ABSTRACT

Title of Dissertation:	STUDY OF FINE-GRAINED, IRREGULAR PARALLEL APPLICATIONS ON A MANY-CORE PROCESSOR
	James Alexander Edwards Doctor of Philosophy, 2020
Dissertation Directed by:	Professor Uzi Vishkin University of Maryland Institute for Advanced Computer Studies and Department of Electrical and Computer Engineering

This dissertation demonstrates the possibility of obtaining strong speedups for a variety of parallel applications versus the best serial and parallel implementations on commodity platforms. These results were obtained using the PRAM-inspired Explicit Multi-Threading (XMT) many-core computing platform, which is designed to efficiently support execution of both serial and parallel code and switching between the two.

Biconnectivity: For finding the biconnected components of a graph, we demonstrate speedups of 9x to 33x on XMT relative to the best serial algorithm using a relatively modest silicon budget. Further evidence suggests that speedups of 21x to 48x are possible. For graph connectivity, we demonstrate that XMT outperforms two contemporary NVIDIA GPUs of similar or greater silicon area. Prior studies of parallel biconnectivity algorithms achieved at most a 4x speedup, but we could not find biconnectivity code for GPUs to compare biconnectivity against them.

Triconnectivity: We present a parallel solution to the problem of determining the triconnected components of an undirected graph. We obtain significant speedups on XMT over the only published optimal (linear-time) serial implementation of a triconnected components algorithm running on a modern CPU. To our knowledge, no other parallel implementation of a triconnected components algorithm has been published for any platform.

Burrows-Wheeler compression: We present novel work-optimal parallel algorithms for Burrows-Wheeler compression and decompression of strings over a constant alphabet and their empirical evaluation. To validate these theoretical algorithms, we implement them on XMT and show speedups of up to 25x for compression, and 13x for decompression, versus bzip2, the de facto standard implementation of Burrows-Wheeler compression.

Fast Fourier transform (FFT): Using FFT as an example, we examine the impact that adoption of some enabling technologies, including silicon photonics, would have on the performance of a many-core architecture. The results show that a single-chip many-core processor could potentially outperform a large high-performance computing cluster.

Boosted decision trees: This chapter focuses on the hybrid memory architecture of the XMT computer platform, a key part of which is a flexible all-to-all interconnection network that connects processors to shared memory modules. First, to understand some recent advances in GPU memory architecture and how they relate to this hybrid memory architecture, we use microbenchmarks including list ranking. Then, we contrast the scalability of applications with that of routines. In

particular, regardless of the scalability needs of full applications, some routines may involve smaller problem sizes, and in particular smaller levels of parallelism, perhaps even serial. To see how a hybrid memory architecture can benefit such applications, we simulate a computer with such an architecture and demonstrate the potential for a speedup of 3.3X over NVIDIA's most powerful GPU to date for XGBoost, an implementation of boosted decision trees, a timely machine learning approach.

Boolean satisfiability (SAT): SAT is an important performance-hungry problem with applications in many problem domains. However, most work on parallelizing SAT solvers has focused on coarse-grained, mostly embarrassing parallelism. Here, we study fine-grained parallelism that can speed up existing sequential SAT solvers. We show the potential for speedups of up to 382X across a variety of problem instances. We hope that these results will stimulate future research.

STUDY OF FINE-GRAINED, IRREGULAR PARALLEL APPLICATIONS ON A MANY-CORE PROCESSOR

by

James Alexander Edwards

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Acknowledgments

I would like to thank my advisor, Prof. Uzi Vishkin, for his patient guidance. Throughout my academic career, he was always more than willing to listen to my (often convoluted) explanations of my work, provide insightful comments, and help me ask the right questions about my work.

I appreciate the time and effort spent by the dissertation committee members, Prof. Rajeev Barua, Prof. Joseph JaJa, Prof. Alan Sussman, Assoc. Prof. Behtash Babadi, and Dr. Bogdan Kosanovic, especially given the unusual circumstances this year. I would like to thank Prof. Barua in particular for helping me to sharpen the main contributions of this dissertation.

I would like to thank former and current members of the XMT team for their help in making this work possible. George Caragea, Fuat Keceli, Alexandros Tzannes, and Xingzhi Wen not only provided the XMT toolchain but took the time to help me understand how the tools worked and how to upgrade them. The research done by Fady Ghanim and Sean O'Brien helped to validate my own work.

I am grateful to my father for sparking my interest in computers, and my mother and sister for cheering me on and always being there for me whenever I needed them.

And, last but most importantly, I would like to thank God for guiding and uplifting me through this stage of my life.

Specific acknowledgments

We would like to thank the authors of [149] for providing us with the CUDA code for graph connectivity tested in Chapter 2.

We gratefully acknowledge comments provided by Prof. Rina Dechter related to putting prior work on SAT solvers in proper perspective in Chapter 7.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Improvements in the performance of serial processors are reaching a limit due to constraints on power consumption as a result of the end of Dennard scaling. In response to this, computing vendors have produced parallel processors with increasing numbers of cores, and programmers are now expected to employ parallelism in their applications to obtain maximum performance. However, constraints imposed by current computing architectures limit the improvement in performance that can be achieved in many cases.

Despite these limits, current research focuses on off-the-shelf hardware, particularly multi-core CPUs and many-core GPUs. This has led to a chicken-and-egg impasse where vendors look to existing benchmarks to guide architectural improvements while those same benchmarks are built around the limitations of existing hardware. As will be shown in this thesis, attempts to use such hardware to improve the performance of applications beyond simple benchmark kernels have been met with mixed success.

The purpose of this study is twofold: first, to show the speedups that can potentially be obtained for a variety of applications given buildable hardware, and second, to articulate the gap between what such applications require and what existing architectures provide, with the goal of helping to guide the development of future computing platforms. In order to break the above chicken-and-egg impasse, we study the performance of these applications on an experimental architecture in addition to commercial computers. This will help us to identify whether bottlenecks we encounter are inherent to the algorithm or a result of architectural choices.

This study focuses on applications that are challenging for existing platforms. They tend to exhibit *fine-grained* parallelism, which means that threads tend to be short and access small units of data from memory. Furthermore, the parallelism tends to be *irregular*, meaning that the number and length of threads, as well as the memory access patterns, cannot be predicted in advance.

The experimental computer architecture used in this study is the Explicit Multi-Threading (XMT[1]) general-purpose architecture [170], developed at the University of Maryland, which aims to improve single-task completion time and ease-of-programming for parallel applications by supporting Parallel Random Access Model (PRAM) programming [94, 104]. In particular, XMT was designed to efficiently support applications with fine-grained, irregular parallelism.

XMT currently has two embodiments: a 64-core FPGA prototype and a configurable cycle-accurate simulator named XMTSim. XMTSim allows us to study configurations of XMT that would require the same resources (such as silicon area, power, and bandwidth to DRAM) as contemporary platforms for a fair comparison. To further support such comparison, we compare our code on XMT versus results published by others on existing platforms where available to avoid biasing

[1] Not to be confused with the code name Cray XMT used during 2007-2011

the results.

The applications in this study cover a variety of application domains and use various data structures (trees, graphs, strings, vectors) and data types (integer and floating-point). A common theme among all of the parallel algorithms studied here is that each solves the same problem as the best serial algorithm for a given application using asymptotically the same number of operations but with a lower parallel depth.

The remainder of this thesis is organized as follows: Chapters 2 and 3 study biconnectivity and triconnectivity, two advanced graph problems whose parallel algorithms build upon simpler parallel primitives. Chapter 4 studies the Burrows-Wheeler (BW) lossless data compression algorithm and evaluates its potential application to improving effective network bandwidth. Chapter 5 studies the Fast Fourier transform (FFT), a memory-intensive mathematical operation used in digital signal processing. Chapter 6 studies XGBoost, an implementation of the boosted decision tree approach to machine learning; this chapter also examines the trend of multi-core and GPU design towards the same hybrid memory architecture underlying XMT. Chapter 7 studies Boolean satisfiability (SAT), which leads to articulating the need to add efficient support of fine-grained nesting in XMT. Finally, Chapter 9 concludes.

Chapter 2: Graph biconnectivity

2.1 Introduction

Given an undirected graph G, two vertices u and v in G are in the same *connected component* of G if there is a path connecting them, and the *graph connectivity problem* is finding all connected components of an input graph G. The *diameter* of a connected graph is the length of the longest path in the set of all shortest paths between every pair of vertices in the graph.

Biconnectivity is a property of undirected graphs; an undirected graph G is called *biconnected* if and only if it is connected and remains so after removing any vertex and all edges incident on that vertex. A graph S is an *induced subgraph* of G if it comprises a subset of the vertices of G and all the edges of G connecting two vertices in S. A *biconnected component* of G is an induced subgraph of G that is biconnected whose vertex set cannot be expanded while maintaining the biconnectivity of its induced subgraph. A vertex whose removal increases the number of connected components in the graph is called an *articulation point*, and an edge whose removal increases the number of connected components is called a *bridge*. In this chapter, the *biconnectivity problem* is understood as the problem of determining the biconnected components, articulation points, and bridges of an undirected graph, and a *biconnectivity algorithm* is an algorithm that solves the biconnectivity problem.

Connectivity is one of the most elementary graph problems. However, for brevity we pay more attention to biconnectivity, the more advanced problem considered in this work. Biconnectivity is an interesting problem to study for two reasons. First, the biconnected components of a graph can reveal useful information about the graph. For instance, if the graph represents a computer network, then a biconnected component of the graph is a subset of the network that will remain connected even if one computer fails, and articulation points (or bridges) are computers (or connections between computers) whose failure will disconnect the network. Second, biconnectivity algorithms are relatively complex: they are among the most advanced algorithms given in parallel algorithms textbooks and nearly the most advanced in serial algorithms textbooks, and biconnectivity or simpler problems were the basis for papers on other parallel computing platforms. Complex algorithms for natural problems may be better predictors of system behavior than the often used small kernels.

In serial computing, depth-first search is regarded as the best biconnectivity algorithm. However, power constraints impose a limit on the maximum performance of serial processors, and parallel processors are becoming the only way to improve performance. Therefore, it is desirable to find an efficient parallel biconnectivity algorithm. When it comes to programming parallel algorithms it is often the case, more so than with serial algorithms, that there is no single algorithm that performs best in all cases (for example, see [43]). Instead, the best algorithm to use could be sensitive to the computing platform and the properties of the input data. In the PRAM theory of parallel algorithms, the two main performance parameters of an algorithm (assuming synchronous execution and availability of as many processors as needed at each step of the algorithms) are: (i) work – the total number of operations performed by an algorithm, and (ii) depth – its number of steps. In the case of graph algorithms, the performance of a given algorithm may depend not only on the size of the input graph, but other properties of the input as well, such as the ratio of edges to vertices or the diameter of the graph.

Given a platform, this suggests viewing all non-dominated biconnectivity algorithms as a "collage" composed of "patches", where each patch represents a particular biconnectivity algorithm and the whole collage is a complete solution to the biconnectivity problem.

To demonstrate this approach, we evaluate three biconnectivity algorithms on the Explicit Multi-Threaded (XMT)[1] architecture developed at the University of Maryland. Because XMT is an experimental platform, we validate it by comparing it to a better established platform that uses similar silicon area, the NVIDIA GPU. We compare XMT to the GTX 280 (based on the older Tesla architecture) and the GTX 480 (based on the newer Fermi architecture) on significant portions of the biconnectivity algorithms for which optimized CUDA code has already been written by other programmers.

A 1024-core version of XMT, which would use a silicon area between that of one and two quad-core Intel Core i7 920 processors, demonstrated cycle count

[1] Not to be confused with the Cray XMT

speedups of 9x to 33x on biconnectivity relative to a serial biconnectivity algorithm running on the Core i7 920, and further evidence suggests that speedups of 21x to 48x are possible when the investment in the design of the parallel processors matches that of the serial processor. The quantitative contributions of this chapter include

- stronger speedups than in prior parallel biconnectivity studies (9x to 33x vs. $\leq$4x) across a varied family of graphs and
- stronger speedups on parallel connectivity than GPUs of similar or greater area (between 2x and 4.9x faster than the GTX 480).

Since Cong and Bader [36] appears to provide the most relevant prior work, we discuss the significance of the contributions by relating it to their discussion of the challenges they faced with adopting the Tarjan-Vishkin parallel biconnectivity algorithm to a 12-processor SMP. Cong and Bader noted that: (i) the TV algorithm is representative of many parallel algorithms that take drastically different approaches than the sequential algorithm to solve certain problems, and it employs basic parallel primitives such as prefix sum, pointer jumping, list ranking, sorting, connected components, spanning tree, Euler-tour construction and tree computations, as building blocks; (ii) while prior studies demonstrated reasonable parallel speedups for these parallel primitives on SMPs, they left unclear whether an implementation using these techniques achieves good speedup compared with the best sequential implementation because of the cost of parallel overheads encountered (i.e., of resorting to using all these primitives in the first place instead of doing DFS with a stack, per Hopcroft and Tarjan's original serial algorithm); (iii) looking at the whole algorithm rather than at individual primitives allows focusing on algorithmic overhead instead of communication and synchronization overhead; considering one primitive at a time tends to focus on input representations that do not necessarily fit together when used by a single algorithm; converting representations is not trivial, and incurs a real cost in implementations; and (iv) direct implementation of TV on SMPs fell behind the sequential implementation even at 12 processors. Their conclusion was to follow the major steps of TV, but use different approaches for several of the steps, guided by the challenge of reducing the overheads of TV in order to get ahead of the sequential implementation on the 12-processor SMP.

Our goal is different. While reducing overheads remains important, we try to stay much closer to the original PRAM description of TV taking advantage of the scalable XMT platform that was engineered to accommodate that. It is remarkable that XMT manages to get the strong speedups reported with such a relatively modest silicon budget. Also, our implementation demonstrates for the first time the potential advantage of enhancing XMT by supporting in hardware more thread contexts, perhaps through context switching between them. Namely, the significance of the contributions is

- new evidence supporting the practicality of algorithms derived from parallel random-access machine (PRAM) algorithmic theory for speedups and ease-of-programming,
- new evidence demonstrating the advantages of the XMT architecture for the same, and

- the demonstration of a synergistic approach to the design of algorithms and architectures.

The results presented herein are specific to graph connectivity and biconnectivity. Other papers [28, 29] show similar or better speedups for other graph and non-graph problems on XMT. Admittedly, these results do not (and cannot) establish the advantage of XMT for all possible tasks for which one might want to use a general-purpose computer. However, the importance of this work goes a bit beyond just providing one more point of reference. In a similar way that performance, efficiency and effectiveness of a car should not be tested only in first gear, productivity horizons of programming parallel algorithms on a given platform cannot only be studied using elementary algorithms. Graph connectivity problems provide a test case for a proverbial low gear with the more basic graph connectivity algorithms, and higher gear with more advanced graph algorithms for biconnectivity. This and other papers will enable more informed judgment on the overall relative productivity of various approaches. Such documented comparisons will reduce the risk to vendors, allowing them to make better decisions regarding platforms they may want to build.

2.1.1 Related Work

Although no studies of biconnectivity algorithms have previously been published for many-core processors, [36] examines such algorithms on a symmetric multiprocessor (SMP). Also, list ranking and connected components algorithms, two major components of the Tarjan-Vishkin biconnectivity algorithm, are examined in

[11] on an SMP and on the Cray MTA.

Another parallel framework that bears limited resemblance to the many-core platforms evaluated here is MapReduce, which uses large clusters of computers to take advantage of massive parallelism in very large problems. This approach was used for estimating the diameter of large graphs in [97], and the potential to adapt PRAM algorithms into computationally feasible MapReduce algorithms was discussed in [99]. However, the applicability of MapReduce to high-end many-core platforms is not clear and the algorithms examined in this chapter are not necessarily optimal for use in distributed systems such as MapReduce.

2.2 Evaluated Algorithms

Given a graph with n vertices and m edges, the biconnectivity problem can be efficiently solved on a serial computer in $O(n + m)$ time with an algorithm by Hopcroft and Tarjan [90] that performs a depth-first search (DFS) on the graph. This algorithm does not appear to have an efficient, poly-logarithmic-time implementation [139]. It is possible to extract some parallelism from this algorithm using the approach outlined in Exercise 36 in [163], and the resulting algorithm, which we will refer to as parallel DFS (pDFS), runs in $O(n)$ time using $\lceil m/n \rceil + 1$ processors. The main weakness of this algorithm is that the amount of parallelism available depends on the m/n, the "density" of the graph: vertices are processed in serial, and the parallelism available at a vertex is limited by its degree. This algorithm provides little to no parallelism for sparse graphs, where m/n is small.

A more scalable alternative is a biconnectivity algorithm given by Tarjan and Vishkin in [156] that runs in $O(\log n)$ time using $O(n+m)$ processors. The theoretical running time of this algorithm depends only on the size of the graph, not on its structure. This scalability comes at a cost, however: the Tarjan-Vishkin (TV) algorithm performs more operations per vertex and per edge than are required by the serial algorithm or pDFS. Thus, TV may be outperformed by other algorithms in certain situations despite being asymptotically more efficient, especially when running on computer hardware supporting a modest amount of parallelism (e.g. a 4- or 8-core processor).

In these situations, it may be worth modifying TV to be more work efficient. TV is a modular algorithm that calls upon parallel algorithms for simpler problems to do its work. The most significant of these in terms of running time is an algorithm to compute the connected components of an undirected graph (connectivity algorithm). To obtain the complexity bounds in [156], Tarjan and Vishkin used a variation of the Shiloach-Vishkin (SV) connectivity algorithm [145], which runs in $O(\log n)$ time using $O(n+m)$ processors. This algorithm is efficient in asymptotic terms, but its running time has a large constant factor due to the need to revisit vertices and edges multiple times throughout the algorithm. In some cases, it may be beneficial to use another connectivity algorithm, such as breadth-first search (BFS), in place of SV.

In this chapter, we evaluate three biconnectivity algorithms, which we describe below: parallel depth-first search and two versions of the Tarjan-Vishkin algorithm, one using the SV connectivity algorithm and another using BFS in addition to SV.

2.2.1 Input and Output

The input to a biconnectivity algorithm is an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Without loss of generality, we assume that G is connected; if not, the biconnectivity problem can be solved for G by applying a biconnectivity algorithm to each connected component of G. To allow using directly the three biconnectivity algorithms, the input graph is given in the following format:

- Each undirected edge (u, v) in E is represented as a pair of antiparallel directed edges, $u \rightarrow v$ and $v \rightarrow u$. These $2m$ directed edges are stored in an array $edges[2m]$ sorted by the first endpoint.

- For each directed edge $edges_i = u \rightarrow v$ in $edges$, an array $antiparallel[2m]$ stores the index j of its antiparallel copy $edges_j = v \rightarrow u$ such that $antiparallel_i = j$ and $antiparallel_j = i$.

- An array $vertices[n]$ stores indices into the $edges$ array such that, if $vertices[u] = i$, then $edges_i$ is the first edge in $edges$ whose first endpoint is u.

- An array $degrees[n]$, where $degrees_v$ is the degree of vertex v.

Given the data listed above, the algorithm is expected to produce the following output:

- An array $bcc[2m]$ that identifies the biconnected component to which each edge belongs such that for any pair of edges $edges_i$ and $edges_j$, $bcc_i = bcc_j$ if and only if $edges_i$ and $edges_j$ are in the same biconnected component.

- An array $artic\_points[a], 0 \leq a \leq n$ of all the articulation points in G.

- An array $bridges[b], 0 \leq b \leq 2m$ of the indices in *edges* of all the bridges in G.

2.2.2 Parallel Depth-First Search (pDFS)

The intuition behind pDFS is that, although vertices cannot be visited in parallel without potentially violating the order required by a depth-first traversal, edges can be. Initially, all edges are considered *active*. Whenever a vertex is visited in the DFS traversal, all edges leading to that vertex are *canceled*, or removed from the set of active edges. Only active edges are considered when checking for adjacent vertices. Given an input graph in the format described in section 2.2.1, a parallel version of the standard DFS algorithm proceeds as follows:

1. For each vertex v, create a doubly-linked list of its incident edges.

    - Using one thread per vertex, create an array $head[n]$ such that $head_v$ is the index of the first active edge in *edges* originating from v, or -1 if no such edge exists. Initially, all edges are active, so $head_v \Leftarrow vertices_v$ if $degrees_v > 0$ and $head_v \Leftarrow -1$ otherwise.

    - Using one thread per edge, create the arrays $next[2m]$ and $prev[2m]$ such that $next_i$ and $prev_i$ are the indices in *edges* of the next and previous active edges, respectively, that originate from the same vertex as $edges_i$, or -1 if no such edge exists. Initially, $next_i \Leftarrow i + 1$ and $prev_i \Leftarrow i - 1$ with the following exceptions: $prev_i \Leftarrow -1$ if $edges_i$ is the first edge in *edges* that shares its origin and $next_i \Leftarrow -1$ if it is the last such edge.

This list contains all of the active edges originating from v.

2. Define the procedure dfs(v) as follows:

   (a) In parallel, for every edge $edges_i$ originating from v, remove $edges_j = w \rightarrow v$, where $j = antiparallel_i$, from the doubly-linked list in which it is contained:

   - if $prev_j \neq -1$ then $next[prev_j] \Leftarrow next_j$ else $head_w \Leftarrow next_j$
   - if $next_j \neq -1$ then $prev[next_j] \Leftarrow prev_j$

   (b) While $head_v \neq -1$, invoke dfs(w), where $v \rightarrow w = edges[head_v]$.

3. Invoke dfs(r) for some arbitrary vertex r

In order to use DFS to solve the biconnectivity problem, we need two pieces of information about each visited vertex v: its preorder number, pre_v; and the smallest preorder number seen while performing DFS on v and its descendants, low_v. In serial DFS, pre_v can be computed by keeping track of the number of vertices visited so far in a global variable *count*. Every time a new vertex v is visited, pre_v is set to *count*, and then *count* is incremented by 1. The value of low_v is determined by initializing low_v to pre_v upon entering v and updating low_v after (re)visiting a child w as follows: $low_v \Leftarrow min(low_v, low_w)$.

