Poster: Easy PRAM-based High-performance Parallel Programming with ICE [*]

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

[1]Electrical and Computer Engineering Department
[2]University of Maryland Institute for Advanced Computer Studies (UMIACS)
University of Maryland
College Park, MD, 20742, USA
{fghanim,barua,vishkin}@umd.edu

ABSTRACT

Large performance growth for processors requires exploitation of hardware parallelism, which, itself, requires parallelism in software. In spite of massive efforts, automatic parallelization of serial programs has had limited success mostly for regular programs with affine accesses, but not for many applications including irregular ones. It appears that the bare minimum that the programmer needs to spell out is which operations can be executed in parallel. However, parallel programming today requires so much more. The programmer is expected to partition a task into subtasks (often threads) so as to meet multiple constraints and objectives, involving data and computation partitioning, locality, synchronization, race conditions, limiting and hiding communication latencies. It is no wonder that this makes parallel programming hard, drastically reducing programmer's productivity and performance gains hence reducing adoption by programmers and their employers.

Suppose, however, that the effort of the programmer is reduced to merely stating operations that can be executed in parallel, the 'work-depth' bare minimum abstraction developed for PRAM (the lead theory of parallel algorithms). What performance penalty should this incur? Perhaps surprisingly, the upshot of our work is that this can be done with no performance penalty relative to hand-optimized multi-threaded code.

CCS Concepts

•Theory of computation → Parallel algorithms;•Computing methodologies → Parallel algorithms; Parallel programming languages;•Computer systems organization → *Parallel architectures;*

Keywords

Parallel Programming Languages; ease of programming; ICE; Parallel Algorithms; PRAM

1. INTRODUCTION

Parallel programming for performance is hard. But, can parallel programming languages alone mitigate that? While languages can help, we believe that the answer lies in the whole ecosystem they represent: the underlying algorithmic theory (or lack thereof) they are based on, and whether parallel hardware can support irregular applications. What is needed is a well-integrated ecosystem of (i) an easy-to-program programming model, backed by (ii) a solid algorithmic theory tailored to (iii) a hardware architecture capable of exploiting parallelism in irregular programs, and supported by (iv) an efficient compiler that make them work in concert. While relying on a published platform, our work contributes: new programming model and compiler, and experimental evaluation of the compiler efficiency. In this work we use a new programming model called Immediate Concurrent Execution (ICE).

The Algorithm Theory. ICE is grounded in the Parallel Random Access Machine/Model (PRAM) algorithmic theory. Developed in the 1980s and early 1990s, PRAM is the parallel equivalent of the standard Random Access Machines serial algorithmic theory. PRAM is very rich and feature diverse in algorithms and techniques, and is very simple to use. The lead PRAM abstraction known as work-depth (WD) defines ICE. WD prescribes a sequence of time steps, each containing a set of operations to be executed concurrently within unit time. The ICE computation model abstracts away opportunities for using local memories, and minimizing communication or synchronization, work distribution and scheduling making it much easier to specify PRAM parallel algorithms. PRAM/ICE is a desirable choice for parallel programmers due to its expressiveness, brevity, and its aforementioned properties.

The Programming Language. The ICE programming model transcribes PRAM algorithms as-is. ICE extends the C language by the keyword `pardo`(imported from PRAM) for specifying lock-step parallelism. In a lock-step parallel program, each statement in a parallel loop executes exactly in the same cycle across all iterations. **Example.** Figure 1(b) provides ICE code for the basic PRAM list ranking algorithm, defined in 1(a). In every iteration of the serial while loop the `pardo` instruction guides that for every element i whose successor $S(i)$ is not yet the last element: (i) its weight is updated to the distance to the successor of its current successor; and (ii) this successor of successor is first read (for all elements) and only later stored as the new successor; namely, the right hand side of all concurrent instructions is executed prior to to the left hand side of any of these instructions.

The System Platform. We test case our programming model using the explicit multi-threaded (XMT) platform developed at the University of Maryland (UMD) [1] (not to be confused with the CRAY XMT). XMT was designed with irregular programs such as those found in PRAM algorithms in mind, which allows it to attain excellent performance for such programs. This made it an ideal platform for testing our work. The XMT architecture is programmed through a multi-threaded programming language called XMTC, which is a modest extension to the C language. XMTC uses `spawn` to express concurrent threads.

