Easy PRAM-based High-performance Parallel Programming with ICE

Fady Ghanim Rajeev Barua Uzi Vishkin

Technical Contribution

Multi-threaded execution is the norm

Problem statement Can we enable tightly-synchronous threading-free programming for multi-threaded execution?

Current understanding No. Performance programming must be multi-threaded

New result Yes:

- Parallel programming can be lock-step
- With **no** performance penalty

Significance

Hardware parallelism is increasing

Auto parallelization in hardware or software?

Limited success and scaling. Not for irregular programs → Parallel algorithms & programming

- But, how to *minimize human effort*?

Our goal: *Specify what is parallelizable, but nothing else*

- Nobody knows to do less…

Fact: parallel programmer must specify much more. He/she is expected to partition a task into subtasks (threads) so as to meet multiple constraints and objectives, involving data and computation partitioning, locality, synchronization, race conditions, limiting and hiding communication latencies

- Pain of parallel programming of the available ecosystem: commodity hardware and parallel programming languages

Intermediate Concurrent Execution (ICE) Model

- A parallel algorithm is expressed as a series of time steps of parallel operations
- Lock-step execution model; A time step is not executed until all operations of the previous time step are completed
- Parallel Random Access Machines (PRAM) is the main parallel algorithmic theory
  - The "Work-Depth (WD)" abstraction. Pseudocode uses "pardo". defines ICE
  - PRAM is a large latent knowledge base of algorithms and technique
- Uses the XMT platform developed at UMD
  - Designed with irregular algorithms (like those in PRAM) in mind
  - Programmed using threaded parallel language called XMT-C
  - XMTC uses 'spawn' keyword to create concurrent threads
- The ICE compiler translates the ICE high level language into XMTC

The ICE Language

- The ICE language is based on the C language
  - Extends C by adding a new keyword "pardo". Used to specify parallelism as in WD
  - Shared variables are declared outside the pardo block
  - Private variables are declared within the pardo block

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

ICE Language Syntax

- In ICE, unlike threaded languages, a programmer only needs to specify parallelism
- ICE compiler produces high performance XMTC code
- ICE is the first language that can transcribe PRAM algorithms and automatically translates them into effective threaded programs

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

Pointer Jumping Example

Translation: ICE to XMTC

- Threaded model (XMTC) is incompatible with lock-step model (ICE)
  - In lock-step, different parallel contexts progress in concert one step at a time
  - Threads each progresses on its own pace regardless of other threads
- Correct translation requires synchronizing threads by introducing barriers between dependent memory accesses
- A 'pardo' block is split into multiple 'spawn' blocks
  - The splitting occurs wherever barriers were added
  - Use temporary variables to communicate data and control flow between different 'spawn' blocks