In pDFS, pre_v can be computed the same way because the vertices are still visited serially. However, low_v cannot be because, unlike in serial DFS, visited vertices are never revisited since all edges leading to a visited vertex are always canceled. The key observation that allows us to compute low_v in parallel is the following: the final value of low_v is not needed until returning from the visit to v. Therefore, low_v can be computed just before returning from v as follows: $low_v \Leftarrow min(pre_v, min_{w \in \text{children}(v)}(low_w))$. The remainder of the pDFS algorithm is identical to its serial counterpart.

2.2.3 Tarjan-Vishkin (TV)

The Tarjan-Vishkin biconnectivity algorithm [156] is a PRAM algorithm that was designed as a scalable alternative to DFS. It uses the same principle as the DFS biconnectivity algorithm:3: two edges in a graph are in the same biconnected component if and only if they are on a common simple cycle. However, TV can use any spanning tree, and it performs an Euler tour of the spanning tree to compute information equivalent to that computed in the DFS biconnectivity algorithm. (An Euler tour of a graph is a cycle that visits every vertex in the graph and visits every edge exactly once.) Given an input graph G, TV proceeds as follows:

1. Use a parallel connectivity algorithm to find a spanning tree T of G.

2. Compute an Euler tour of T', where T' is formed by replacing every undirected edge in T with a pair of antiparallel directed edges. This results in a linked list L of edges in T'.

3. Perform list ranking [43] on L to determine the distance of each edge from the end of the Euler tour. Use these distances to determine for each vertex v in T (1) the preorder pre_v of v in T and (2) the size $size_v$ of the subtree of T rooted at v.

4. For each vertex v, compute low_v and $high_v$. These are the lowest and highest preorder numbers, respectively, of the vertices in the set consisting of v, the descendants of v, and all vertices that are adjacent to v or one of its descendants by an edge in $G - T$.

5. Construct an auxiliary graph G', where the vertex set of G' equals the edge set of T and the edge set of G' is constructed as follows, where $p(v)$ denotes the parent of v in T and $v \to w$ denotes an edge in T such that $v = p(w)$:

   - for each edge $\{v, w\}$ in $G - T$, add $\{\{p(v), v\}, \{p(w), w\}\}$ to G' if and only if v and w are unrelated in T and

   - for each edge $v \to w$ in T, add $\{\{p(v), v\}, \{v, w\}\}$ if and only if $low_w < v$ or $high_w \geq v + size_v$.

6. Compute the connected components of G'. This defines an equivalence relation on the edges of T such that a pair of edges in T are in the same connected component of G' if and only if they are in the same biconnected component of G.

7. Extend the equivalence relation on the edges of T to the edges of $G - T$ by defining $\{v, w\}$ equivalent to $\{p(w), w\}$ for each edge $\{v, w\}$ of $G - T$ such that $pre_v < pre_w$.

8. Identify the bridges in G, which are the edges $v \to w$ of T such that low_w and $high_w$ are both descendants of w.

9. Identify the articulation points in G, which are the vertices of G that exist in more than one biconnected component of G.

In steps (1) and (6), any connectivity algorithm may be used without affecting the correctness of the overall biconnectivity algorithm. The version of this algorithm originally described by Tarjan and Vishkin uses the SV connectivity algorithm; we refer to this version simply as the Tarjan-Vishkin (TV) biconnectivity algorithm.

Our implementation of TV on XMT merits some discussion since it is path-breaking effort towards dual validation of the XMT platform and PRAM algorithmics. Originally inspired by PRAM algorithmics and its complexity analysis, the long-term objective of the XMT platform was to revisit the more advanced PRAM algorithms and show that their merit transcends theory. Each PRAM algorithm whose implementation beats the competition for the respective problem it addresses would constitute partial accomplishment of this objective. We are not aware of any prior implementation of a biconnectivity algorithm on XMT or any similar platform. Only the concomitant work [28] represents implementation of an algorithm of similar complexity on XMT.

Implementation The high-level description given in the original paper [156] focuses on achieving complexity results, requiring us to find an implementation that provides good performance. In contrast to [36], we leave the core algorithm as is without reducing its available parallelism, but we choose an implementation that minimizes the amount of work done by the algorithm. In steps (1) and (6), we compact the adjacency list every few iterations as more vertices are discovered to be in the same connected component. In step (3), we accelerate the iterations by choosing faster but more work demanding list ranking algorithms for different iterations ("accelerating cascades", [33]). Also, to save work we transition as many computations as possible from the original input graph to the spanning tree.

The following insights were observed in programming the TV PRAM algorithm. They attest that the practical challenge of effectively programming this theoretical parallel algorithm has a similar flavor to the practice of programming serial algorithms and are much simpler than parallel programming approaches such as [39] with their requirements for decomposition, assignment, orchestration and mapping.

1. Although the same connectivity algorithm is used in steps (1) and (6), it is worthwhile to code two variants of it: one that saves the spanning tree computed by the connectivity algorithm and one that does not. These two versions take different approaches to handling the arbitrary concurrent writes that result when multiple vertices try to hook on the same vertex. The version that saves the spanning tree needs to know which of the writes succeeded in order to know which edge should be added to the spanning tree. On XMT, this is accomplished by performing a prefix sum to memory on a gatekeeper array. On the other hand, if the spanning tree is not needed, then it is not necessary to know which processor succeeded, and this extra work can be avoided, as the connectivity algorithm is in the common CRCW model.

2. The best data structure for storing the spanning tree is the same one as used

for the input graph. This can be derived from the output of step (1) in the following way. Step (1) produces an array T with one entry per edge in the input graph where entry i is 1 if edge i is in the spanning tree and 0 if it is not. The edge list for the spanning tree should be produced by the standard order-preserving PRAM compaction algorithm. The remaining arrays (vertices, degrees, and antiparallel) can then be trivially derived from the corresponding arrays in the input graph. If we use instead a platform-specific optimization (such as prefix sum to registers on XMT) to create the edge list, then we will not be able to derive the necessary tree data structure from the input graph, and it will be difficult to implement the rest of the biconnectivity algorithm (especially the Euler tour) efficiently.

3. Depending on the platform, it may be worthwhile to explicitly relabel the vertices in the graph after rooting the spanning tree by creating a new edge array where the entry corresponding to the edge (u, v) contains the entry $(preorder(u), preorder(v))$. This is an expensive operation up front, but it can save more time in later steps of the algorithm when compared to the alternative of accessing the preorder array each time a relabeled vertex number is needed.

4. When computing global low and high numbers for each vertex, it is necessary to find the minima/maxima of some subarrays of preorder numbers. The PRAM algorithms for doing this first find prefix minima/maxima and suffix minima/maxima relative to subarrays that occur naturally as a result of using

a balanced binary tree over an array representing an Euler tour. It has been observed in [162] that a balanced k-ary tree will be more efficient in practice than a balanced binary tree with the exact k depending on the specific machine at hand. Replacing a binary tree by such a k-ary tree generates different subarrays. This implies finding prefix minima/maxima and suffix minima/maxima relative to these subarrays, and to retrieving low and high numbers from them.

2.2.4 Tarjan-Vishkin with a BFS Spanning Tree (TV-BFS)

For some inputs, better performance can be obtained using a connectivity algorithm with worse asymptotic time bounds but a lower constant factor on work, such as breadth-first search (BFS). BFS naturally lends itself to a parallel implementation, and such an implementation runs in $O(h \log n)$ time and $O(n+m)$ work, where h is the number of layers in the BFS traversal of the graph [50]. The value of h depends on the size and shape of the graph as well as the starting vertex for the traversal, and it can be as large as the diameter of the graph. Notably, for graphs with a diameter that is $O(\log n)$, BFS runs in poly-logarithmic time and thus is an asymptotically efficient parallel algorithm. Even on graphs with somewhat larger diameters, BFS can run more quickly than SV due to its lower constant factor, but for graphs with a large diameter relative to the number of vertices (long, thin graphs), there is too little parallelism available for BFS to be efficient.

In theory, BFS can be used in place of SV for computing both the spanning tree of the original graph and the connected components of the auxiliary graph. However, the most natural representation for the auxiliary graph generated by TV is a list of edges in arbitrary order. This representation is not suitable as input to BFS, which requires the graph to be represented as an adjacency list. Therefore, BFS cannot be used to find the connected components of the auxiliary graph as is. It is possible to convert the edge list produced by TV to an adjacency list, but doing so requires sorting the edge list, which reduces or eliminates the benefit of using BFS in place of SV, so we do not consider it further. If the input to the biconnectivity algorithm is in the proper format, BFS can be used in place of SV to find the spanning tree of the input graph, and we call this variation TV-BFS.

2.3 Evaluated Platforms

We briefly review relevant specifics of the computing platforms on which our experiments are performed. A more detailed overview can be found in [29]. Specifications of the specific configurations evaluated can be found in Table 2.1.

2.3.1 GPUs

Though not originally designed for general-purpose computing, modern graphics processing units (GPUs) are capable of being used as highly parallel computing platforms; this usage of GPUs is referred to as general-purpose GPU (GPGPU). Examples of prevalent GPGPU architectures include Tesla and Fermi, both by NVIDIA. GPUs based on the Tesla architecture are widely used, and there are many parallel applications available to run on them. GPUs based on the Fermi architecture are newer, and there are fewer applications optimized specifically for

	GTX 280	GTX 480	XMT-1024	XMT-2048
Principal Computational Resources				
Cores	240 SP	480 SP	1024 TCU	2048 TCU
Integer Units	240 ALU +MDU	480 ALU +MDU	1024 ALU, 64 MDU	1024 ALU, 64 MDU
(Floating Point Units)[a]	240 FPU, 60 SFU	480 FPU, 60 SFU	64 FPU	64 FPU
On-chip Memory				
Registers	1920KB	1920KB	128KB	256KB
Prefetch Buffers	-	-	32KB	64KB
Regular caches	480KB	1728KB[b]	4104KB	4104KB
Constant cache	240KB	120KB	128KB	128KB
Texture cache	496KB	120KB	-	-

[a] None of the algorithms in this chapter use the floating-point units.
[b] 64KB configurable shared memory/L1 cache per SM and 768KB unified L2 cache

Tab. 2.1: Specifications of the platforms evaluated in the experiments (1 KB = 1024 bytes, SP = Streaming Processor, TCU = Thread Control Unit, ALU = Arithmetic/Logic Unit, MDU = Multiply/Divide Unit, SFU = Special Function Unit)

them, though they are backward compatible with applications written for the Tesla architecture.

The Tesla architecture consists of a number of Streaming Multiprocessors (SMs) connected to a number of DRAM controllers and off-chip memory through an interconnection network. An SM consists of a shared register file, shared memory, constant and instruction caches, special function units (SFUs), and a number of streaming processors (SPs) with integer and floating point ALU pipelines. SFUs are 4-wide vector units that can handle complex floating-point operations.

With respect to biconnectivity algorithms, which do not use floating-point operations, the main advantage of the Fermi architecture over Tesla is the addition of L1 and L2 caches. In Fermi, each SM has 64 KB of memory, which can be split into shared memory and L1 cache in one of two ways: 48 KB shared memory and 16 KB L1 cache or 16 KB shared memory and 48 KB L1 cache [3]. There is also a

768 KB L2 cache shared by all the SMs.

For more information about Tesla, see [112], and for Fermi, see [128].

2.3.2 XMT

The Explicit Multi-Threading (XMT) general-purpose computer architecture is designed to improve single-task completion time. It does so by supporting programs based on Parallel Random-Access Machine (PRAM) algorithms but relaxing the synchrony required by the PRAM model. The XMT programming model differs from the strict PRAM model in two ways:

1. The PRAM model requires specifying the instruction that will be executed by each processor at each point in time, but XMT uses the work-depth methodology [146], which allows the programmer to specify all of the operations that can be performed at each point in time while leaving to the runtime environment the assignment of those operations to processors.

2. The PRAM model requires instructions to be executed in lockstep by all processors at once, but XMT programs follow independence-of-order semantics: parallel sections of code are delimited by spawn-join instruction pairs, and threads only synchronize when they reach the join instruction at the end of the parallel section.

The XMT architecture consists of the following: a number of lightweight cores (TCUs) grouped into clusters, a single core (master TCU or MTCU) with its own local cache, a number of mutually-exclusive cache modules shared by the TCUs and

MTCU, an interconnection network connecting the TCUs to the cache modules, and a number of DRAM controllers connecting the cache modules to off-chip memory. Each TCU has a register file, a program counter, an execution pipeline, and a lightweight ALU. Each TCU also contains prefetch buffers, which can be used by the compiler to prefetch data from memory before it is needed, reducing the length of the sequence of round trips to memory (LSRTM) and improving performance [162]. Each cluster has one or more multiply/divide units (MDUs), floating-point units (FPUs), and a compiler-managed read-only cache, all of which are shared by the TCUs within the cluster. When a parallel section of code is reached, the MTCU broadcasts the instructions in that section to all of the TCUs, and each TCU stores the instructions in a buffer. Virtual threads are assigned to TCUs using a dedicated prefix-sum network.

As noted, a more detailed overview of XMT and the GTX 280 can be found in [29].

2.3.3 Evaluated configurations

The Tesla and Fermi architectures are used in commercially-available products. Therefore, we do not need to establish the practicality of their implementation. We choose the GTX 280 GPU, based on the Tesla architecture, and the GTX 480, based on the Fermi architecture, to represent their respective architectures.

Because XMT is an experimental platform, we establish that XMT is competitive with single-chip multi-cores and many-cores currently available on the market by choosing a configuration of XMT that would use resources comparable to the

GTX 280, the less resource-intensive of the two GPUs evaluated. The GTX 280 uses 576 mm^2 of silicon in 65 nm technology, and according to [29], a 1024-TCU configuration of XMT would use a comparable silicon area. The GTX 480 uses 529 mm^2 of silicon in 40 nm technology and contains more SPs and memory than the GTX 280. Therefore, it can be argued that a 1024-TCU configuration of XMT (XMT-1024) would use at most 529 mm^2 of silicon, and likely less, in 40 nm technology. The 45-nm Intel Core i7 920 quad-core processor, which uses 263 mm^2 of silicon, is half the area of the GTX 480. This places an upper bound on the area of XMT-1024; a lower bound of $576\text{mm}^2 \times \left(\frac{45\text{nm}}{65\text{nm}}\right)^2 = 276\text{mm}^2$ can be found by assuming ideal scaling from 65 nm to 45 nm. In summary, XMT-1024 would use

- about the same area as the GTX 280, while remaining in the same power envelope [102],
- less area than, or at worst the same area as, the GTX 480, and
- an area somewhere between that of one and two Core i7 920 quad-core processors.

To determine the sensitivity of the biconnectivity algorithms to the number of concurrent hardware threads, we also consider a configuration of XMT identical to XMT-1024 with the exception of having twice as many TCUs per cluster; we call this configuration XMT-2048. We do not attempt to argue here that the silicon area of XMT-2048 matches the aforementioned GPUs but merely use it as a reference point.

To collect cycle counts for programs executed on the XMT-1024 and XMT-2048 configurations, we used XMTSim, the cycle-accurate simulator of the XMT architecture. XMTSim and the XMTC compiler are described in [100] and have already been the basis for several publications including [29].

2.4 Experimental Evaluation

2.4.1 Tested Graphs

Data set	Vertices	Edges	Average Degree	Diameter Min.	Diameter Max.
1kv-500ke-complete	1,000	499,500	999.00	1	1
20kv-5me-random	20,000	5,000,000	500.00	2	4
1mv-3me-planar	1,000,002	3,000,000	6.00	333,333	333,333
USA-road-d.LKS	2,758,119	3,397,404	2.46	3,240	6,480
web-Google-con	855,802	4,291,352	10.00	15	30

Tab. 2.2: Properties of the graphs used in the experiments. For graphs whose diameter is not known, lower and upper bounds are given based on the number of layers in a BFS traversal of the graph.

In our experiments, we use three synthetic graphs and two graphs derived from real-world data. Properties of these graphs are given in Table 2.2. The synthetic graphs are as follows:

- 1kv-500ke-complete: The complete graph of 1,000 vertices (and ∼500,000 edges)
- 20kv-5me-random: A graph with 20,000 vertices generated by adding 5 million unique edges between randomly selected pairs of vertices
- 1mv-3me-planar: A maximal planar graph with 1 million vertices generated

layer by layer using the following rules:

- The first layer is the complete graph of three vertices (and three edges). Call this graph G_1 and its three vertices the *external vertices* of G_1.
- Given a graph G_i generated according to these rules with external vertices a, b, and c, generate a new graph G_{i+1} by adding vertices a', b', and c' and the following edges: (a', a), (a', b), (a', b'), (b', b), (b', c), (b', c'), (c', c), (c', a), (c', a'). Vertices a', b', and c' are the external vertices of G_{i+1}.

The real-world graphs are as follows:

- USA-road-d.LKS: A graph of the road network in the Great Lakes region, taken from [2].
- web-Google-con: The largest connected component of the Google web graph of web pages and hyperlinks between them, taken from [4]. This is actually a directed graph, but we convert it to an undirected graph by treating each edge in the original graph as an undirected edge.

Of the five graphs, the first two (the complete graph and the random graph) are of less interest in practical applications of biconnectivity because random graphs are very unlikely to have "interesting" articulation points or bridges (those that divide the graph into large blocks), and complete graphs have none at all. They are included only to show the behavior of the algorithms on dense graphs.

It is possible that larger graphs than the ones listed here may provide more parallelism. However, for the purposes of this chapter, the evaluated graphs are sufficiently large; they provide enough parallelism for the SV connectivity algorithm and the TV biconnectivity algorithm, and the parallelism available to TV-BFS and pDFS depends on the shape of the input graph.

2.4.2 Results for comparing GPUs and XMT

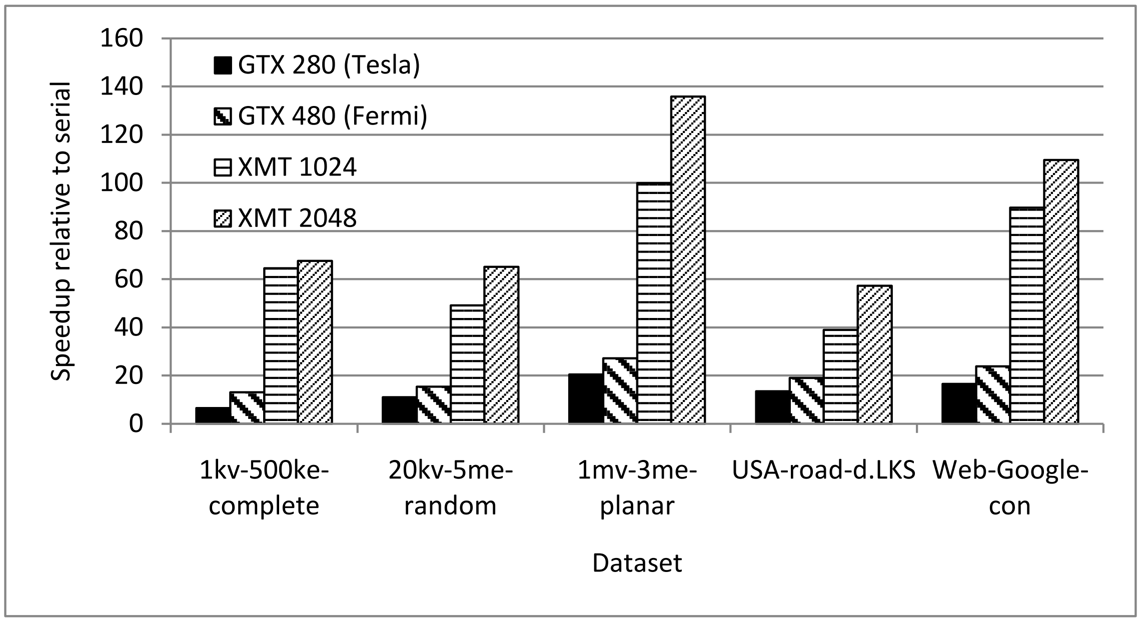


Fig. 2.1: Speedups of the parallel SV connectivity algorithm on the evaluated platforms with respect to serial DFS running on the Core i7 920.

Dataset	GTX 280	GTX 480	XMT 1024	XMT 2048
1kv-500ke-complete	6.60	13.13	64.54	67.56
20kv-5me-random	10.98	15.41	49.09	65.06
1mv-3me-planar	20.45	27.11	99.85	135.79
USA-road-d.LKS	13.45	19.04	38.99	57.35
Web-Google-con	16.58	23.82	89.75	109.53

Tab. 2.3: Speedups of the parallel SV connectivity algorithm on the evaluated platforms with respect to serial DFS running on the Core i7 920.

To support a fair comparison of XMT with the GPUs, we compare against code optimized by others for GPUs. However, at the time of this writing, no such code exists to solve the biconnectivity problem on GPUs. Therefore, we could only test the most time-consuming algorithms used in the Tarjan-Vishkin biconnectivity algorithm, which are logarithmic-time connectivity and BFS.

The 1024-TCU configuration of XMT was already shown to perform better than the GTX 280 on BFS by a factor of 5 in [29], so we will not consider it any further in this chapter. Instead, we focus on logarithmic-time connectivity and compare our implementation of the Shiloach-Vishkin connectivity algorithm on XMT against code written by Soman et al. in [149], the only implementation of graph connectivity on GPUs we are aware of at the time of this writing. As shown in Figure 2.1 and Table 2.3, XMT with 1,024 TCUs outperforms the stronger among the GTX 280 and the GTX 480 by factors ranging between 2.2x and 4x on all input data sets considered.

Soman et al. [148, 149] report that irregular memory access algorithms such as the ones for finding connected components are not a good fit for the GPU computation model, which relies heavily on regularity of memory access; they review both the practical improvements they introduced to the SV algorithm in order to reduce its number of operations, as well as the non-trivial problems they had to overcome in order to fit the GPU model. While our work shares similar features with the former, the flexibility of the XMT architecture freed us from the latter concerns.