[*]A detailed technical report of this work is available at http://drum.lib.umd.edu

PACT '16 September 11-15, 2016, Haifa, Israel
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4121-9/16/09.
DOI: http://dx.doi.org/10.1145/2967938.2974053

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

Figure 1: Pointer jumping Algorithm.

The ICE Lock-step assumption created a serious implementation challenge. Due to a variety of technology constraints (such as hierarchical memory and power), the quest for parallel scalability is mainly through multi-threading in hardware. Hence, for scaling practically all parallel programming languages use multi-threading, where a thread proceeds at its own pace until the next synchronization point. The XMT system platform is also multi-threaded and cannot efficiently implement the tight synchrony dictated by the lock-step assumption. This meant that our same-cycle lock-step assumption had to be enforced by the compiler translating ICE to XMTC. See Section 2. Contrasting the ICE code in Figure 1(b) and the hand-optimized XMTC code in Figure 1(c) can help appreciate: (i) the challenge in providing effective translation, (ii) the simplicity of programming in ICE over XMTC, (iii) the difference in code size, and (iv) the implied relative ease in producing the ICE code.

Our main accomplishment in this new work is that the translation does not incur, on average, any performance penalty over hand-optimized multi-threaded programming in XMTC. See Section 3. Consequently, ICE is the first language that can transcribe PRAM algorithms and automatically translates them into effective threaded programs. Combined with prior XMT work, this establishes feasibility of reducing the main (work-depth) abstraction of parallel algorithmic theory to practice, arguably a major result.

2. TRANSLATION

In this work we translate programs written in ICE to the XMTC high level language. Parallel regions in ICE are expressed using the `pardo` keyword as seen in Figure 1(b). During translation barriers are introduced into parallel sections to maintain the correctness of the lock-step ICE program. Threads advance unpredictably at their own pace, which may result in executing memory reads and writes in a different order than originally intended. Barriers will ensure that the lock-step pacing is maintained. During translation, we split a pardo block into multiple spawn blocks. Splitting occurs at points where a barrier was added.

Splitting a pardo into multiple spawns and using memory to communicate information between spawns degrades performance, due to overhead. This is exacerbated when the number of splits is high. Hence it is crucial to avoid splitting whenever possible, and to mitigate the effects of the unavoidable splits. To minimize the number of splits needed, we rearrange memory accesses within a pardo region into clusters. Each cluster represents a spawn block. These clusters consist of a group of memory accesses that have no dependencies between them across different parallel contexts.

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

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

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

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

Figure 2: The clustering algorithm.

When a pardo region is first split into multiple spawns, often there are more splits than necessary. By rearranging and grouping independent memory accesses into same spawn block we keep only the necessary splits. This is performed through a list scheduling scheme we call clustering. Figure 2 shows the algorithm used.

3. RESULTS

In this section we present the results of our experiments comparing the performance of ICE and hand-optimized XMTC. We developed a suite of 11 benchmarks based on common PRAM 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. The results are summarized in figure 3.

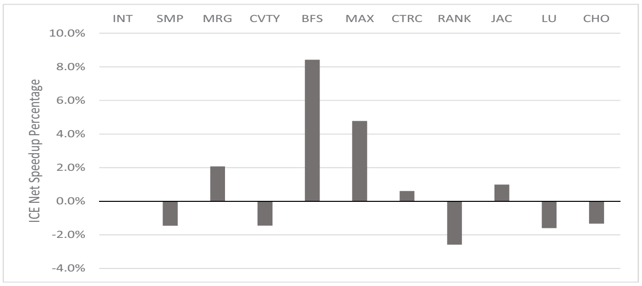

Figure 3: ICE Net speedup normalized to optimized XMTC

As we see in figure 3, ICE code achieves comparable performance to hand-optimized XMTC code, in spite of the latter taking considerably greater programming effort. ICE has a 0.76% speedup on average, with maximum slowdown of 2.5% when compared to the performance of optimized XMTC. We also notice that for some benchmarks, ICE has achieved a speed up when compared to hand-optimized XMTC. However, we do not claim in this work that ICE can provide speed ups over XMTC.

4. ACKNOWLEDGMENTS

Partial support by NSF award 1161857 is gratefully acknowledged.

References

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