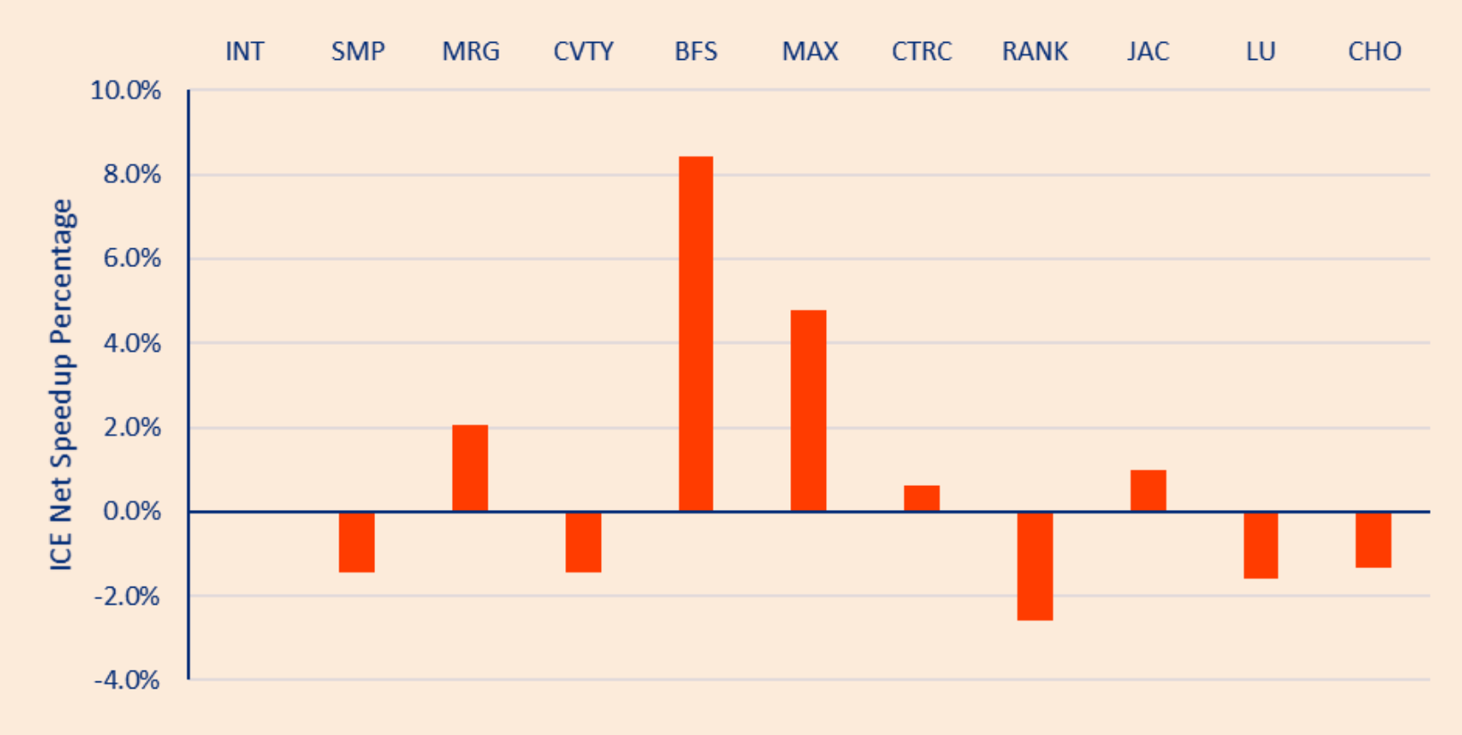
Translation: Optimization

- Splitting a pardo block into multiple spawn blocks causes performance degradation
- So does using shared memory to communicate information
- Minimizing the number of splits is crucial to high performance
  - Consolidate unnecessary splits wherever possible
  - Use a list scheduling algorithm to group independent memory accesses into clusters
  - Each cluster becomes a spawn block later on
  - Called clustering algorithm

```
$M$: 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
```

Experimental Results

- Goal: ICE produces XMTC code that has a comparable performance to hand optimized XMTC
- Developed a benchmark suite consisting of 11 PRAM algorithms
- The experiment was conducted by
  - Producing a pseudocode for each algorithm in the suite
  - Using the pseudocode, two implementations were produced; an XMTC version manually optimized for best performance, and an ICE version
  - Compile and execute each version on a 64 core XMT processor
- ICE achieves comparable performance to optimized XMTC while requiring considerably less effort
  - Average speedup of ICE across all benchmarks is 0.76%
  - Maximum slowdown was 2.7%, Maximum speedup was 8.3%
- We do not claim that ICE will provide speedups compared to hand-optimized XMTC



Net Speedup of ICE normalized to hand-optimized XMTC

Benchmark Suite

Abrv.	Algorithm name
INT	Integer sort
SMP	Sample Sort
MRG	Merge
CVTY	Connectivity
BFS	Breadth First Search
MAX	Maximum finding
CTRC	Tree Contraction
RANK	Tree Ranking
JAC	Jacobi
LU	LU Factorization
CHO	Cholesky Factorization

Conclusion

- Transcribe PRAM algorithms right out of the textbook & go fishing

- Freeing parallel programmers from current pain points
- Get the best performance with proper compiler and architecture
- Was it premature to replace the Parallel Algorithms section by a Multithreaded Algorithms section in some standard algorithms texts?
- To be fair, we surprised even ourselves. The XMT (explicit multi-threading) platform expected a manual workflow: starting from PRAM algorithms produce multi-threaded programs. Not directly-transcribed PRAM.
- New work goes back to : U. Vishkin, Synchronized Parallel Computation, D.Sc. Dissertation, CS, Technion, 1981, where WD was introduced.