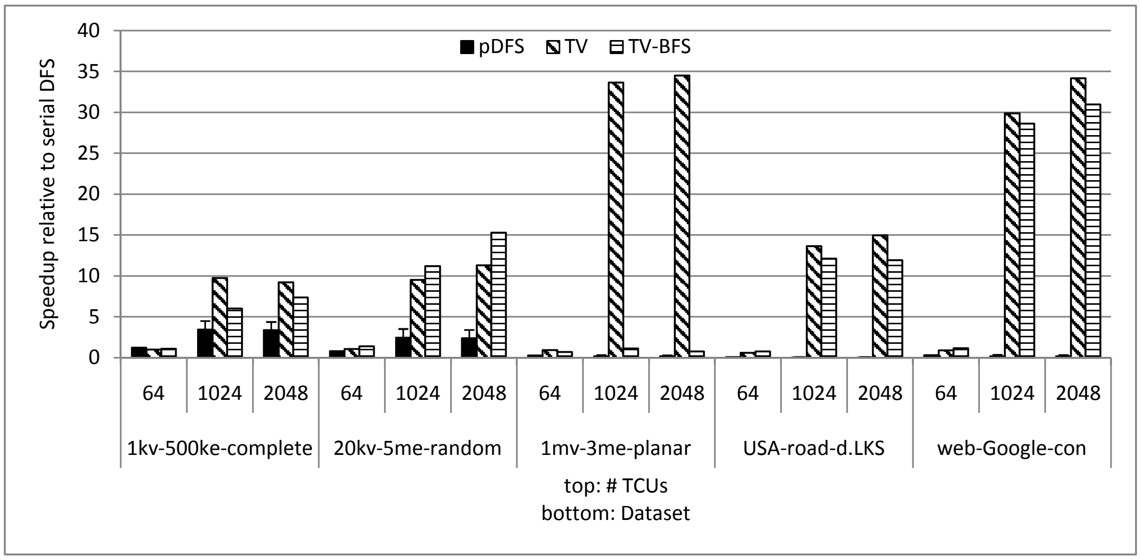
This is one of the main results of this chapter. We also expect XMT to perform competitively in solving the biconnectivity problem. This result should not be generalized much further beyond this; in particular, we do not claim that XMT provides a similar performance advantage over GPUs on applications with regular memory access patterns, for which GPUs were designed.

2.4.3 Biconnectivity Algorithms: Overall Speedups and Comparison of Algorithms

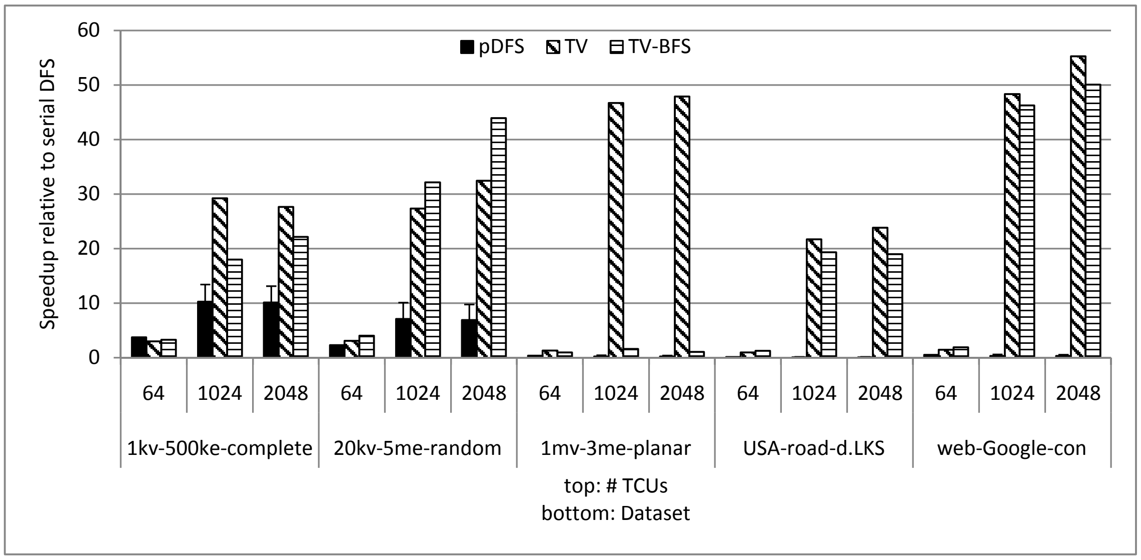
		Speedup vs. Core i7 920			Speedup vs. XMT MTCU		
Data set	TCUs	pDFS	TV	TV-BFS	pDFS	TV	TV-BFS
1kv-500ke-complete	64	1.25	1.01	1.10	3.73	3.02	3.30
	1024	3.45 (4.49)	9.77	6.02	10.31 (13.44)	29.23	18.02
	2048	3.38 (4.39)	9.25	7.40	10.13 (13.13)	27.67	22.15
20kv-5me-random	64	0.81	1.08	1.41	2.31	3.09	4.06
	1024	2.48 (3.53)	9.53	11.21	7.13 (10.13)	27.37	32.19
	2048	2.42 (3.40)	11.30	15.31	6.94 (9.76)	32.44	43.96
1mv-3me-planar	64	0.29	0.97	0.72	0.40	1.35	1.00
	1024	0.19 (0.32)	33.63	1.16	0.26 (0.45)	46.72	1.61
	2048	0.18 (0.30)	34.50	0.79	0.25 (0.42)	47.92	1.10
USA-road-d.LKS	64	0.09	0.63	0.79	0.14	1.00	1.26
	1024	0.05 (0.10)	13.66	12.14	0.09 (0.16)	21.74	19.32
	2048	0.05 (0.10)	14.98	11.95	0.08 (0.15)	23.85	19.01
web-Google-con	64	0.32	0.92	1.19	0.52	1.49	1.93
	1024	0.21 (0.38)	29.89	28.62	0.34 (0.61)	48.32	46.26
	2048	0.20 (0.35)	34.19	30.97	0.32 (0.57)	55.26	50.06

Tab. 2.4: Speedups of the evaluated biconnectivity algorithms on XMT relative to the serial DFS-based Hopcroft-Tarjan biconnectivity algorithm (values in parentheses for pDFS are based on compensated cycle counts). Key: pDFS = parallel DFS, TV = Tarjan-Vishkin, TV-BFS = Tarjan-Vishkin using BFS to find the spanning tree.

Figure 2.2a and the left half of Table 2.4 show the speedups of the three parallel biconnectivity algorithms on XMT with respect to serial DFS on the Core i7 920. We used our implementation of Tarjan's serial DFS algorithm, similar to Cong and Bader, who used theirs. The 64-TCU results were obtained from the Paraleap FPGA [170], and the 1024-TCU and 2048-TCU results were obtained from the XMT simulator. The simulator produces inaccurate cycle counts for serial code because it does not simulate the local cache of the MTCU. The FPGA does have a local cache for the MTCU, so it provides more accurate cycle counts for serial code. The



(a) Speedups vs. the Core i7 920



(b) Speedups vs. the XMT MTCU

Fig. 2.2: Speedups of the evaluated biconnectivity algorithms on XMT relative to the serial DFS-based Hopcroft-Tarjan biconnectivity algorithm. For pDFS, the filled black bar marks a lower bound and the top of the "T" above the bar marks an upper bound. Key: pDFS = parallel DFS, TV = Tarjan-Vishkin, TV-BFS = Tarjan-Vishkin using BFS to find the spanning tree.

following steps were taken to compensate for this discrepancy:

- Cycle counts for the serial versions of the algorithms, which are used as baseline values for the speedups of the parallel algorithms versus the XMT MTCU, were measured on the FPGA.

- For pDFS, which is the only parallel algorithm with a significant serial component evaluated in this chapter, cycle counts for serial sections and parallel sections of execution were measured separately. For the 1024-TCU and 2048-TCU results, we added the serial cycle count from the FPGA to the parallel cycle count from the simulator to obtain a compensated cycle count. This compensated cycle count is lower than the true cycle count because it does not account for the additional delay of the larger interconnection network in the simulated configurations. Thus, it forms a lower bound on the true cycle count. The non-compensated cycle count is larger than the true cycle count and therefore forms an upper bound. We report speedups based on both sets of cycle counts.

We make the following observations about the results and their significance:

- The lack of significant speedups for the 64-TCU configuration is due in part to the parallel algorithms performing more work than the serial algorithm. What make achieving speedups relative to the serial Hopcroft-Tarjan biconnectivity algorithm particularly challenging is that it is very compact, requiring a single visit to each vertex and each edge, as opposed to several visits in the TV-based algorithms.

- The TV algorithm provides speedups of at least 9x relative to the Core i7 and 21x relative to the XMT MTCU on all inputs with 1,024 TCUs. This implies that TV is a good general-purpose parallel biconnectivity algorithm.

- For the 1mv-3me-planar graph, TV provides significantly higher speedups than the other algorithms considered. This is because this graph has a very large diameter and low degree per vertex, which means that there is too little parallelism for pDFS and TV-BFS to exploit. TV is the only algorithm that can provide adequate performance in this case. It is worth noting that this graph is a good representative of many real-world graphs for which one might want solve the biconnectivity problem, so the results for this graph are likely to show the performance of the algorithms in typical usage.

- On the 20kv-5me-random graph, TV-BFS provides the best performance because this graph has a very small diameter. This means that in situations where the graphs being considered are known to be of low diameter, TV-BFS is preferable to TV. Also, for large, dense graphs, with many more edges than vertices, TV-BFS is likely to provide superior performance to TV.

- The presented results assume that given an input it is known which algorithm of the collage to apply. If this is not the case, then a default option would be to use TV, or pDFS if the ratio $|E|/|V|$ is sufficiently large.

- For the data sets considered, all of the algorithms except for pDFS benefit from increasing the number of hardware threads from 1,024 to 2,048 when enough parallelism is available. This is especially noticeable for TV-BFS on the 20kv-5me-random data set and TV on the web-Google-con data set. Biconnectivity algorithms are not very arithmetic-intensive, so additional hardware threads serve primarily to hide memory latency. This technique works as long as there is enough parallelism to keep all of the threads busy and enough bandwidth to DRAM to fulfill the additional requests. This case, where additional hardware threads are needed for latency hiding but not computation, suggests that it would be worthwhile to augment the XMT architecture with support for thread context switching, where each TCU stores two or more sets of thread state and switches contexts whenever a memory request blocks. The silicon area required to support context switching would be less than that required to increase the number of TCUs as functional units would not need to be duplicated.

- The speedups relative to the XMT MTCU, as shown in Figure 2.2b and the right half of Table 2.4, are between 1.3x and 3x larger than the corresponding speedups relative to the Core i7 920. Although we base our primary speedup claims on the Core i7, the speedups relative to the XMT MTCU are in a sense more relevant, as the MTCU reflects the same technology and engineering effort as the rest of the XMT architecture and we expect them to scale up at the same rate as they are further developed. This suggests that speedups of 21x to 48x could be obtained if XMT were brought up to industry grade on par with the Core i7.

The reported speedups are made possible by support in the XMT architecture for the efficient execution of programs with fine-grained, irregular parallelism. The XMT implementation of TV consists of many short parallel sections of code due to the synchronous nature of the algorithm. The instruction broadcast and prefix-sum network provide a low overhead for entering parallel sections and starting threads within a section, which allows even short threads to be profitable. Also, there are many indirect accesses to memory that, depending on the structure of the graph, may exhibit poor locality of reference. The TV algorithm provides a large amount of parallelism (one thread per vertex or per edge), which allows many memory requests to be issued in parallel, reducing the impact of the latency of any one request.

2.5 Discussion

The discussion below suggests that contrary to common practice (or belief) there appears to be no principled need to compromise ease-of-programming in order to get strong speedups.

- The new NSF/IEEE-TCPP curriculum [5] views the PRAM model as overly simplistic. In contrast, using the XMT architecture we were able to obtain stronger speed-ups than in prior parallel biconnectivity studies, 9x to 33x through direct implementation of PRAM algorithms versus the previously reported of up to 4x in [36]. Interestingly, [36] was also driven by PRAM algorithms, though they had to work around an SMP architecture.
- Another example is the BFS algorithm. [5] also suggests teaching BFS. The recent paper [133] reported that none of the 42 students who took a joint

UIUC/UMD parallel algorithms/programming class in Fall 2010 was able to get any speedups using OpenMP on an 8-processor SMP machine, while the speedups on a 64-processor XMT hardware, which uses at most 1/4 of the silicon area of the 8-processor machine, ranged between 7x and 25x. BFS is an example where OpenMP programming was not substantially different than XMT programming, but the XMT architecture allowed the speedup difference. See also the comment on bandwidth later in this section.

- The TCPP curriculum does not include any of the poly-logarithmic PRAM graph algorithms. However, this chapter shows that they provide robust speedups on XMT that are unmatched by any of the graph algorithms the curriculum lists.

- Speedup problems with OpenMP are not new (for example, see [67]). The reason for comparing them with XMT above is that ease of programming is a priority for both. A short comparison on ease of programming follows. Teaching of XMT programming was done in parallel algorithms courses without any introduction to architecture and only a 20-minute introduction to XMT programming [133]. In contrast, the TCPP curriculum ranks parallel algorithms as third in priority of teaching after architecture and programming. Introduction of OpenMP is typically tied to architecture concepts such as the memory hierarchy.

- Interestingly, [29] and the current chapter show that XMT is also competitive on performance with GPUs, which are performance-driven but are much more challenging to program effectively, as demonstrated in the comparison with [148, 149] in Section 4.2. The starting point of this research was that the SV parallel connectivity was given as a programming assignment in parallel algorithm courses at the University of Maryland and was even solved by a couple of 10th graders in a course offered at a nearby high school. While our work reduced the total of operations (without the changing the basic work complexity of SV), our biggest effort was the extension beyond connectivity to biconnectivity. For this reason, the fact that no GPU biconnectivity implementation has been reported in spite of the mushrooming of GPU research is perhaps another demonstration of the practicality of XMT programming relative to GPU programming. Personal communication with the authors of [149] regarding the wording of their reference to possible use of their GPU connectivity program in a biconnectivity one confirmed that it was not meant to pass judgment on the relative difficulty of the two programs.

- In contrast to the implementation of biconnectivity for SMPs by Cong and Bader [36], which consists of over 5,800 lines of C code, our implementation for XMT only requires about 1,600 lines of code. Also, the effort required to tune and debug our implementation was comparable to that required for a serial program of similar size. In fact, serial debugging tools (GDB and Valgrind) were sufficient to catch and fix nearly all bugs in our parallel XMTC code.

- Much of the effort in writing the parallel biconnectivity code was in writing and tuning functions to perform basic tasks in parallel such as prefix sum, range-minimum queries, finding a spanning tree of a graph, and computing the preorder numbering of the nodes in a tree. These basic tasks are more general than biconnectivity and can be separated into a standalone library for reuse in other software projects.

- Using the above library, we have given biconnectivity as an optional programming assignment to graduate classes since the Spring 2012 semester. Providing the library to the students reduced the complexity of the task to that of understanding how the PRAM algorithm works and seeing how the building blocks provided by the library can be assembled to construct a working implementation.

For placing this debate in historical context, recall that claims that the main reason that parallel machines are difficult to program is that the bandwidth between processors and memories is so limited are not new, as formally demonstrated in [114, 166]. [21] suggested that: 1. Machine manufacturers see the cost benefit of lowering performance of interconnects, but grossly underestimate the programming difficulties and the high software development costs implied. 2. Their exclusive focus on runtime benchmarks misses critical costs, including: (i) the time to write the code, and (ii) the time to port the code to different distributions of data or to different machines that require different distribution of data. The XMT platform [13, 14] was finally able to demonstrate an affordable prototype providing the bandwidth that the 1994 paper [21] sought, but using today's technology.

Competition among hardware vendors in the desktop computing space has greatly diminished in recent years. Yet, the adoption of the few industry many-core solutions falls far behind serial platforms, which is a cause of extra concern. As believers in the eventual power of ideas, we are doing our best with XMT to keep some intellectual competition alive in spite of the huge funding gap with industry.

2.6 Conclusion

Of the biconnectivity algorithms evaluated, the logarithmic-time Tarjan-Vishkin algorithm, derived using PRAM algorithmic theory, provided the best performance overall. Of the parallel computing platforms evaluated, the XMT platform, designed with PRAM algorithms in mind, provided the best performance. These two facts demonstrate that with the proper many-core architecture, the relative simplicity of the PRAM can, perhaps surprisingly, be combined with the best performance.

More generally, this work provides another example that should help void PRAM criticism and address asymptotic analysis criticism. Criticism of the PRAM model has sometime been confused with criticism of the constants hidden by asymptotic analysis. In our opinion the XMT platform, which was originally inspired by PRAM algorithmics, and the performance it facilitated have voided much of the criticism on the PRAM model. However, one has to be a bit more careful with understanding the issue of constant factors. In the same way that theoretical papers on serial algorithms and their asymptotic analysis were often followed by separate efforts minimizing constant factors, the current work complements the original theory PRAM papers by reducing them to practice with respect to XMT, accounting for constant factors and concrete speedups. This often amounts to first modifying a published PRAM algorithm to another PRAM algorithm or other supporting data structures whose constant factors are better, which is, in fact, where the intellectual merit of this work lies; only then the revised PRAM algorithm is programmed for the XMT platform, which turns out to be a rather simple task. For biconnectivity, even optimizing LSRTM for performance tuning of XMT (per [162] as noted earlier), was adequately picked up by the compiler.

Chapter 3: Graph triconnectivity

3.1 Introduction

A k-*(vertex-)cut* of an undirected graph is a set of k vertices whose removal results in the graph being disconnected. An undirected graph is k-*(vertex-)connected* if it has no cut of size $k-1$ or less. A 1-connected graph is said to be *connected*, a 2-connected graph *biconnected*, and a 3-connected graph *triconnected*. A *biconnected component* of a graph G is a maximal biconnected subgraph of G.

The triconnected components of a graph G are defined in [91]. A 2-cut is also called a *separation pair*. Briefly, assuming that G is biconnected, it is repeatedly *split* into two subgraphs with respect to one of its separation pairs. Each time G is split using a pair $\{u, v\}$, an edge (u, v), called a *virtual edge*, is added to both subgraphs. When no more splitting is possible, the resulting graphs (called *split components*) are of one of three types: triconnected graphs, triangles (rings of 3 vertices), and triple bonds (multigraphs consisting of 3 parallel edges). Then, split components of the same type that share a common virtual edge are *merged* (the inverse of splitting); triangles are merged to (recursively) form polygons (rings), and triple bonds are merged to (recursively) form n-bonds (with n parallel edges). The (unique) graphs that result are called the *triconnected components of G*. If all the triconnected components of a graph are merged together, the result is the original graph. The triconnected components of a general graph G are the triconnected components of its biconnected components.

The triconnected components of a graph provide useful information about the graph, such as the resilience of an underlying network to defects. The *SPQR-tree* of a graph G, with a vertex for every triconnected component of G and an edge between any two components that share a virtual edge, can be used to represent the planar embeddings of a graph; this can be used, for instance, to test whether a graph would remain planar after adding a given edge [15].

In this chapter, we evaluate an implementation of an efficient PRAM triconnectivity algorithm on the experimental Explicit Multi-Threading (XMT) architecture, developed at the University of Maryland to efficiently support PRAM-like programming and shown [27, 52] to support some advanced graph algorithms. Nevertheless, the current work represents the most complex algorithm that has been tested on XMT, and uses quite a few building blocks, which are simpler PRAM algorithms. The speedups obtained (up to 129x) and their scalability provide points of reference for comparing XMT to other approaches beyond that of a simple benchmark kernel. The importance of going beyond simple kernels and toy problems for comparing architectures has long been recognized in the SPEC benchmarks and in the standard text [84]. The source code for our implementation is available at `http://www.umiacs.umd.edu/users/vishkin/XMT/OPEN_SOURCE_GRAPH_ALGS/`.

3.2 Triconnectivity Algorithms

For a graph with n vertices and m edges, an efficient $O(n+m)$-time **serial algorithm** to determine its triconnected components based on depth-first search is given by Hopcroft and Tarjan [91]. This algorithm was implemented and tested by Gutwenger and Mutzel [82] and was made available as part of [32]. Neither we nor the authors of [82] are aware of any other publicly-available implementation of a linear- (or near-linear-) work triconnected components algorithm, either serial or parallel.

Several **parallel triconnectivity algorithms** have been described. Miller and Ramachandran (MR) [119] proposed an efficient algorithm that runs in $O(\log^2 n)$ time while performing $O(m \log^2 n)$ work on a CRCW PRAM and is based on finding an open ear decomposition [115] of the input graph. Their algorithm has two parts, one to find the nontrivial candidate sets of the input graph (sets of vertices such that any two vertices in a set are a separation pair) and one to split the graph into its triconnected components based on its nontrivial candidate sets. An earlier algorithm for finding nontrivial candidate sets by Ramachandran and Vishkin (RV) [138] required only $O(\log n)$ time while still performing $O(m \log^2 n)$ work.

Our **implementation on XMT** uses the RV algorithm to find the nontrivial candidate sets and the MR algorithm to split the graph into its triconnected components. The most significant contributor to its runtime is the need to make $O(\log n)$ calls to a connected components routine. We use the Shiloach-Vishkin (SV) [145] connectivity algorithm, which runs in $O(\log n)$ time and $O(m \log n)$ work on a CRCW PRAM. Each call to SV actually computes the connected components of multiple subgraphs derived from the input graph. This combining of inputs is done to permit flattening of memory allocation and parallelism to improve performance. Some care is required to keep the subgraphs from interacting; details are omitted due to space limitations.

3.3 The XMT Many-Core Platform

The Explicit Multi-Threading (XMT) general-purpose computer architecture [164] is designed to improve single-task completion time. It does so by supporting programs based on Parallel Random-Access Machine (PRAM) algorithms but relaxing the synchrony required by the PRAM model. A key enhancement of XMT is providing hardware support for things that are done in software in other architectures.

