *U.S. Patent*, no. 6 463 527, 2002.

[36] U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman, "Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract)," in *Proc. 10th annu. ACM symp. on Parallel algorithms and architectures (SPAA)*, 1998.

[37] X. Wen and U. Vishkin, "FPGA-based prototype of a PRAM-on-chip processor," in *Proc. ACM Computing Frontiers*, 2008.

[38] (). X10: Performance and productivity at scale, [Online]. Available: `http://x10-lang.org/`.