The XMT architecture consists of the following: a number of lightweight cores (thread control units or TCUs) grouped into clusters, a single core (master TCU or MTCU) with its own local cache, a number of mutually-exclusive cache modules shared by the TCUs and MTCU, an interconnection network connecting the TCUs to the cache modules, and a number of DRAM controllers connecting the cache modules to off-chip memory. Each TCU has a register file, a program counter, an execution pipeline, a lightweight ALU, and prefetch buffers. Each cluster has one or more multiply/divide units (MDUs) and a compiler-managed read-only cache, all of which are shared by the TCUs within the cluster. When a parallel section of code is reached, the MTCU broadcasts the instructions in that section to all of the TCUs, and each TCU stores the instructions in a buffer. Virtual threads are dynamically assigned to TCUs using a dedicated prefix-sum network. A more detailed overview of XMT can be found in [29].

3.4 Experimental Results

We measured the speedups of our parallel triconnectivity implementation by comparing it to the best available serial implementation [32] running on an Intel Core i7 920 CPU. We used three types of graphs in our comparison.

- Random graphs are generated by randomly selecting $|E|$ edges with uniform probability.
- Planar3 graphs are planar graphs generated level by level. The first level is a triangle of three vertices. Each succeeding level consists of three vertices, an edge between each pair of vertices in that level, and an edge from each vertex in that level to a distinct vertex in the previous level.
- Ladder graphs are similar to planar3 graphs, but with only two vertices per level (they are also planar).

The random graphs are used as representatives of dense graphs, in contrast to the sparse planar3 and ladder graphs. The input graphs we used are listed in Table 3.1.

We ran our parallel code on two versions of the XMT architecture: (i) a 64-core FPGA prototype whose cycle counts reflect those of an 800-MHz ASIC [170] and

Data set	**Vertices** (n)	**Edges** (m)	**Sep. pairs** (s)
Random-10K	10K	3000K	0
Random-20K	20K	5000K	0
Planar3-1000K	1000K	3000K	0
Ladder-20K	20K	30K	10K
Ladder-100K	100K	150K	50K
Ladder-1000K	1000K	1500K	500K

Tab. 3.1: Properties of the graphs used in the experiments.

(ii) a 1024-core configuration simulated on the cycle-accurate XMTSim simulator. The results are shown in Figure 3.1.

It is shown in [29] that the latter configuration would use about the same area as an NVIDIA GTX 280 GPU; [103] showed similar power using a similar clock speed. Parameters such as the number and sizes of cache modules, number of clusters, and latencies of pipelines were calibrated according to the number of TCUs.

Next, we discuss the relationship between the FPGA results and a simulated 64-TCU configuration of XMT. As [100] notes, currently, only on-chip components are simulated in a cycle-accurate manner, but DRAM is modeled as simple latency with controlled rate of memory requests. However, we expect the simulator results to be the correct representative of the capability of XMT as its assumed bandwidth is realistic for state-of-the-art industry-grade processors. The DRAM controller used in the FPGA is not representative of controllers that would be used in an industry-grade implementation of XMT. Also, DRAM bandwidth has increased since the FPGA prototype was built (e.g., transition from DDR2 to DDR3); this is in line with [8] that reports a trend of drastic improvement in bandwidth over latency (by a factor of 300 over three decades). Specifically, the simulator ran between 14%

and 52% faster than the FPGA, with the larger gaps for larger input graphs. The graph with the highest speedup, Random-20K, ran 16% faster. These gaps are due to the behavior of the DRAM controller in the FPGA and in the simulator, as can be shown by reducing the clock frequency of the simulated DRAM controller, and thus its peak bandwidth, by 33%. In this case, the situation is reversed, and the simulator becomes between 10% slower and 5% faster than the FPGA, with a 20% to 31% drop in performance relative to the original configuration. Intuitively, this makes sense because larger graphs are less likely to have a working set that fits in cache, and bandwidth to DRAM becomes a dominant factor. The transition from DDR2 to DDR3 itself would double this bandwidth [84], eliminating the gaps.

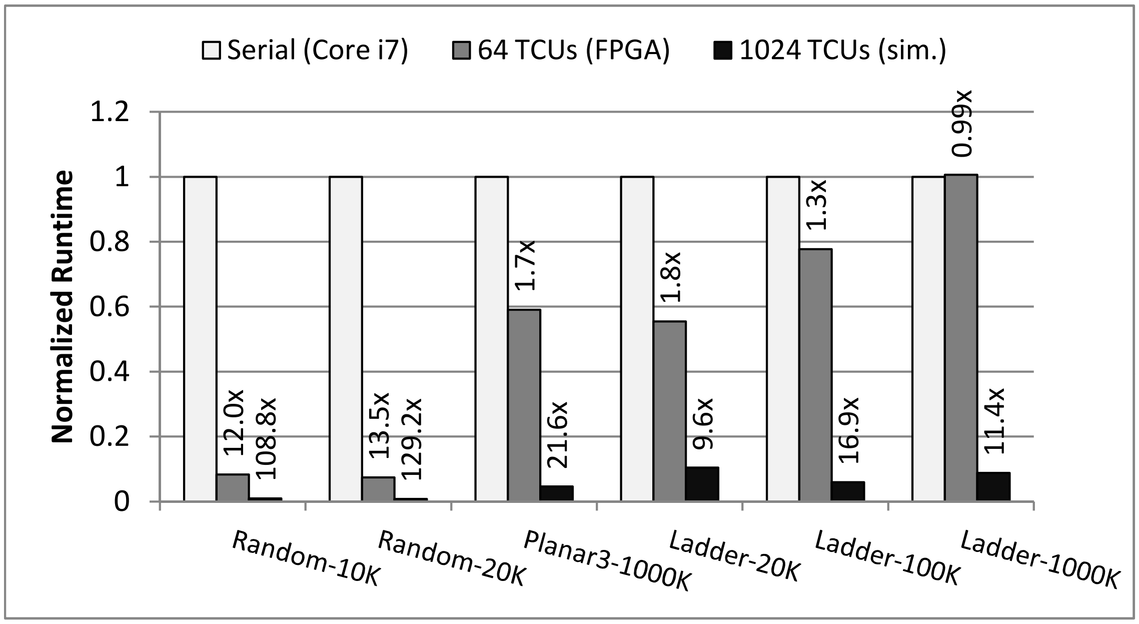


Fig. 3.1: Performance of the parallel triconnectivity algorithm; numbers above bars represent speedup relative to serial (Core i7).

The parallel algorithm scales well, with a 9x to 13x improvement in speedup in all but one case when moving from 64 TCUs to 1024 TCUs. The weaker improvement (5x) for the Ladder-20K graph is because it is too small, providing insufficient

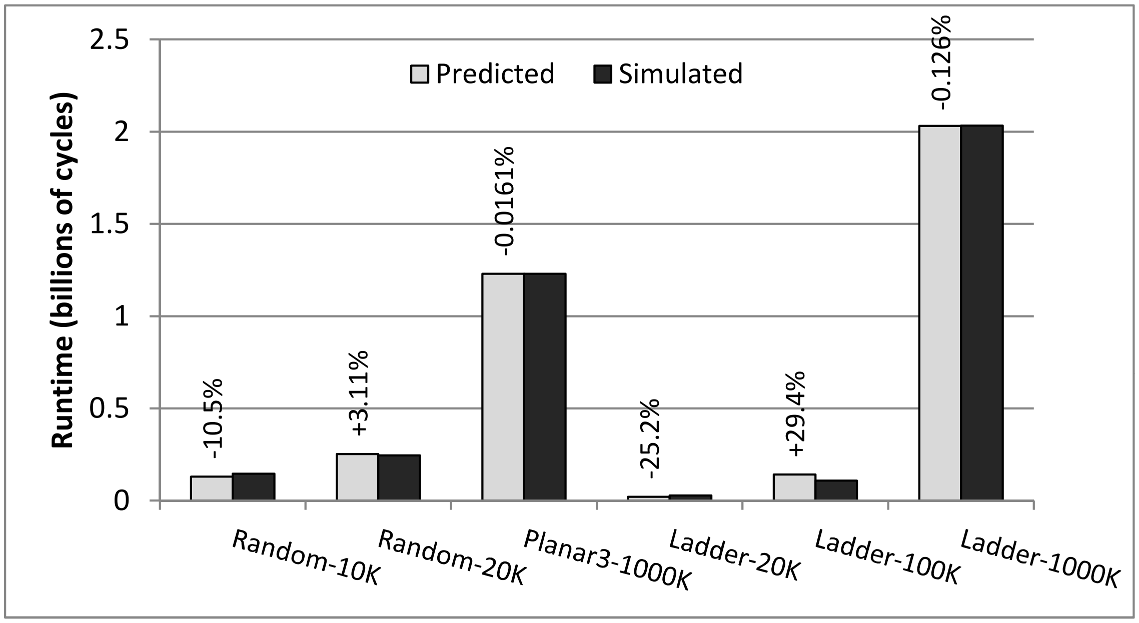


Fig. 3.2: Predicted vs. observed runtime for the 1024-TCU configuration using Equation 3.1; numbers above bars represent percent error of prediction relative to simulation.

parallelism to take full advantage of the additional TCUs.

The lower speedups on sparse graphs (Planar3- and Ladder-) are due to the parallel algorithm performing more work than the serial algorithm. Unlike the serial algorithm, which revisits the input graph a constant number of times, the parallel algorithm revisits the input $O(\log^2 n)$ times, with the coefficient in the final (splitting) stage of the algorithm depending on the number of separation pairs in the input graph. The runtime of the parallel triconnectivity algorithm running under the 1024-TCU configuration of XMT, in cycles, for a graph with n vertices, m edges, and s separation pairs can be approximated by

$$T(n, m, s) = (2.38n + 0.238m + 4.75s) \log^2 n \tag{3.1}$$

where the base of the logarithm is 2 and lower-order terms have been neglected;

see Figure 3.2. Note that there are only 3 degrees of freedom (# data points - # constants in equation), so more data points will be required to verify that the equation holds in general; this is work in progress.

3.5 Future Work

An alternate algorithm to the one tested here is the algorithm of [68]. It runs in $O(\log n)$ time using $O(m \log\log n)$ work (or $O(m \log n)$ work if implemented using the SV connected components algorithm) and is based on finding the biconnected components of a "local replacement graph" derived from the *st*-numbering of the input graph. It would be worthwhile to see if this algorithm provides better performance on large graphs. However, we have not yet implemented it because the implementation is more involved and builds upon the work presented here.

3.6 Conclusion

The XMT architecture provides good performance and scalability on triconnectivity, which relies on subroutines such as connected components. The fact that XMT performs well on this complex problem demonstrates that (i) the previously demonstrated advantages of XMT are not limited to small kernels and (ii) the advantage of XMT on small kernels extends to larger problems that rely on those kernels. [27] demonstrated strong XMT speedups of up to 108x for max-flow (versus the best serial implementation). The recent paper [52] reported XMT speedups of up to 33x, and the question this raised was whether connectivity problems are less amenable to parallel speedups than other graph problems, such as max-flow. The strong speedups of up to 129x for triconnectivity reported here suggests it was perhaps the compactness of the serial biconnectivity algorithm of [90] that needs only a single visit of the vertices and edges that limited speedups, rather than a more general problem.

Chapter 4: Burrows-Wheeler (BW) compression

4.1 Introduction

A *lossless compression function* is an invertible function $C(\cdot)$ that accepts as input a string S of length n over some alphabet Σ and returns a string of length $\Theta(n)$ over some alphabet Σ' where, on average, fewer bits are required to represent $C(S)$ than S. A *lossless compression algorithm* for a given lossless compression function is an algorithm that accepts S as input and produces $C(S)$ as output; the corresponding *lossless decompression algorithm* accepts $C(S)$ for some S as input and produces S as output.

In [25], Burrows and Wheeler describe their eponymous lossless compression algorithm and corresponding decompression algorithm; it has been shown [6, 7] to be among the best such algorithms, and its operation is reviewed in this chapter. The *Burrows-Wheeler (BW) Compression problem* is to compute the lossless compression function defined by the algorithm of [25], and the *Burrows-Wheeler (BW) Decompression problem* is to compute its inverse. The algorithm of [25] solves the BW Compression problem in $O(n \log^2 n)$ serial time and solves BW Decompression problem in $O(n)$ serial time. Later work reduced a critical step of the compression algorithm to the problem of computing the suffix array of S, for which linear-time algorithms are now known, so both problems can now be solved in $O(n)$ optimal serial time.

We propose an $O(\log^2 n)$-time, $O(n)$-work PRAM algorithm for solving the BW Compression problem and a $O(\log n)$-time, $O(n)$-work PRAM algorithm for solving the BW Decompression problem. These algorithms appear to be the first polylogarithmic-time work-optimal parallel algorithms for any standard lossless compression scheme.

We implement our parallel algorithm and experimentally validate it. A parallel-algorithmic approach to BW compression may not have been seriously considered in the past because the fine-grained parallelism provided by such an approach is difficult for existing computing hardware to exploit. However, the Explicit Multi-Threading (XMT)[1] architecture developed at the University of Maryland was designed specifically to provide good performance on such algorithms. Using our parallel algorithm in conjunction with XMT, we obtain speedups of up to 25x for compression and 13x for decompression for small inputs (say, up to 1MB) where no speedup was possible before. This is especially important for real-time applications, where single-task completion time is more important than throughput.

In passing, we note that commonly-used compression programs divide the input into uniformly-sized blocks and apply a serial implementation of BW compression to each block independently. The blocks can be compressed in parallel; however, this does not solve the BW Compression problem for the original input and thus is not a parallel algorithm for solving it. It is worth noting that our

[1] Not to be confused with the Cray XMT

parallel-algorithmic approach is orthogonal to the foregoing block-based approach, and the two approaches could conceivably be combined to obtain better speedups than either alone.

One application where our implementation shows a distinct advantage over existing compression libraries is in compressing data that is sent over a network. In warehouse-scale computers such as those found in data warehouses, the bandwidth available between various pairs of nodes can be extremely different, and for pairs where the bandwidth is low can be debilitating [84]. A way to mitigate this is to compress data before it is transmitted over the network and decompress it on the other side. This approach is taken, for example, by Google via their Snappy [78] library. The goal of Snappy is to compress data very quickly, even at the expense of less compression, providing a larger increase in effective network bandwidth than other libraries for all but very low-bandwidth networks. As shown in Section 4.5.2, our implementation outperforms Snappy and similar libraries for point-to-point bandwidths of up to 1 Gbps.

This technical report augments the theory results of [51] with experimental speedups. For an extended description of the algorithms, please see the companion report [51].

4.1.1 Related Work

There are applications where BW compression would be useful but is not currently used because of performance. One such application is JPEG image compression. JPEG compression consists of a lossy compression stage followed by a lossless stage. The work [173] considered replacing the currently-used lossless stage with the BW compression algorithm. For high-quality compression of "real-world" images such as photographs, this yielded up to a 10% improvement, and for the compression of "synthetic" images such as company logos, the improvement was up to 30%. The author cites execution time as the main deficiency of this approach.

A commonly-used, serial implementation of the block-based approach noted above is bzip2 [144]; the algorithm it applies to each block is based on the original BW compression algorithm of [25]. There are also variants of bzip2, such as pipeline bzip [73], that compress multiple blocks simultaneously. However, these variants do not achieve speedup on single blocks while our approach does. There exists at least one implementation of a linear time serial algorithm for BW compression, bwtzip [108]. However, bwtzip is a research-grade implementation that emphasizes modularity over performance, unlike the focus of this chapter.

The survey paper [61] articulates some of the issues involved in parallelizing BW for a GPU; decompression is not discussed. The author gives an outline of an approach for making some parts of the algorithm parallel and claims that the remaining parts would not work well on GPUs due to exhibiting poor locality. [135] reports such parallelization, and indeed was unable to demonstrate a speedup for compression using the GPU, instead obtaining a slowdown of 2.78x. Note that our results reflect a speedup of 70x over [135]. Parallelization of decompression was left as future work, and no speedups or slowdowns are reported. Furthermore, no asymptotic complexity analysis is given, and our own analysis shows their algorithm to be non-work-optimal. To their credit, they appear to be the first to formulate

MTF encoding (Section 4.2.1) in terms of a binary associative operator. However, two challenges, (i) work-optimal parallelization of BW and (ii) feasibility of speedups on buildable hardware, remained unmet.

A parallel algorithm for Huffman decoding is given in [105]. However, the algorithm is not analyzed therein as a PRAM algorithm, and its worst case run time is $O(n)$. Our PRAM algorithm for Huffman decoding runs in $O(\log n)$ time.

The rest of the chapter is organized as follows: Section 4.2 gives an overview of the serial BW compression and decompression algorithms and Section 4.3 describes our parallel algorithms for the same along with their complexity analysis, ending the theoretical part of the chapter. The remainder of the chapter is devoted to experimental validation of the algorithms. Section 4.4 describes the experimental comparison to bzip2, Section 4.5 contains a discussion of the results we obtained, and Section 4.6 concludes.

4.2 Serial Algorithm

In their original paper, Burrows and Wheeler [25] describe a lossless data compression algorithm consisting of three stages in the following order: a reversible block-sorting transform (BST)[2], move-to-front (MTF) encoding, and Huffman coding. The corresponding decompression algorithm performs the inverses of these stages in reverse order: Huffman decoding, MTF decoding and inverse BST (IBST). See Figure 4.1.

[2] This transform is also known as the Burrows-Wheeler Transform (BWT). We refrain from using this name to avoid confusion with the similarly-named Burrows-Wheeler compression algorithm which employs it as a stage.

Given an input string of length n, their original decompression algorithm runs in $O(n)$ serial time, as do all stages of their compression algorithm except the (forward) BST, which requires $O(n \log^2 n)$ serial time [143]. More recently, linear-time serial algorithms [98, 130] have been developed to compute suffix arrays, and the problem of finding the BST of a string can be reduced to that of computing its suffix array, so Burrows-Wheeler (BW) compression and decompression can be performed in $O(n)$ serial time. The linear-time BST algorithms are relatively involved, so we refrain from describing them here and instead refer interested readers to the cited papers.

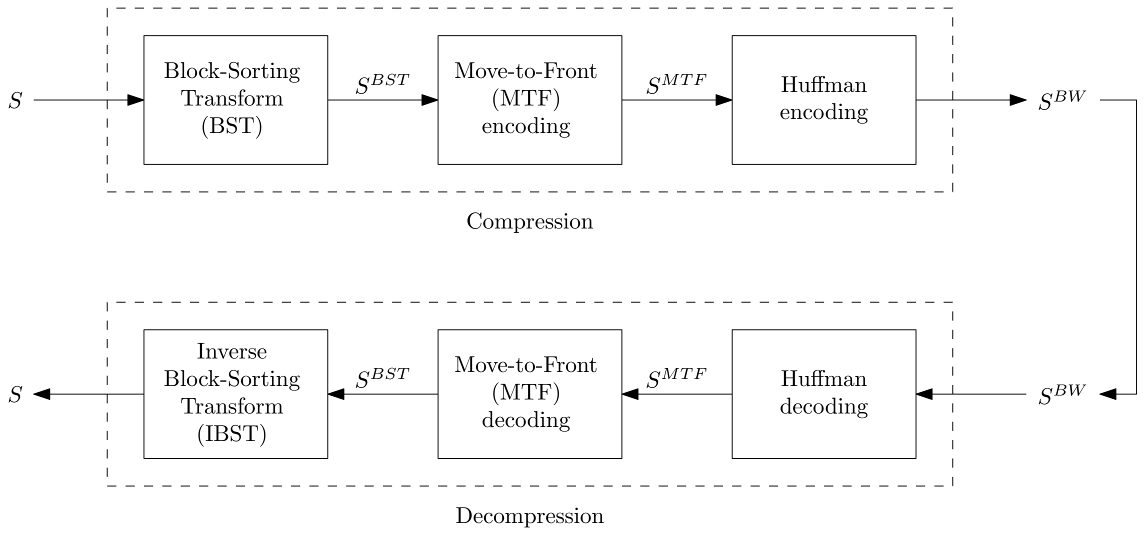

S
Block-Sorting Transform (BST)
S^{BST}
Move-to-Front (MTF) encoding
S^{MTF}
Huffman encoding
S^{BW}
Compression
S
Inverse Block-Sorting Transform (IBST)
S^{BST}
Move-to-Front (MTF) decoding
S^{MTF}
Huffman decoding
S^{BW}
Decompression

Fig. 4.1: Stages of BW compression and decompression.

4.2.1 Compression

Given a string S of length n from an alphabet Σ, where $|\Sigma|$ is constant with respect to n, the compression algorithm proceeds in three stages as follows.

The BST stage takes S as input and produces as its output S^{BST}, a permutation of S. S^{BST} is formed by making a list of all the rotations of S (each of which is also a string of length n), sorting the list of rotations lexicographically, and outputting the last character of each rotation in the sorted list starting with the first. See Figure 4.2. The BST has two properties that make it useful for lossless compression: (1) it has an inverse and (2) S^{BST} tends to have many occurrences of any given character in close proximity, even when S does not. Property (1) ensures that the decompressor can reconstruct S given only S^{BST} and Property (2) allows the following stages to work effectively.

$$\text{banana\$} \xrightarrow{\text{rotate}} \begin{array}{l} \text{banana\$} \\ \text{anana\$b} \\ \text{nana\$ba} \\ \text{ana\$ban} \\ \text{na\$bana} \\ \text{a\$banan} \\ \text{\$banana} \end{array} \xrightarrow{\text{sort}} \underbrace{\begin{array}{l|l|} \text{\$banan} & \text{a} \\ \text{a\$bana} & \text{n} \\ \text{ana\$ba} & \text{n} \\ \text{anana\$} & \text{b} \\ \text{banana} & \text{\$} \\ \text{na\$ban} & \text{a} \\ \text{nana\$b} & \text{a} \end{array}}_{M} \xrightarrow{\text{output}} \text{annb\$aa}$$

Fig. 4.2: BST of the string "banana\$". The sorted list labeled M can be viewed as a matrix of characters.

The critical step in the BST algorithm is the sorting of the list of rotations of S. The BST algorithm given in [25] is actually a combination of two algorithms: direct comparison and doubling [143]. The direct comparison algorithm sorts the list of rotations of S using a comparison-based sorting algorithm that compares rotations in the list character-by-character. Therefore, it requires $O(n \log n)$ string comparisons, and since comparing two strings of length n requires $O(n)$ comparisons in the worst case, the direct comparison algorithm has a worst-case running time of $O(n^2 \log n)$.

The doubling algorithm works in $O(\log n)$ iterations. Each iteration applies comparison-based sorting to the current list of rotations of S as in the direct comparison algorithm. However, each comparison is limited to the first d characters of the rotations being compared, with $d \geq 2$ a constant, so the list will not be completely sorted if two rotations begin with the same sequence of d characters. Afterwards, a new string S' is constructed over the alphabet $[0, n-1]$ such that S_i, $0 \leq i < n$, is the rank of the ith rotation in the partially-sorted list. To complete the iteration, S is replaced by S'. During the next iteration, up to d character comparisons are made again as in the first iteration, but now each character in S' gives the rank of d consecutive characters in S, so the character comparisons are spaced d characters apart to give a new partially-sorted list based on the ranks of the first d^2 characters. The spacing increases by a factor of d each iteration, so after $\lceil \log_d n \rceil$ iterations, all comparisons are guaranteed to reach the end of the string. Since d is constant, each comparison takes constant time, so each partial sort takes $O(n \log n)$ time. Because the partially-sorted list is in non-decreasing order, ranking its elements can be done in $O(n)$ time. Therefore, each iteration takes $(n \log n)$ time, and the overall doubling algorithm takes $O(n \log^2 n)$ time.

The two algorithms can be combined as follows: begin by using direct comparison, keeping track of cumulative number of character comparisons that exceed a depth of d. If the cumulative number exceeds some constant value, switch to the doubling algorithm. This heuristic ensures that the direct-comparison algorithm never performs more than $O(n \log n)$ character comparisons before it either completes or is abandoned in favor of the doubling algorithm. Therefore, the overall BST algorithm takes at most $O(n \log^2 n)$ time.

Move-to-Front (MTF) Coding

The input to the MTF coding stage is the string S^{BST}. Given an initial list L_0 of the characters in Σ in arbitrarily-defined order, the output, denoted by S^{MTF}, is a string of length n over the alphabet of integers $\Sigma' = [0, |\Sigma| - 1]$. See Figure 4.3. MTF coding exploits property (2) of the BST to produce a string that can be readily compressed by an entropy coding technique such as Huffman coding. MTF coding is performed by scanning the characters of S^{BST} in order of increasing index. For each character $c = S_i^{BST}$, the output character S_i^{MTF} is set to the index of c in L_i, and then c is moved to the front of L_i to produce L_{i+1}. That is, L_{i+1} is set to L_i, then c is removed from L_{i+1} and reinserted as the first element of L_{i+1}. Because $|\Sigma|$ is constant, the size of L is constant as well, and each update to L takes $O(1)$ time. The list is updated n times, so MTF coding takes $O(n)$ time. See Figure 4.4.

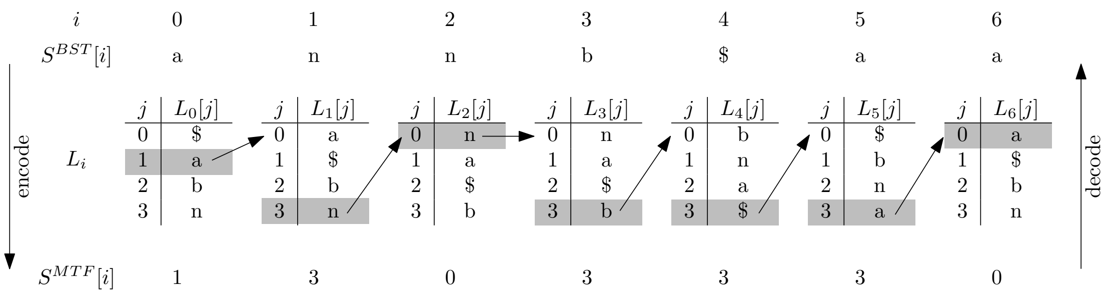
$$\Sigma = \{\$, a, b, n\}$$
$$S^{BST} = (a, n, n, b, \$, a, a)$$

assumed prefix

i	0	1	2	3	4	5	6	7	8	9	10
$S^{BST}[i]$	n	b	a	\$	a	n	n	b	\$	a	a
$prev[i]$	-	-	-	-	2	0	5	1	3	4	9
$C[i]$	-	-	-	-	{\$}	{\$,a,b}	{}	{\$,a,n}	{a,b,n}	{\$,b,n}	{}
$\|C[i]\|$	-	-	-	-	1	3	0	3	3	3	0

$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$

Fig. 4.3: MTF of the string "annb\$aa". $C[i]$ is the set of characters between $S^{BST}[i]$ and its previous occurrence.



i: 0 1 2 3 4 5 6
$S^{BST}[i]$: a n n b $ a a
L_i
j | $L_0[j]$: 0 $, 1 a, 2 b, 3 n
j | $L_1[j]$: 0 a, 1 $, 2 b, 3 n
j | $L_2[j]$: 0 n, 1 a, 2 $, 3 b
j | $L_3[j]$: 0 n, 1 a, 2 $, 3 b
j | $L_4[j]$: 0 b, 1 n, 2 a, 3 $
j | $L_5[j]$: 0 $, 1 b, 2 n, 3 a
j | $L_6[j]$: 0 a, 1 $, 2 b, 3 n
encode
decode
$S^{MTF}[i]$: 1 3 0 3 3 3 0

Fig. 4.4: MTF encoding and decoding. Observe that $S^{BST}[i] = L_i[S^{MTF}[i]]$. In both the encoder and the decoder, the shaded elements are moved to the front of the list according to the arrows. In the encoder, the shaded element is identified by searching the list L_i for the character $S^{BST}[i]$. In the decoder, the shaded element is chosen to be the one whose index is $j = S^{MTF}[i]$; no searching is necessary.

The purpose of MTF coding is to maintain L as a list of the characters seen so far in most-recently used (MRU) order. As a consequence, c will now have an index in L of zero, and if c is immediately followed by more repetitions of c, then the MTF coder will output zero for each of those subsequent repetitions. Each run of m repetitions of any character in S^{BST} will be converted to a nonzero integer followed by $m-1$ zeros in S^{MTF}. Even if c is not immediately followed by another occurrence of c, there will likely be one nearby. In that case, since only a few characters are moved ahead of c between the two occurrences of c in S', c will be assigned an index close to zero.

Huffman Coding

The input to the Huffman coding stage is the string S^{MTF}, and it produces as output (1) the string S^{BW}, a binary string (i.e., a string over the alphabet $\{0, 1\}$) whose length is $\Theta(n)$ and (2) a coding table T, whose size is constant given that $|\Sigma|$ is constant. In S^{MTF}, smaller integers tend to occur more frequently than larger integers, even if the characters in Σ occur an equal number of times in S. Therefore, S^{MTF} is amenable to entropy coding, even when S is not. This means that S^{BW} is typically shorter than any fixed-length encoding of S (e.g., the way it was originally stored on disk).

Huffman coding proceeds in three steps. In step 1, S^{MTF} is scanned once to build a frequency table F indicating how many times each character in Σ occurs in S^{MTF}; this takes $O(n)$ time. In step 2, the coding table T is constructed using a heap-based algorithm that takes only F as input. Since $|\Sigma|$ is constant, the size of F is also constant, so this takes $O(1)$ time. In step 3, S^{MTF} is scanned once more, and for each character S_i^{MTF}, the corresponding codeword $T(S_i^{MTF})$ is written to S_{BW}; this takes $O(n)$ time. See Figure 4.5. Overall, Huffman coding takes $O(n)$ time.

$$T = \begin{array}{l} 0 \rightarrow 10 \\ 1 \rightarrow 11 \\ 3 \rightarrow \ 0 \end{array}$$
$$S^{MTF} = (1, 3, 0, 3, 3, 3, 0)$$
$$S^{BW} = 11\ 0\ 10\ 0\ 0\ 0\ 10$$

Fig. 4.5: Huffman table and encoding of S^{MTF} (spaces added for clarity). Recall that this is, in fact, the compression of the original string "banana\$".

The output of the entire BW compression algorithm has size $\Theta(n)$ and consists of S_{BW} and T. The overall run time is dominated by the BST stage. If a linear-time suffix array algorithm is used to compute the BST, the overall runtime is $O(n)$. If the BST algorithm described herein is used instead, the overall runtime is $O(n \log^2 n)$.

4.2.2 Decompression

With the exception of the IBST, the decompression algorithm is simply the reverse of the compression algorithm:4: given S_{BW} and T, S_{BST} can be constructed in $O(n)$ time by applying the respective algorithms with the lookup tables inverted. The major difference is the IBST, which is simpler than the BST and consists of two steps. In step 1, the individual characters of S_{BST} are sorted using stable integer sorting, which takes $O(n)$ time. The resulting list of ranks is equivalent to a linked ring (a linked list whose tail points back to its head) of the characters in S_{BST} in the order they appear in S; see [25] or [51] for an explanation of why this is true. In step 2, the linked ring is traversed once, beginning from the character \$, to produce the characters of S in reverse order; this traversal takes $O(n)$ time. Therefore, the IBST, and thus the overall BW decompression algorithm, has a runtime of $O(n)$.

4.3 Parallel Algorithm

The parallel BW compression and decompression algorithms follow the same sequence of stages as the foregoing serial algorithms, but the sequential algorithm of each stage is replaced by an equivalent PRAM algorithm. As is the case in the serial algorithm, the dominant stage of the compression algorithm is the BST stage. Our PRAM algorithm for the BST stage, described below, requires the same work as the serial BST algorithm described above. If an $O(n)$-work compression algorithm is desired, the work-optimal algorithm of [140] can be used to compute the BST in $O(\log^2 n)$ time.

4.3.1 Compression

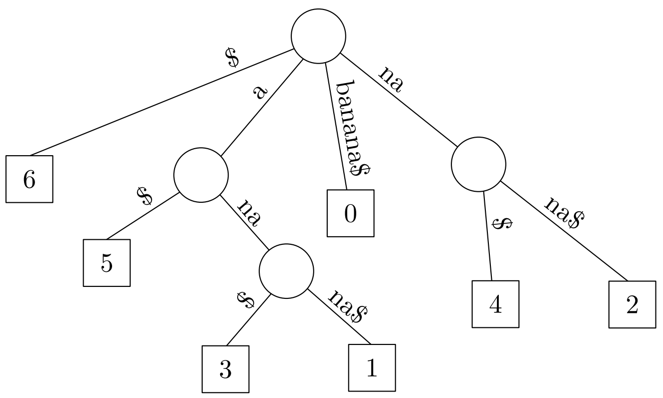
As in the serial algorithm, the input is a string S of length n over an alphabet Σ, where $|\Sigma|$ is constant with respect to n. The overall PRAM compression algorithm consists of the following three steps.

Block-Sorting Transform (BST)

The BST of a string S of length n can be computed as follows. Add a character \$ to the end of S that does not appear elsewhere in S. Sorting all rotations of S is equivalent to sorting all suffixes of S, as \$ never compares equal to any other character in S. Such sorting is equivalent to computing the suffix array of S, which can be derived from a depth-first search (DFS) traversal of the suffix tree of S (see Figure 4.6). The suffix tree of S can be computed in $O(\log^2 n)$ time and $O(n)$ work using the algorithm of [140]. The order that leaves are visited in a DFS traversal of the suffix tree can be computed using the Euler tour technique [155] within the same complexity bounds, yielding the suffix array of S. Given the suffix array SA of S, we derive S^{BST} from S in $O(1)$ time and $O(n)$ work as follows:

$$S^{BST}[i] = S[(SA[i] - 1)_{\bmod n}], 0 \leq i < n$$

Overall, computing the BST takes $O(\log^2 n)$ time using $O(n)$ work.



i	0	1	2	3	4	5	6
$S[i]$	b	a	n	a	n	a	\$
$SA[i]$	6	5	3	1	0	4	2
$S[SA[i]-1]$	a	n	n	b	\$	a	a

Fig. 4.6: Suffix tree and suffix array (SA) for the string S = "banana$".

Move-to-Front (MTF) Coding

Let $S_{i,j}^{BST}$, $0 \leq i \leq j \leq n$ be the substring $[S_i^{BST}, ..., S_{j-1}^{BST}]$; $S_{i,j}^{BST}$ is defined to be the null string when $i = j$. Let $\sigma_{i,j}$ be the set of characters contained within $S_{i,j}^{BST}$ and $M_{i,j}$ be the listing of the characters in $\sigma_{i,j}$ in order of last occurrence in $S_{i,j}^{BST}$ (i.e., in MRU order); this is the empty list when $i = j$. Denote by $x \oplus y$ the list formed by concatenating to the end of y the list formed by removing from x all elements that are contained in y. The key idea behind the PRAM algorithm for MTF coding is the observation, noted in the discussion of the serial MTF algorithm, that L_i is the MRU listing of the characters of $S_{0,i}^{BST}$ followed by the remaining characters of Σ in their originally defined order. That is, $L_i = L_0 \oplus M_{0,i}$.

Observe that $M_{i,j} = M_{i,k} \oplus M_{k+1,j}$ for all k, $i \leq k < j$. This implies that $M_{i,j} = \oplus_{k=i}^{j-1} M_{k,k+1}$. By definition, $M_{k,k+1}$ is simply the list $[S_k^{BST}]$. Furthermore, $\oplus$ is associative, and by the assumption that $|\Sigma|$ is constant, takes $O(1)$ time and work to compute. Therefore, $M_{0,i}$, and thus L_i, for $0 \leq i < n$ can be computed in $O(\log n)$ time using $O(n)$ work by the standard PRAM algorithm for computing all prefix-sums with respect to the operation $\oplus$. The prefix sums algorithm works in two phases:

1. Adjacent pairs of MTF lists are combined using $\oplus$ in a balanced binary tree approach until only one list remains (see Figure 4.7).

2. Working back down the tree, the prefix sums corresponding to the rightmost leaves of each subtree are computed using the lists computed in phase 1 (see Figure 4.8).

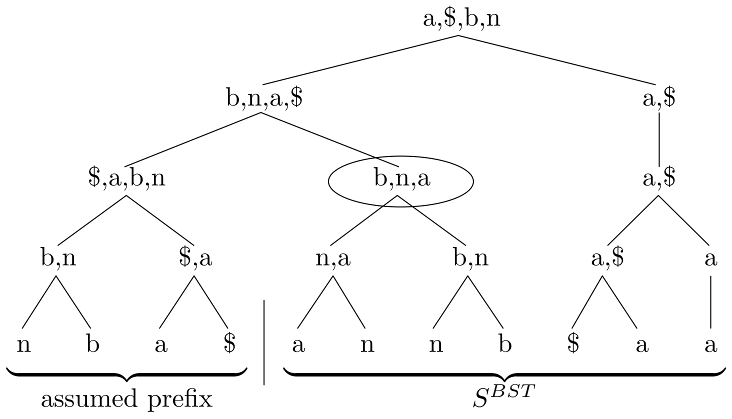


Fig. 4.7: Phase 1 of prefix sums: Computing local MTF lists for "annb$aa" using the operator $\oplus$. Each node in the tree is the $\oplus$-sum of its children. For example, the circled node is (n, a) $\oplus$ (b, n).

Given L_i, S_i^{MTF} is simply the index in L_i of S_i^{BST}, which can be found for all characters independently in $O(1)$ time and $O(n)$ work. Therefore, MTF coding can be performed in $O(\log n)$ time using $O(n)$ work.

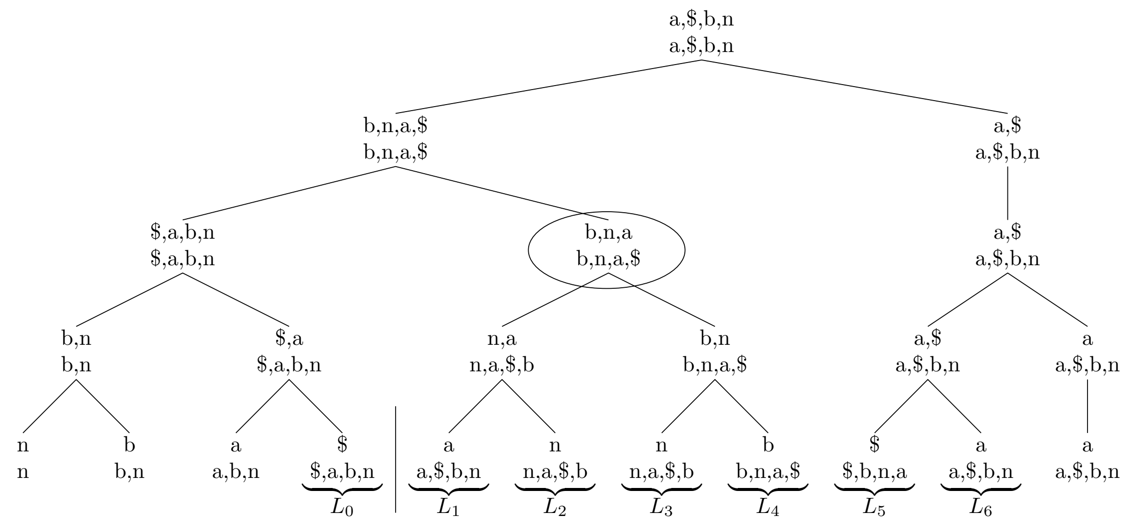


Fig. 4.8: Computing the prefix sums of the output of the BST stage, "annb$aa", with respect to the associative binary operator $\oplus$. The top line of each node is copied from the tree in Figure 4.7. The bottom line of a node V is the cumulative $\oplus$-sum of the leaf nodes starting at the leftmost leaf in the entire tree and ending at the rightmost child of V (i.e., the prefix sum up to the rightmost leaf under V). For example, the circled node contains the sum of leaves corresponding to the prefix "nba$annb". Observe the correspondence of the labeled lists with Figure 4.4.

Huffman Coding

The PRAM algorithm for Huffman coding follows readily from the serial algorithm. In step 1, F is constructed using the integer sorting algorithm outlined in [33], which sorts a list of n integers in the range $[0, r-1]$ in $O(r + \log n)$ time using $O(n)$ work. Because $r = |\Sigma|$ is constant, step 1 runs in $O(\log n)$ time and $O(n)$ work. Step 2 of the serial algorithm runs in $O(1)$ serial time, so the same algorithm can be used to construct T from F in $O(1)$ time and work. Step 3 is performed in as follows. First, the prefix-sums of the code lengths $|T(S_i^{MTF})|$ are computed into the array U in $O(\log n)$ time and $O(n)$ work. Then, in parallel for all i, $0 \le i < n$, $T(S_i^{MTF})$ is written to S^{BW} starting at position U_i in $O(1)$ time using $O(n)$ work. Therefore, the overall Huffman coding stage runs in $O(\log n)$ time using $O(n)$ work.

The above discussion proves the following theorem:

Theorem 1: The above algorithm solves the Burrows-Wheeler Compression problem in $O(\log^2 n)$ time using $O(n)$ work.

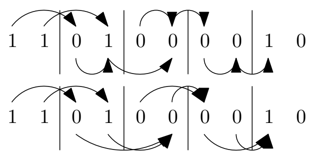
4.3.2 Decompression

Huffman Decoding

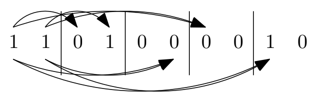
The main obstacle to decoding S^{BW} in parallel is that, because Huffman codes are variable-length codes, we do not know where the boundaries between codewords in S^{BW} lie. We cannot simply begin decoding from any position, as the result will be incorrect if we begin decoding in the middle of a codeword. Thus, we must first identify a set of valid starting positions for decoding. Then, we can trivially decode the substrings of S^{BW} corresponding to those starting positions in parallel.

Our algorithm for locating valid starting positions for Huffman decoding is as follows. Let l be the length of the longest codeword in T, the Huffman table used to produce S^{BW}; l is constant because $|\Sigma|$ is. Without loss of generality, we assume that $|S^{BW}|$ is divisible by l. Divide S^{BW} into partitions of size l. Our goal is to identify one bit in each partition as a valid starting position. The computation will proceed in two steps: (1) initialization and (2) prefix sums computation.

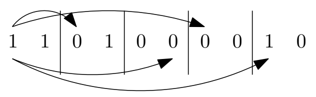
For the initialization stage, we consider every bit i, $0 \leq i < |S^{BW}|$, in S^{BW} as if it were the first bit in a string to be decoded, henceforth S_i^{BW}. In parallel for all i, we decode S_i^{BW} (using the standard serial algorithm) until we cross a partition boundary, at which point we record a pointer from bit i to the stopping point. Now, every bit i has a pointer $i \rightarrow j$ to a bit j in the immediately following partition, and if i happens to be a valid starting position, then so is j. See Figure 4.9(a).



(a) Step 1: initialization.



(b) Step 2: prefix sums.



S^{BW}	$\underline{1}1$	$\underline{0}$ 10	$\underline{0}$	$\underline{0}$ 0	$\underline{1}0$
S^{MTF}	1	3, 0	3	3, 3	0

(c) Pointers from bit 0, corresponding to valid starting positions in S^{BW} (underlined).

Fig. 4.9: Huffman decoding of S^{BW} (from Figure 4.5).

For the prefix sums stage, we define the associative binary operator $\oplus$ to be the merging of adjacent pointers (that is, $\oplus$ merges $A \to B$ and $B \to C$ to produce $A \to C$). See Figure 4.9(b). The result is that there are now pointers from each bit in the *first* partition to a bit in every other partition. Finally, we identify all bits with pointers from bit 0 as valid starting positions for Huffman decoding (see Figure 4.9(c)); we refer to this set of positions as V. All this takes $O(\log n)$ time and $O(n)$ work.

The actual decoding is straightforward and proceeds as follows.

1. Employ $|S^{BW}|/l$ (which is $O(n)$) processors, assign each one a different starting position from the set V, and have each processor run the serial Huffman decoding algorithm until it reaches another position in V in order to find the number of decoded characters. Do not actually write the decoded output to memory yet. This takes $O(1)$ time because the partitions are of size $O(1)$.

2. Use prefix sums to allocate space in S^{MTF} for the output of each processor. ($O(\log n)$ time, $O(n)$ work)

3. Repeat step (1) to actually write the output to S^{MTF}. ($O(1)$ time, $O(n)$ work)

These three steps, and thus the entire Huffman decoding algorithm, take $O(\log n)$ time and $O(n)$ work.

Move-to-Front (MTF) Decoding

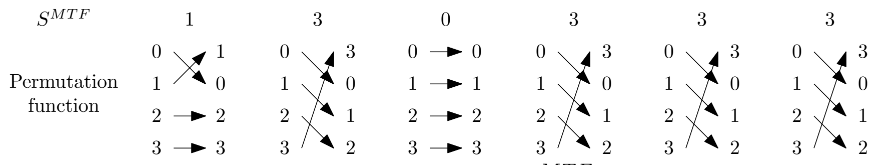
The parallel MTF decoding algorithm is similar to the parallel MTF encoding algorithm but uses a different operator for the prefix sums step. In contrast to MTF encoding, MTF decoding uses the characters of S^{MTF} directly as indices into the MTF list. Therefore, S_i^{MTF} defines a fixed permutation function that maps L_i to L_{i+1}. Denote by $P_{i,j}$ the permutation mapping L_i to L_j. Then, $P_{0,j}$ can be computed for all j, $0 \leq j < n$, using prefix sums with function composition as the associative operator. See Figure 4.10. A permutation function for a list of constant size can be represented by another list of constant size, so composing two permutation functions takes $O(1)$ time and work. Therefore, the prefix sums, and the overall MTF decoding algorithm, take $O(\log n)$ time and $O(n)$ work.

Inverse Block-Sorting Transform (IBST)

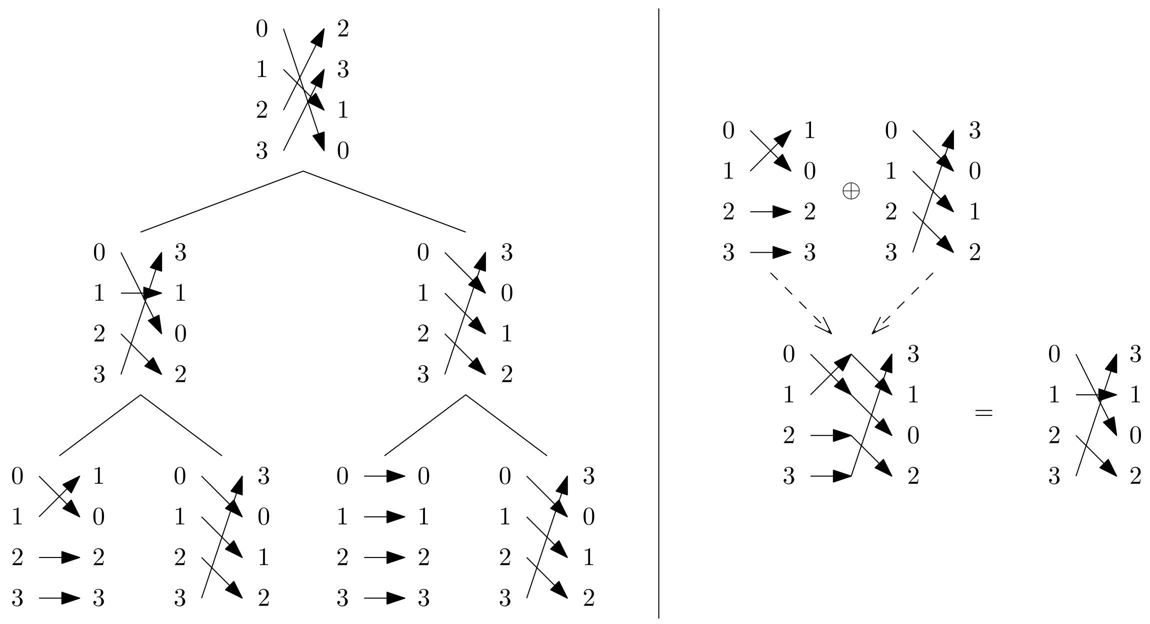
The parallel IBST algorithm proceeds in two steps, analogous to the serial algorithm. In step 1, the integer sorting algorithm of [33] is used to sort the characters of S^{BST}. Because $|\Sigma|$ is constant, the characters have a constant range, and so this step takes $O(\log n)$ time and $O(n)$ work. In step 2, and the list ranking algorithm of [34] is used to rank the linked list in $O(\log n)$ time and $O(n)$ work. Finally, the characters of S^{BWT} are written to S according to their rank in the linked list; this

1 3 0 3 3 3 0

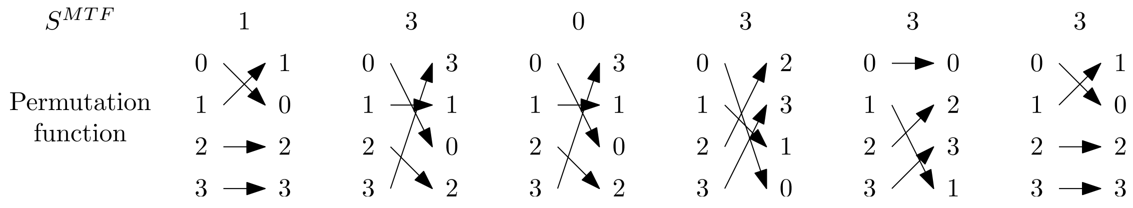
(a) S^{MTF} (from Figure 4.9).



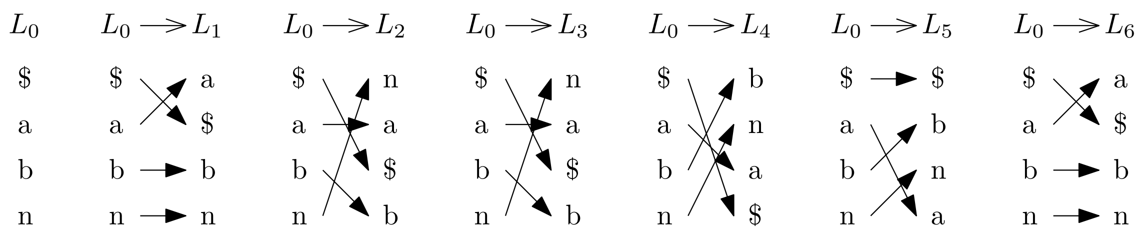
(b) Initialization: the permutation function defined by $S^{MTF}[i]$ moves element i to the front of its input list.



(c) (Left) Prefix sums: composition of permutation functions using a balanced binary tree (here, we show the tree for the first four elements).
(Right) Computing the $\oplus$-sum of the leftmost two leaves of the tree. The result is the parent of the two leaves.



(d) Output of prefix sums: composed permutation functions.



(e) Applying the composed permutation functions of (d) to L_0 to produce L_1, L_2, etc.

Fig. 4.10: MTF decoding of S^{MTF} from Figure 4.9: construction of L_i in parallel using composed permutation functions. The last character of S^{MTF} is not used in this construction because the corresponding list L_7 is not needed. Observe the correspondence of the labeled lists in (e) with Figure 4.4.

takes $O(1)$ time and $O(n)$ work. Overall, the IBST takes $O(\log n)$ time and $O(n)$ work.

The above discussion proves the following theorem:

Theorem 2: The above algorithm solves the Burrows-Wheeler Decompression problem in $O(\log n)$ time using $O(n)$ work.

4.4 Experimental Validation

4.4.1 The XMT Platform

The Explicit Multi-Threading (XMT) general-purpose computer architecture is designed to improve single-task completion time. It does so by supporting programs based on Parallel Random-Access Machine (PRAM) algorithms but relaxing the synchrony required by the PRAM model. The XMT programming model differs from the strict PRAM model in two ways:

1. The PRAM model requires specifying the instruction that will be executed by each processor at each point in time, but XMT uses the work-depth methodology ([146, 164]), which allows the programmer to specify all of the operations that can be performed at each point in time while leaving to the runtime environment the assignment of those operations to processors.

2. The PRAM model requires instructions to be executed in lockstep by all processors at once, but XMT programs follow independence-of-order semantics: parallel sections of code are delimited by spawn-join instruction pairs, and

threads only synchronize when they reach the join instruction at the end of the parallel section.

The XMT architecture consists of the following: a number of lightweight cores (TCUs) grouped into clusters, a single core (master TCU or MTCU) with its own local cache, a number of mutually-exclusive cache modules shared by the TCUs and MTCU, an interconnection network connecting the TCUs to the cache modules, and a number of DRAM controllers connecting the cache modules to off-chip memory. Each TCU has a register file, a program counter, an execution pipeline, and a lightweight ALU. Each TCU also contains prefetch buffers, which can be used by the compiler to prefetch data from memory before it is needed, reducing the length of the sequence of round trips to memory (LSRTM) and improving performance [162]. Each cluster has one or more multiply/divide units (MDUs), floating-point units (FPUs), and a compiler-managed read-only cache, all of which are shared by the TCUs within the cluster. When a parallel section of code is reached, the MTCU broadcasts the instructions in that section to all of the TCUs, and each TCU stores the instructions in a buffer. Virtual threads are assigned to TCUs using a dedicated prefix-sum network.

An overview of XMT with details relevant to work on application can be found in [29].

4.4.2 Evaluated Configurations

Because XMT is an experimental platform, we establish that XMT is competitive with single-chip multi-cores and many-cores currently available on the market by choosing a configuration of XMT that would use resources comparable to one such commercially-available chip. The most recent comparison of XMT with existing chips is [29], in which a 1024-TCU configuration of XMT with 4 MB[3] shared cache (herein called XMT-1024) is shown to use a comparable silicon area to the NVIDIA GTX 280 GPGPU, which uses 576 mm^2 of silicon in 65 nm technology. In [103], XMT-1024 was shown to remain in the same power envelope as the GTX 280 as well. Since then, silicon technology has improved, and the current successor to the GTX 280, the GTX 680, is manufactured in 28 nm technology, with a die size of 294 mm^2, just over half that of the GTX 280; however, recall that nominally 294 mm^2 in 28 nm technology offers more than twice the device capacity of 576 mm^2 in 65 nm technology.

We compare our parallel implementation of Burrows-Wheeler[4] compression running on a 64-TCU FPGA prototype of XMT [170], and also on XMT-1024, against bzip2 running on one core of the Intel Core i5-2500K CPU with 6 MB of L3 cache. To obtain results for the XMT-1024 configuration, we used XMTSim, the cycle-accurate simulator of the XMT architecture. XMTSim and the XMTC compiler are described in [100] and have already been the basis for several publications including [29]. The most recent validation of the cycle-accuracy of the simulator is [101, Chapter 4], which shows that the simulator cycle counts match those of the FPGA except in a minority of cases, where the discrepancy may be up to 33%, due in part to interconnect and DRAM technology limitations in the FPGA prototype

[3] 1 MB = 2^{20} bytes

[4] Available at `http://www.umiacs.umd.edu/users/vishkin/XMT/OPEN_SOURCE_ALGS/`

File	Description	Size (bytes)
bible.txt	The King James version of the bible	4,047,392
E.coli	Complete genome of the E. Coli bacterium	4,638,690
world192.txt	The CIA world fact book	2,473,400

Tab. 4.1: Files in the Large Corpus

that would not exist in an ASIC product. For BW compression, the difference due to these limitations is 15%.

4.4.3 Data Sets

We perform our comparison using the Large Corpus from the Canterbury Corpus [136], a standard set of files used to evaluate compression algorithms. We use a block size of 900,000 bytes for both bzip2 and our parallel implementation, and we obtain speedup results for each block separately since both implementations compress one block at a time. We use the notation *file.i* to denote block i of a file named *file*. Because the file sizes are not evenly divisible by the block size, the last block of each file is smaller than 900,000 bytes, and such blocks are denoted using parentheses. For comparison purposes with bzip2 implementations, our experimental results are reported with respect to blocks. It should also be noted that our fine-grained approach is orthogonal to existing coarse-grained ones allowing one to benefit from both in a single implementation.

4.4.4 Implementation Details

Prefix Sums

We use a k-ary tree to implement prefix sums operations. To improve the performance of these operations, we cluster threads in the spawn block immediately preceding each prefix sums operation into groups of size c and merge them with the first iteration of the prefix sums operation. Similarly, we merge the last iteration with the following spawn block. In our code, we fixed c at 256, corresponding to the 8-bit character set used by bzip2.

Run-Length Encoding (RLE)

To reduce the size of its output, bzip2 adds two run-length encoding (RLE) stages to the basic BW compression algorithm. We added these stages to our implementation as well. Since this enhancement is not part of the core compression algorithm and thus not covered in the theoretical portion of this chapter, we state without proof that the RLE algorithm we implemented runs in $O(\log n)$ time and $O(n)$ work.

Multiple Huffman Tables

Bzip2 also implements a heuristic that switches among multiple Huffman tables to possibly reduce the size of its output. We are not aware of a parallel algorithm that can decode data encoded using this heuristic within the bounds given by Theorem 2. To enable a fair comparison with our implementation, we modified bzip2 to only use a single Huffman table. For the inputs we used in our comparison, this caused the average size of the compressed output to increase by 2.75% relative to that of the unmodified bzip2. On average, the modified bzip2 compression ran 1.5% faster than the unmodified version, and the decompression ran 7.5% faster.

Block-Sorting Transform (BST)

To provide better practical performance for small inputs, we use a randomized, recursive variant of shared-memory sample sort to compute the BST. Although a serial recursive sample sort algorithm is described and analyzed in [8], there appears to be no prior polylogarithmic-time PRAM analog of such an algorithm. We describe the sorting algorithm below; after the rotations of S are sorted, S^{BST} can be derived in $O(1)$ time using n processors by having each processor i, $0 \leq i < n$, output the last character of string i in the list of sorted strings.

The initial list of the n rotations of S is passed to a procedure called $SAMPLESORT$. Let T_c be the time, and W_c be the work, required to compare two strings. Given a list L as input, $SAMPLESORT(L)$ proceeds in five steps. (1) A subset of n/k splitters, with $k > 1$ a constant, is randomly selected from L and placed in the list L'. (2) If L' contains more than one element, $SAMPLESORT(L')$ is called recursively. (3) Each element of L is ranked within L' using binary search. (4) The elements of L are partitioned according to their rank in L'. Because the ranks are integers in the range $[0, n-1]$, this partitioning can be done using, e.g., the integer sorting algorithm of [83], which runs in $O(\log n)$ time and $O(n\sqrt{\log n})$ work. (5) The partitions are sorted in parallel, with a serial comparison-based sort applied to each partition.

4.5 Results

4.5.1 Comparison with bzip2

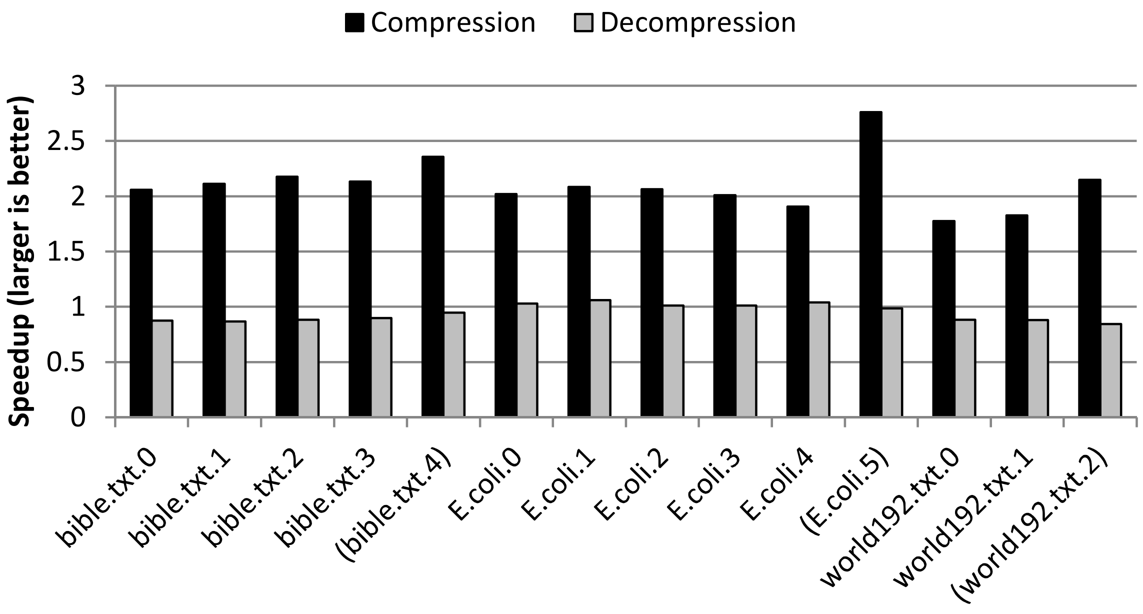


Fig. 4.11: Speedups obtained using the 64-TCU FPGA prototype. Partial blocks at the ends of files are indicated with parentheses.

Speedups for the 64-TCU FPGA prototype are shown in Figure 4.11 and are in the range 1.8-2.8x for compression and 0.8-1.1x for decompression. Speedups for the simulated XMT-1024 configuration are shown in Figure 4.12 and are in the range 12x-25x for compression and 11x-13x for decompression. The main reason that speedups for 64 TCUs are low, but then scale up nicely for 1024 TCUs, is the extra work that our parallel algorithms do beyond the original serial algorithm.

On the FPGA, speedups for partial blocks are higher than for the preceding full blocks in the same file. This is because the partial blocks fit better than full blocks

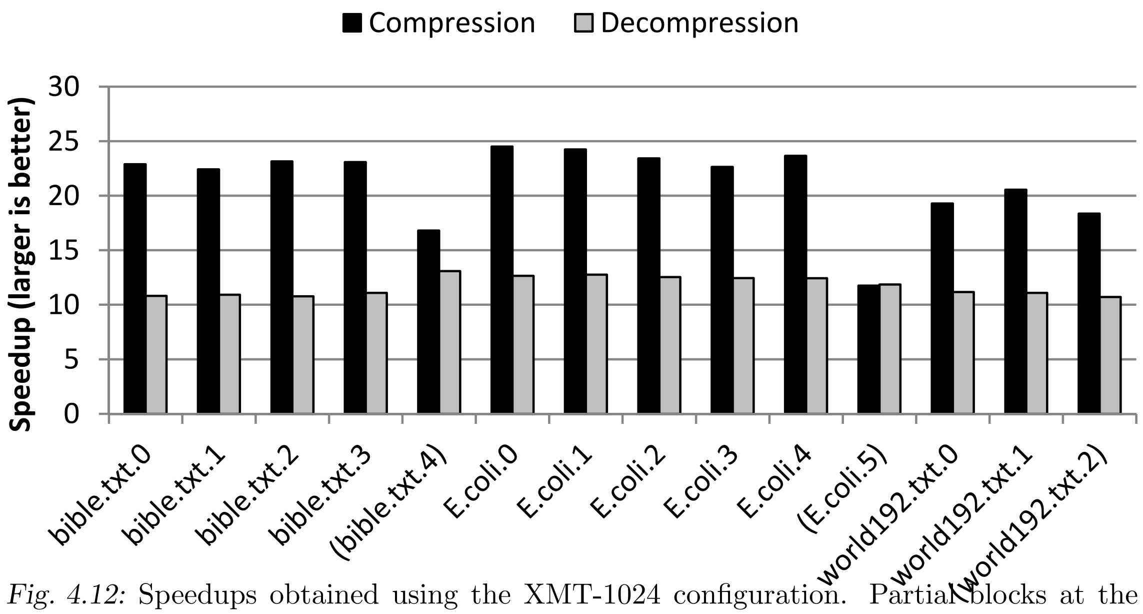


Fig. 4.12: Speedups obtained using the XMT-1024 configuration. Partial blocks at the ends of files are indicated with parentheses.

into the limited cache size (256 KB[5]) of the FPGA. The situation is reversed for XMT-1024, where the partial blocks bible.txt.4 and E.coli.5 exhibit lower speedups than full blocks in the same file. This is because we tuned our code to provide optimal performance on 900 KB blocks. For smaller inputs, performance can be improved by tuning the code to spread the work among a larger number of threads, decreasing granularity (at the cost of higher overhead). For example, decreasing the factor k in $SAMPLESORT$ and the clustering factor c provides up to 1.3x higher speedup for partial blocks.

Of all the stages in the parallel implementation, the BST in the compression routine is the most time consuming, and the corresponding inverse BST (IBST) in the decompression routine is the second most time-consuming step. This is equally true for bzip2 compression; it may be true for bzip2 decompression as well, but the stages in the bzip2 decompressor are interleaved, so we could not separate out

[5] 1 KB = 2^{10} bytes

IBST. Therefore, improving the performance of these stages has the greatest effect on overall runtime.

The aforementioned BST and IBST stages have irregular parallelism and memory access patterns. In addition, all of the stages in the compression and decompression routines employ fine-grained parallelism. In contrast to many parallel computing platforms, which have difficulty running such algorithms efficiently, the XMT platform is designed with such algorithms in mind. This is perhaps one reason that others have overlooked a parallel-algorithmic approach.

In addition to allowing parallelism to be exploited within a block, our approach has the advantage that it only requires working space for a single block, as blocks are processed one at a time. Therefore, all working data fits in cache, and DRAM is only accessed to read input blocks and write output blocks. In contrast, if we were to compress multiple blocks simultaneously using a single XMT chip, we would only be able to process a few blocks in parallel without spilling working data to DRAM. Therefore, our approach may have more efficient cache utilization than block-parallel approaches.

The current embodiments of the XMT platform have the limitation that memory can only be addressed in terms of 32-bit words; threads cannot write to individual bytes without overwriting all bytes within a word. Therefore, if multiple threads need to be able to write to arbitrary elements of an array, the elements of that array must be stored as 32-bit words even if they could otherwise be stored as single bytes. Commercial-grade platforms, such as the Intel processor we compare against, do not have this limitation. This means that our results are conservative relative to a more complete version of XMT with this restriction removed.

For XMT-1024, our parallel decompression implementation performs better on E.coli.5 than on any other input. This is because it is smaller than every other input (by at least a factor of 3), and thus the working set fits into cache for this input alone. To verify this, we tested a variant of XMT-1024 with 16 MB of cache and found that the minimum speedup increased from 10x to 12x. This suggests that it may be worthwhile to take advantage of improvements in silicon technology noted earlier to increase the size of the shared cache of XMT.

4.5.2 Using Compression to Increase Bandwidth

We compare our implementation (henceforth xmt-bw) to a number of other compression libraries by providing as input the entire (11 MB) Large corpus[6] and measuring the compression ratio and speed; except for xmt-bw and pbzip2, all libraries are serial. Figure 4.13 shows our results. Each library has two regions. (1) As long as the effective bandwidth does not exceed the maximum compression speed, the effective bandwidth is limited only by the compression ratio (sloped portion, bandwidth-limited). (2) Once the maximum compression speed is reached, no further increase is possible (horizontal portion, compute-limited).

We compare xmt-bw against two classes of compression libraries:

- High compression, low speed: zlib [69], bzip2 [144], pbzip2[7] [73], and xz [35].
- Low compression, high speed: Snappy [78], LZO [131], QuickLZ [1], liblzf

[6] All of the implementations tested here (including our own) subsequently divide the input into blocks.

[7] 4 cores

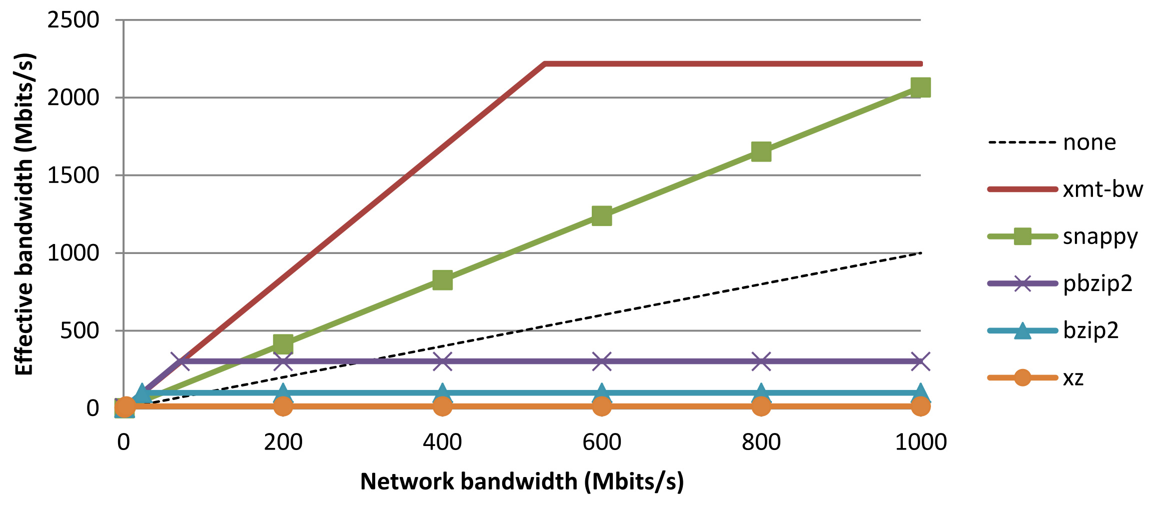


Fig. 4.13: Comparison of maximum transfer rates over a bandwidth-limited network using BW compression on XMT (xmt-bw) and existing compression libraries. The dotted diagonal line represents the baseline of no compression.

[109], and FastLZ [88].

Of these, QuickLZ, liblzf and FastLZ are dominated by Snappy, and zlib is dominated by pbzip2. Snappy outperforms LZO up to 1.46 Gbits/s, beyond which LZO provides 6% more effective bandwidth. Results for the remaining libraries are shown in Figure 4.13.

For network bandwidths up to 3 Mbits/s, xz outperforms xmt-bw by 4% due to its slightly higher compression ratio. Beyond that, for network bandwidths up to 1 Gbit/s, xmt-bw is dominant; it is only outperformed by Snappy, LZO, and QuickLZ at higher bandwidths. Remarkably, this breakpoint coincides with the peak bandwidth of Gigabit Ethernet, which is commonly used on commodity systems. Even on networks with a higher peak bandwidth, the point-to-point bandwidth depends on network load and may fall into the range where xmt-bw provides an advantage. Finally, beyond 3.1 Gbits/s, it is more efficient to transmit data uncompressed.

4.6 Conclusion

This chapter is the first to demonstrate work-optimal algorithmic and empirical feasibility of parallel compression which compromises neither speed nor compression quality. For small inputs, it provides speedups where no other approach does. For transmission of data over a network, it provides a larger increase in effective bandwidth than other approaches over a wide range of network bandwidths.

Today's parallel architectures allow good speedups on regular dense-matrix type programs, but are basically unable to match this success outside this, including for: 1. irregular problems/programs; and, 2. strong scaling. Extending parallel hardware to address these domains could potentially lead to phenomenal growth in supercomputing: 1. Nearly all serial algorithms in the CS curriculum are irregular; how many more programmers and applications will migrate to parallel computing if such parallel algorithms will deliver good speedups? 2. Publication of slow-down results, as in [135], are extremely rare and reflect an unusual level of interest in a problem. How many more will become interested if commercial hardware allows speedups?

Chapter 5: Fast Fourier Transform (FFT)

5.1 Introduction

The parallel computing community has increasingly shifted its attention to communication avoidance as a way to address the end of Dennard scaling and the attendant difficulty in scaling down power consumption: see for example the National Academies report [120], work by Jim Demmel's group [47] on communication avoidance upper and lower bounds and many of the recent books in the computer architecture series by publisher Morgan and Claypool such as [147]. However, there are limits to the performance improvements that can be attained by focusing on reducing data movement. The strength of current parallel architectures lies in solving problems, such as dense-matrix multiplication, that can be solved by algorithms that are regular and require limited communication. For other algorithms, which are irregular or require high bandwidth, these platforms have been able to demonstrate limited speedups. Furthermore, the challenges of communication avoidance have arguably harmed programmers' productivity [165].

Still, the default mode for parallel programming research is reliance on off-the-shelf hardware. But what if alternative machines, or hardware features, are feasible, and can offer significant advantages? Clearly, such out-of-the-box hardware and the enabling technologies it may require are unlikely to ever be developed before their advantages are sufficiently understood. In contrast to work that seeks to avoid data movement, the current work examines the problem from an alternate angle: assuming that is it possible to reduce the energy cost of data movement, is it possible to obtain strong speedups on problems for which such speedups have proven elusive?

This question has been partially answered in the affirmative by prior work on the Explicit Multi-Threading (XMT[1]) general-purpose architecture [170], which aims to improve single-task completion time and ease-of-programming for parallel applications by supporting Parallel Random Access Model (PRAM) programming [94, 104]. Such work, discussed in Section 5.3 (in part, by way of reference to [52]), has focused largely on speedups for highly irregular parallel algorithms. Here, we begin to examine another class of algorithms that also appear too challenging for current platforms, namely, those that are regular but communication intensive.

Specifically, we examine one such algorithm, the fast Fourier transform (FFT). The FFT is an important numerical algorithm used in fields such as signal processing and scientific computing. What sets the FFT apart from other regular numerical algorithms is its high communication needs; given $O(S)$ local memory, it requires $O(n \log n / \log S)$ I/O operations [89], which suggests that caches are of limited use in reducing the bandwidth required by the FFT. Indeed, prior work using existing platforms obtained modest speedups relative to the hardware invested; see Section 5.1.1 for some speedup examples, and for a comparison of prior speedups and hardware

[1] XMT at the University of Maryland, not to be confused with the code name Cray XMT used during 2007-2011

invested in them with the results (speedups and assumed hardware) of the current chapter, see Section 5.5.1 and Table 5.6.

Companion work on XMT [132] investigates the use of enabling technologies including 3D VLSI and microfluidic cooling to increase communication bandwidth on chip to shared cache, concluding that these technologies indeed enable XMT to scale up to 8x larger than would be possible without them. It also briefly considers the potential of photonics to extend this improvement off chip to greatly increase bandwidth to DRAM. The companion work uses XMT as a vehicle for performing a quantitative feasibility analysis of the enabling technologies in terms of temperature and power.

The intellectual merit of the work presented here is to complement the foregoing feasibility analysis with a quantitative analysis of the corresponding performance benefit, again using XMT as a vehicle. The two combined lay the groundwork for future analysis of the enabling technologies. In particular, the purpose of this work is to resolve the chicken-and-egg problem posed by enabling technologies: development of enabling technologies will not advance without evidence of their benefit, while such evidence apparently cannot be obtained until these technologies have already been developed. In order to resolve this impasse, we obtain preliminary results using a simulator (Section 5.3.1), which does not require actual hardware to already exist. We recognize that the validity of these results is limited and will only reach the level of those for existing systems once we are able to obtain results on actual hardware.

While it is expected that increased bandwidth enabled by such technologies would lead to improved performance the (high) rate of improvement shows great promise. Of particular interest is the potential benefit of silicon photonics. Development of photonic technologies advanced enough to enable the largest systems considered herein would require non-trivial engineering effort. Although photonics is a topic of current interest, even the most recent progress has been modest in scale (e.g., [152]). In order to motivate more ambitious effort, we demonstrate that photonics could enable a single-chip many-core processor to outperform a much larger high-performance computing (HPC) cluster of nodes interconnected via traditional optics. This is especially true when the application, such as FFT, greatly underutilizes the peak computational capacity of the HPC system due to limited inter-node bandwidth. Here, XMT is a natural fit as the high off-chip bandwidth is coupled with matching on-chip bandwidth, permitting significantly higher utilization of available computational resources.

Our analysis consists of two parts. First, we measure the speedup of FFT on XMT relative to existing platforms. Specifically, we compare against FFTW [64], a highly-optimized implementation of the FFT, running on a single core of a modern Intel processor and also on multiple cores. In addition, we compare against a tuned FFT implementation running on Edison, a large HPC cluster. Second, we use the Roofline [172] model to evaluate how close our FFT implementation comes to achieving the peak performance possible on selected configurations of XMT and how performance may be further improved.

5.1.1 Comparison to prior work on the FFT

GPGPU Researchers at Microsoft [79] demonstrated performance of up to 300 GFLOPS on the NVIDIA GTX 280, with speedups of 2-4X over NVIDIA's cuFFT and 8-40X over Intel's MKL. The best result for a 2D FFT was around 120 GFLOPS, achieved with an input size of 1024×1024. No results are reported for 3D FFT.

Using a hybrid GPU-CPU algorithm, Chen and Li [30] achieved up to 43 GFLOPS for a 2D FFT and up to 27 GFLOPS for a 3D FFT on an NVIDIA Tesla C2075.

MPI Recent work [150] considers large 3D FFT on two high-end Cray systems using up to 32,768 cores. For an input of size 1024^3, the best result was 13,603 GFLOPS using 32,768 cores. Weak scaling results ranged from 159 GFLOPS for an input of size 512^3 to 17,611 GFLOPS for an input of size $4096 \times 4096 \times 2048$.

Similar work on large MPI clusters [129] shows that a 3D FFT on an input of size 1024^3 can be computed in as little as 49 milliseconds (i.e., 3287 GFLOPS) using 16384 cores of an IBM-BlueGene/Q cluster.

Prior work on XMT Prior work on the FFT on XMT [141] did not consider 3D FFT and was limited to fixed-point arithmetic. Also, the prior work focused exclusively on FFT as an application rather than as a benchmark for evaluating the benefit of augmenting a computer architecture with enabling technologies.

5.2 Background

5.2.1 XMT Architecture

The Explicit Multi-Threading (XMT) general-purpose architecture [170] is a many-core architecture which aims to improve single-task completion time and ease-of-programming for parallel applications by supporting Parallel Random Access Model (PRAM) programming [94, 104]. For some advantages of XMT, see Section 5.3.

The XMT processor includes a master thread control unit (MTCU); processing clusters, each comprising several light-weight thread-control units (TCUs); a high-bandwidth low-latency interconnection network; memory modules (MM), each comprising on-chip cache and off-chip memory; prefix-sum (PS) unit(s); and global registers. The shared-memory-modules block (bottom left of Fig. 5.1) suppresses the sharing of a memory controller by several MMs. The processor alternates between serial mode (in which only the MTCU is active) and parallel mode. The MTCU has a standard private data cache (used in serial mode) and a standard instruction cache. The TCUs, which lack a write data cache, share the MMs with the MTCU.

The overall XMT design is guided by a general design ideal we call no-busy-wait finite-state-machines, or NBW FSM, meaning the FSMs, including processors, memories, functional units, and interconnection networks comprising the parallel machine, never cause one another to busy-wait. It is ideal because no parallel machine can operate that way. Nontrivial parallel processing demands the exchange

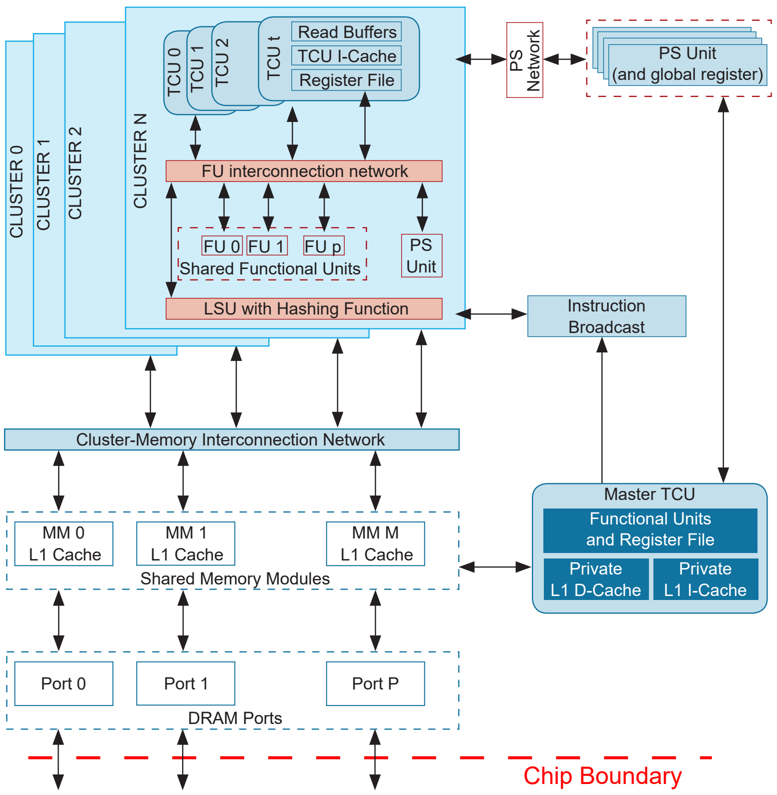


Fig. 5.1: Block Diagram of the XMT Architecture

of results among FSMs. The NBW FSM ideal represents our aspiration to minimize busy-waits among the various FSMs comprising a machine.

We cite the example of how the MTCU orchestrates the TCUs to demonstrate the NBW FSM ideal. The MTCU is an advanced serial microprocessor that also executes XMT instructions (such as spawn and join). Typical program execution flow can also be extended through nesting of sspawn commands. The MTCU uses the following XMT extension to the standard von Neumann apparatus of the program counters and stored program. Upon encountering a spawn command the MTCU broadcasts the instructions in the parallel section starting with that spawn command and ending with a join command on a bus connecting to all TCU clusters. The largest ID number of a thread the current spawn command must execute (Y) is also broadcast to all TCUs. The largest ID (index) of the executing threads is stored in a global register X. In parallel mode, a TCU executes one thread at a time. Executing a thread to completion (upon reaching a join command), the TCU does a prefix-sum using the PS unit to increment global register X. In response, the TCU gets the ID of the thread it could execute next; if the ID is $\leq$Y, the TCU executes a thread with this ID. Otherwise, the TCU reports to the MTCU that it finished executing. When all TCUs report they have finished, the MTCU continues in serial mode. The broadcast operation is essential to the XMT ability to start all TCUs at once in the same time it takes to start one TCU. The PS unit allows allocation of new threads to the TCUs that just became available within the same time as allocating one thread to one TCU. This dynamic allocation provides run-time load-balancing of threads coming from an XMTC program.

We are now ready to connect with the NBW FSM ideal. From the moment the MTCU starts executing a spawn command until each TCU terminates the threads allocated to it, no TCU can cause any other TCU to busy-wait for it. An unavoidable busy-wait ultimately occurs when a TCU terminates and begins waiting for the next spawn command.

TCUs, with their own local registers, are simple in-order pipelines, including fetch, decode, execute/memory-access, and write-back stages. A cluster includes functional units shared by several TCUs and one load/store port to the interconnection network shared by all its TCUs.

The global memory address space is evenly partitioned into the MMs through a form of hashing. The XMT design eliminates the cache-coherence problem, a challenge in terms of bandwidth and scalability. In principle, there are no local caches at the TCUs. Within each MM, the order of operations to the same memory location is preserved.

Quite a few performance enhancements have been incorporated into the XMT hardware, including compiler and run-time scheduling methods for nested parallelism and prefetching methods.

5.2.2 NoC (Network on Chip)

The high-throughput interconnection network required for the XMT architecture presents an implementation challenge. A unique data path can be provided for each pair of clusters and cache modules, such that there is no blocking in the network, using a mesh of trees (MoT) network. However, the number of switches required is proportional the product of the number of clusters and the number of cache modules, which translates to a large silicon area. For example, an XMT architecture in 22 nm technology with 8k TCUs requires silicon area of 190 mm^2 just for an MoT NoC. The area required for an MoT NoC of an XMT architecture with 16k TCUs is 760 mm^2, and would not fit on a single silicon layer. In order to reduce network area, a hybrid MoT and butterfly network can be used, where the inner levels of the "pure" MoT network are replaced with butterfly levels [14].

5.3 Motivation for using the XMT framework in this chapter

Our choice to use XMT in this chapter is motivated by several factors, described below.

5.3.1 Ease of experimentation

A practical reason for using XMT is the availability of XMTSim, a cycle-accurate simulator of the XMT architecture. XMTSim allows setting various architectural parameters such number of clusters, number of cache modules, and number of DRAM ports, which determines bandwidth to DRAM. This allows us to model the various configurations given in Section 5.4. XMTSim and the XMTC compiler are described in [100] and have already been the basis for several publications including [29]. The most recent validation of the cycle-accuracy of the simulator is [101, Chapter 4], which shows that the simulator cycle counts match those of the FPGA except in a minority of cases, where the discrepancy may be up to 33%, due in part to interconnect and DRAM technology limitations in the FPGA prototype that would not exist in an ASIC product. For the FFT, the difference due to these limitations is 5%.

5.3.2 Past XMT Speedups

For placing this debate in historical context, recall that claims that the main reason that parallel machines provide limited speedups is that the bandwidth between processors and memories is so limited are not new, as formally demonstrated

	Algorithm	XMT	GPU/CPU	Factor
	Graph Biconnectivity [52]	33X	4X, but only on random graphs	≫8
	Graph Triconnectivity [53]	129X	Only serial result	129
	Max Flow [28]	108X	2.5X	43
Burrows Wheeler Transform - bzip2	Compression [55]	25X	X/2.5 on GPU	70
	Decompression [55]	13X	Only serial result	11

Tab. 5.1: XMT Speedups

in [114, 166].

PRAM is the main theory of parallel algorithms. A “proof-of-performance” with respect to PRAM algorithms demonstrated speedups between 1 and 2 orders of magnitude (up to 129X) on the most advanced parallel algorithms in the literature relative to the best known results on any machine (e.g., on GPUs) for any algorithms for the same problem. See Table 5.1. Other published speedups include 20.4X on a 64-TCU XMT versus 4X on a 16-core AMD (using the same silicon area) for FFT [141] and 100X on a gate-level simulation benchmark suite [80].

5.3.3 Ease of programming

For brevity, we refer interested readers to Section 5 of [52] for an extensive discussion of results demonstrating ease of programming on the XMT platform.

5.4 Experimental configurations

A goal of this chapter is to examine the level of enabling technology needed to build various sizes of parallel systems and determine the opportunities that such systems provide to applications. To that end, we choose some configurations of XMT that represent what can be achieved with a given level of technology and explain what the barrier is to reaching the following level. For most configurations below, we consider a 2 cm by 2 cm chip (4 cm^2) using 22 nm technology, though the largest ones assume 14 nm technology. These configurations are summarized in Table 5.2, and the required silicon area is given in in Table 5.3.

	4k	8k	64k	128k x2	128k x4
TCUs	4096	8192	65536	131072	131072
Clusters	128	256	2048	4096	4096
Memory Modules	128	256	2048	4096	4096
NoC MoT Levels	14	16	8	6	6
NoC Butterfly Levels	0	0	7	9	9
MMs per DRAM Ctrl.	8	8	8	4	1
FPUs per Cluster	1	1	1	2	4
TCUs per Cluster			32		
ALUs per Cluster			32		
MDUs per Cluster			1		
LSUs per Cluster			1		

Tab. 5.2: XMT Architecture Configurations

	4k	8k	64k	128k x2	128k x4
Technology Node (nm)	22	22	22	14	14
Silicon (Si) Layers	1	2	8	9	9
Si Area per Layer (mm^2)	227	276	380	365	393
Total Si Area (mm^2)	227	551	3046	3284	3540

Tab. 5.3: XMT Physical Configurations

5.4.1 Baseline: 4096 TCUs ("4k")

The smallest configuration we consider consists of 4096 TCUs. This is the largest system we can fit in a single silicon layer using 22 nm technology. This configuration is strictly smaller than the one in the next section and therefore does not require any enabling technologies.

5.4.2 3D VLSI: 8192 TCUs ("8k")

To overcome the area limitation of the baseline configuration, we can split the XMT chip across multiple layers using 3D VLSI. Companion work [132] shows that an 8192-TCU configuration of XMT is feasible using air cooling alone, but not a larger one.

Another issue that arises at this point is off-chip bandwidth. The 32 DRAM channels of this configuration require a total of 6.76 Tb/s of off-chip bandwidth. Using a standard parallel memory interface such as DDR3, this would require about 4000 pins on the XMT processor package. This may already be infeasible, as even the NVIDIA Tesla K40 GPU (with 561 mm^2 of silicon area) only has 2397 pins, and this problem becomes more acute for larger XMT configurations that require more off-chip bandwidth. A high-speed serial interface would allow consolidating a DRAM channel into a few pins. For example, using the 32.75 Gb/s GTY transceivers on the Xilinx UltraScale+ line of FPGAs, a DRAM channel can be reduced to 7 pins. A configuration with 32 DRAM channels would then require just 224 pins.

5.4.3 Microfluidic cooling: 65536 TCUs ("64k")

A significant issue with 3D VLSI is that the middle layers of the stack are thermally insulated from the outside of the chip, and therefore cooling those layers is difficult. One possible solution for cooling the middle layers is microfluidic cooling (MFC), which uses a liquid (such as water) pumped through tiny channels between layers to remove heat. Companion work [132] shows that MFC is sufficient to cool even a 65536-TCU configuration of XMT. At this point, the number of layers in the 3D stack becomes a limiting factor. Off-chip bandwidth also becomes a limiting factor, as even with high-speed serial transceivers, the 256 DRAM channels of this configuration would require a total of 1792 pins.

5.4.4 Photonics and 14 nm node: 131072 TCUs ("128k x2")

For larger configurations of XMT, we need to look ahead to smaller technology nodes. For scaling from 22 nm to 14 nm, Intel claims a scaling factor of 0.54 for logic area and similar scaling for power consumption [23]. If we keep the area of the network-on-chip fixed, this allows us to double the number of clusters and memory modules with some area to spare. With the remaining area, we can add more FPUs. Because we double the off-chip bandwidth per memory module (see below), we choose to also double the number of FPUs per cluster to balance computation capability with communication.

In order to provide sufficient off-chip bandwidth for this configuration, we need to replace the copper interconnect with a more advanced one, such as an optical interconnect driven by silicon photonics. A significant issue with this solution is heat. Faster photonic transceivers tend to be less energy efficient than slower ones. For example, by combining eight 10-Gb/s channels in a single transceiver using wave division multiplexing, it is possible to achieve an efficiency of 600 fJ/bit and an I/O density of 700 Gbps/mm^2 [176]. For a 4 cm^2 chip, this solution provides 280 Tb/s of off-chip bandwidth using 168 W, which is enough to double the ratio of DRAM controllers to memory modules. More recent work achieves higher rates per channel but at the cost of an order of magnitude more power; two approaches using 30 Gb/s transceivers without multiplexing require approximately 3 pJ/bit [49] and 8 pJ/bit [96].

If the photonic transceivers are air cooled, then this limits their power dissipation and thus the bandwidth that can be achieved. In 2004, forced air cooling was predicted to achieve little more than 100 W/cm^2 [175, p. 4] to 150 W/cm^2 [160], and this projection has since remained steady [151]. This means that for a 4 cm^2 chip, air cooling can remove no more than 600 W of heat. In this case, the 10-Gb/s channels provide more bandwidth within the power budget than the 30-Gb/s ones.

Another limit at this point is the number of through silicon vias (TSVs) that connect to the network-on-chip (NoC). A practical limit to the number of TSVs on a single layer may be one hundred thousand [161], as beyond this point manufacturing cost quickly increases and total TSV footprint becomes a significant percentage of silicon area. The width of a NoC port is 50 bits; at 3.3 GHz, the required bandwidth is 165 Gb/s per port. Each TSV can operate at 40 Gb/s [154, 169], so five TSVs are required per port. A 131072-TCU configuration with 4096 clusters and 4096 cache modules will require 20480 TSVs for each of the following: from the NoC to processors, from processors to the NoC, from NoC to memory modules, and memory modules to NoC. This is a total of 81920 TSVs, which allows eighteen thousand TSVs for other purposes, namely power delivery. Assuming a TSV pitch of 12 µm [137], one hundred thousand TSVs will require 14.4 mm^2 of silicon area.

5.4.5 MFC-cooled photonics: more off-chip bandwidth ("128k x4")

Although silicon area limits the size of the XMT chip, there is still room for growth. Namely, the amount of off-chip bandwidth could be increased by applying microfluidic cooling to the photonic transceivers as well as the rest of the chip. This would allow using smaller, faster photonic transceivers, which would provide sufficient bandwidth to allow each memory module to have its own DRAM controller rather than sharing bandwidth with other memory modules. We also increase the number of FPUs to four per cluster; beyond this number, we observe diminishing returns.

Another possible application of MFC-cooled photonics is to split the XMT floorplan across multiple chips at the interface between clusters (and/or memory modules) and the network-on-chip. This would allow for reducing the height of the 3D VLSI stack on each chip without reducing the system size. With sufficient off-chip bandwidth, it would even be possible to split the network-on-chip across multiple chips. It is up to future technology development to indicate which approach works better.

5.5 Results

We use XMTSim to obtain cycle counts for computing a single-precision, complex 3D FFT with an input of size $512\times512\times512$. We assume that the clock speed of XMT is the same as that of the Intel processor used as the reference for our speedup figures, namely 3.3 GHz. To allow comparison with other work on the FFT (e.g.,

[150]), we report FLOPS based on the standard rule of $5N \log_2 N$ floating-point operations for an FFT of N elements. An exception to this is Section 5.5.2, as the Roofline model defines FLOPS to be the actual number of floating-point operations (as reported by XMTSim) per second.

5.5.1 Comparison to FFTW

Serial FFTW We evaluate the performance of our implementation of FFT on XMT for the configurations given in Table 5.2 by comparing it to an existing highly-optimized implementation of FFT, namely FFTW version 3.3.4. The baseline for our speedups is serial FFTW running on one core of an 8-core Intel Xeon E5-2690 with 20 MB of cache. Performance in GFLOPS is in Table 5.4, and speedup results are in Table 5.5.

Configuration	4k	8k	64k	128k x2	128k x4
GFLOPS	239	500	3667	12570	18972

Tab. 5.4: FFT Performance on XMT

Configuration	4k	8k	64k	128k x2	128k x4
Speedup vs. serial	31X	66X	482X	1652X	2494X
Speedup vs. 32 threads	2.8X	5.8X	43X	147X	222X

Tab. 5.5: Speedups relative to FFTW

Parallel FFTW The E5-2690 uses 416 mm^2 of silicon area in 32 nm technology. If we assume an ideal scaling to 22 nm, then the E5-2690 would use about 197 mm^2 in 22 nm technology. This implies that the 4k configuration of XMT would use about 1.15 times as much silicon as an E5-2690. We ran parallel FFTW on a system consisting of two E5-2690 processors, which supports up to 32 threads (16 cores with hyper-threading). Notably, the 4k configuration achieves a 2.8X speedup relative to this system while using only 58% of its silicon area.

5.5.2 Evaluation using Roofline model

Speedup results provide useful information, but limited insight. In particular, they do not establish that the problem cannot be solved more quickly, even on the same platform. Because the FFT is regular, its performance can be analyzed by comparison with the peak performance that the platform is capable of.

The Roofline model [172] describes a platform in terms of two parameters: peak computation rate and peak off-chip bandwidth. Peak computation rate is often (but not necessarily) measured in terms of floating-point operations per second (FLOPS), while bandwidth is measured in bytes per second. These two parameters are plotted on a graph whose y-axis is FLOPS and whose x-axis is computational intensity, the ratio of computation to data movement (measured in FLOPs/byte). Algorithms with low computational intensity are data bound; such algorithms fall under the sloped portion of the graph. Algorithms with high computation intensity are compute bound; these fall under the horizontal portion of the graph. Based on a constant-factor analysis of the number of operations performed by the FFT and its I/O complexity, an upper bound for the computational intensity of the FFT is $\log S$ FLOPS/word [62], where S is the size of the last-level cache in words; for single-precision floating-point numbers, this is $0.25 \log S$ FLOPS/byte.

Our multidimensional FFT implementation consists of two phases that are executed once per dimension. First, the FFT of each row is computed. Second, the axes of the array are rotated[2] so that the next time the FFT is applied to the rows of the array, it will actually compute the FFT of what was originally the columns of the array. In our implementation, the rotation is combined with the last iteration of the computation to reduce the number of synchronization points and round trips to memory.

In Fig. 5.2, we show how the observed performance of the overall FFT computation and its two phases compares to the theoretical Roofline model of the tested configurations of XMT. The rotation phases are communication intensive and thus fall to the left of the non-rotation phases, which are more computation intensive. The overall performance of the algorithm is equal to the weighted average of the two phases with respect to the time each cycle takes, so the overall performance falls on the line connecting the two phases. The overall performance is closer to the non-rotation phase since the non-rotation phase takes the majority of the time.

We make the following observations about the results:

(a) In the 4k and 8k configurations, both phases are essentially on the sloped line, indicating that they operate very close to the peak off-chip bandwidth.

(b) In the 64k configuration, the rotation step is beginning to fall below the sloped line. Since virtual parallelism is not lacking, this must be due to the decreased number of mesh-of-trees levels in the interconnection network (ICN), a result of the constraint on interconnection network area. This effect is more pronounced in the 128k x2 configuration, which has even fewer mesh-of-trees levels.

(c) The 128k x4 configuration provides only a 51% improvement over the 128k

[2] In the special case of a 2D array, rotation is equivalent to a matrix transpose.

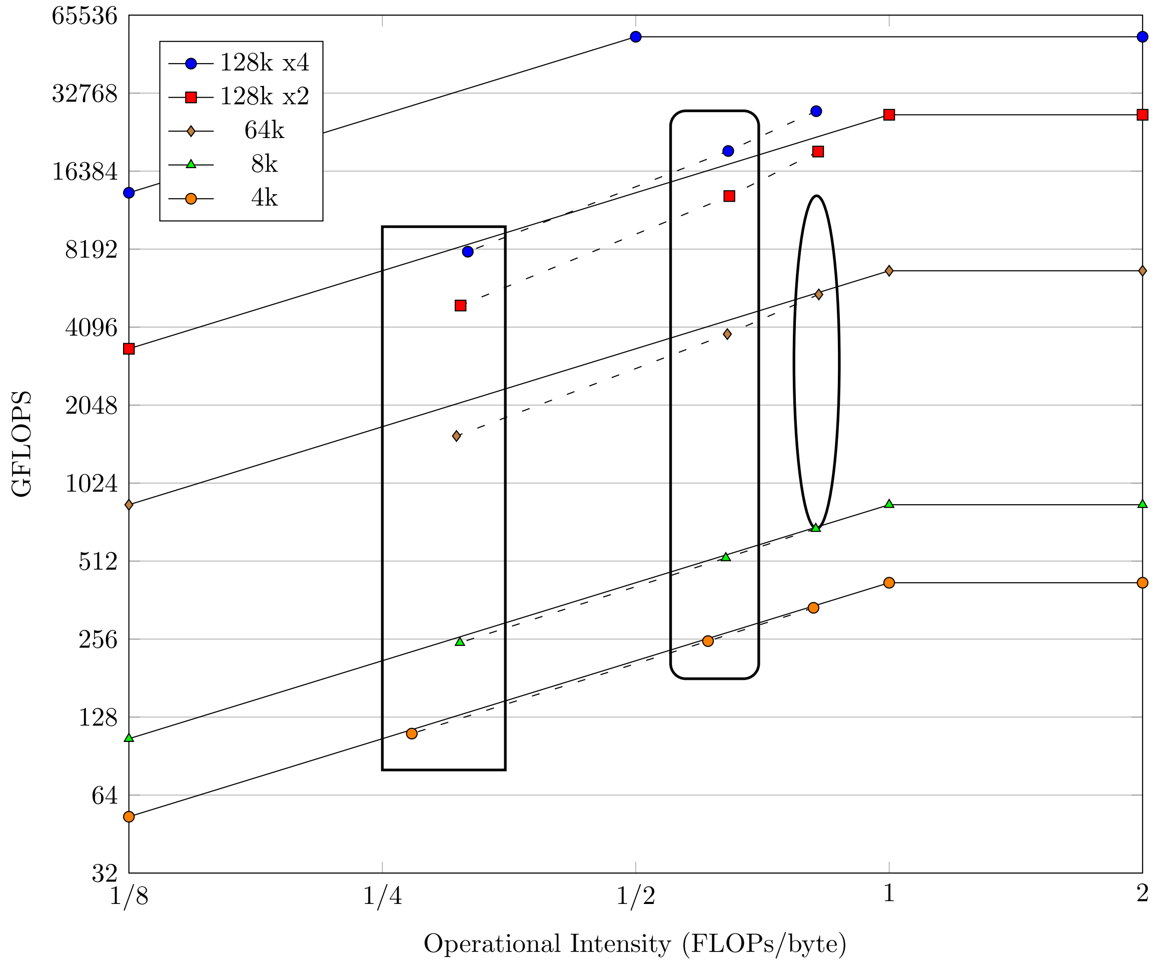


Fig. 5.2: Roofline model of each XMT configuration (solid line with markers) with empirical results for 3D FFT (markers on dashed line). On each dashed line, the marker on the left (inside rectangle) corresponds to iterations where rotation is performed, the marker on the right (inside ellipse) corresponds to iterations where no rotation is performed, and the marker in the middle (inside rounded rectangle) is for the overall FFT algorithm.

x2 configuration. As in (b), this is because the ICN is the bottleneck, and increasing the bandwidth to DRAM beyond that of the 128k x2 configuration does little to reduce congestion in the ICN.

Future technology scaling should allow for a more dense network-on-chip, which would alleviate the bottleneck and allow for an even larger configuration of XMT.

5.5.3 Comparison to Edison

Edison is a Cray XC30 machine consisting of numerous 12-core Intel Xeon E5-2695v2 processors interconnected using a Cray Aries network with a Dragonfly topology. Edison is an enormous machine while even the largest configuration of XMT we consider here is of a much more modest size, as shown in Table 5.6. For example, in order to facilitate comparison of the silicon area required by the two systems, following the row that compares total actual silicon areas and the VLSI process used, we present areas normalized to 22 nm technology.

	Edison	XMT (128k x4)
# processing elements	124,608 cores	131,072 TCUs
# processor groups	5,192 nodes	4,096 clusters
Total cache memory	311,520 MB	128 MB
# chips	10,384 CPU + 1,298 router	1
Total silicon area (process)	56,177 cm^2 (22 nm) + 4,072 cm^2 (40 nm)	35.4 cm^2 (14 nm)
Normalized silicon area (22 nm)	57,409 cm^2	66 cm^2
Peak power consumption	2,500 KW	7.0 KW
Peak teraFLOPS	2,390	54
TeraFLOPS for FFT (size)	13.6 (1024^3)	19.0 (512^3)
% of peak FLOPS	0.57%	35%

Tab. 5.6: Comparison of Edison machine (Cray XC30) to XMT

The 128k x4 XMT system achieves a 1.4X higher speedup than Edison even though the latter requires 870 times the silicon area and 375 times the power of the XMT system. To put the power consumption of the XMT chip into perspective, microfluidic cooling can remove nearly 1 KW/cm^2 of heat per layer; see [157] and

[24] for examples of single layer microfluidic cooling prototypes that have removed 790 W/cm^2 and 681 W/cm^2 respectively.

5.6 Conclusion

We have shown the potential for significant speedups relative to off-the-shelf platforms on the FFT, an important mathematical algorithm. In contrast, without co-design of algorithms and architectures, strong speedups have been elusive. This suggests that it is indeed worth investing further effort into development of a cohort of enabling technologies including silicon photonics for affording higher bandwidth.

Chapter 6: Boosted decision trees (XGBoost)

6.1 Introduction

Since the circa 2004 transition of mainstream computing to parallelism, efforts of the research community have been centered around commercial multi-core or GPU hardware, unwittingly ceding strategic intellectual leadership of the field to vendors. Extensive work on mapping and tuning algorithms and performance programs for a generation of products has dominated contemporary conferences, journals and research dissertations. Vendors have been changing their designs at a rather brisk pace rendering this work Sisyphean: Even for cases where a vendor and a product line remained in business, this work had to often be redone for successive generations, sometimes ab initio. System architecture research has not fared much better. The focus of the "quantitative approach" is on exploring limited updates to commercial systems. As committee peer review is required to rank technically incomparable submissions, it unwittingly conforms with dominant modes of operation. Thus, publication and funding incentives risk upholding futile efforts. The state of educating CS undergraduates to properly benefit from parallelism widely available in the very machines they use suggest further alarming evidence. Reflecting a rather broad computer systems community, the sixth edition [85] justifies a recent shift to heterogeneous platforms by stating: “it seems unlikely that some form of simple multicore scaling will provide a cost-effective path to growing performance”. However, we are concerned that such a shift may augment Sisyphean efforts with Babel-Tower-type problems, making a bad situation even worse.

We believe that promoting fundamental understandings and robust knowledge, independent of commercial players, is key to basic academic research. Thus, it would make sense for us as a community to ask whether we can do better.

A completely different approach, dubbed “Explicit Multi-Threaded (XMT)”, is discussed in [164]. The lead immediate concurrent execution (ICE) abstraction underlying PRAM, the main theory of parallel algorithms, is the concept that each step of a program needs to state all the operations that can done concurrently and assume their lock-step execution in unit time, but nothing else. The horizon envisioned by XMT is that of having the parallel programmer express parallelism using ICE without ever needing to be concerned with threading, race conditions or locality, while achieving competitive performance. A vertically integrated hardware/software on-chip system has been introduced and extensively prototyped, demonstrating speedups by order of magnitude over same-generation commercial platforms for irregular and fine-grained applications. Removing the last obstacles to efficiently implementing textbook PRAM algorithms as-is, this effort culminated in demonstrating [70] that ICE programs can fully match the performance of manually optimized multi-threaded code on XMT, thereby establishing feasibility of the XMT-envisioned horizon. But, would an XMT/PRAM/ICE approach lead to more robust insights? For algorithms the answer is yes. The PRAM algorithms theory has been stable since the 1980s. Using PRAM algorithms as-is per [70] is very appealing and holds promise, especially if supported by architecture as for XMT. But, what are the prospects that architecture insights and, in particular, memory architecture ones meet the test of time?

The hybrid memory architecture underlying the XMT many-core computer features: (i) A master CPU with a traditional cache ("serial mode") and a plurality of CPUs using shared memory cache; none of the parallel CPUs has local write caches. And (ii) low-overhead transition between these serial and parallel memories. In conjunction with a high-bandwidth, on-chip, all-to-all interconnection network, these allow competitive performance regardless of how much parallelism a given code presents.

This chapter presents: (i) new evidence that both multi-core and GPU design have been getting much closer to this hybrid memory architecture given their original starting principles that guided them in the opposite direction, reasoning that their current quest for more effective support of fine-grained irregular parallelism drew them closer to such memory architecture; and (ii) new speedup results of 3.3X over NVIDIA's most powerful GPU to date for XGBoost, a timely machine learning algorithm.

There are several reasons why we believe that this chapter can be stimulating:

- It provides a unique perspective. In particular, raising the provocative question about the aforementioned concerns about the recent shift to heterogeneous platforms is likely to get some commotion from at least some of the audience.

We hope that memory and system researchers will realize the need for operating outside the spell of incremental improvements to commercial systems, and the opportunity for doing that, especially once such incremental approaches are contrasted with the PRAM-based option to do better on both robustness and homogeneity.

- The evidence on multi-core and GPU design getting much closer to the XMT hybrid memory architecture raises the question whether the fundamental nature of parallel algorithms and programs may have a similar effect to gravitation power, drawing us in a certain direction regardless if we are aware of it or not.

- The above mentioned demonstration of ICE programming on XMT, along with success stories such as having XMT programming taught to about 700 students in a single high school (Thomas Jefferson High School for Science and Technology, Alexandria, VA) since 2009, suggest that providing simple multicore scaling and a cost-effective path to growing performance may not be as insurmountable as [85] and many other computer architects opine.

Section 6.2 of this chapter discusses the hybrid memory architecture underlying XMT. Section 6.3 briefly describes boosted decision trees, a timely application which we use to show the benefit of our hybrid memory architecture, as well as the results we obtained for this application. Finally, Section 6.4 evaluates some choices we made in the design of XMT.

6.2 Memory architecture of XMT

6.2.1 Our goal

One old insight of XMT is the need to support effectively in one architecture two memory paradigms: serial and parallel. Many programs consist of both serial sections of code and parallel sections, potentially with varying degrees of parallelism. Amdahl's Law implies that speeding up one section alone will be of limited benefit; to improve performance beyond a certain point, all sections must be sped up. In addition to the need for strong serial support for programs for which no parallel implementation is currently available, serial execution also shows up in more subtle ways in parallel programs. First, portions of some parallel programs may have limited parallelism, and in such cases it may be faster to execute those portions on a strong serial processor rather than underutilizing the parallel processors. Second, programs with fine-grained parallelism, even those with much available parallelism, need to switch between serial and parallel modes of execution frequently to orchestrate the spawning and synchronization of threads; for example, when parallelism is not represented by long running threads communicate infrequently, rarely or not at all, and lower overheads for switching to serial mode and back to parallel mode justify this.

From the memory architecture point of view, there is a tension between the goals of supporting serial and parallel computation. Serial code is more sensitive to memory latency than to bandwidth, as there is limited opportunity for a single thread to hide latency, while parallel code can issue many requests in parallel to hide latency. Reducing latency often requires bringing data closer to the processor, such as in a private cache. On the other hand, parallel computation often requires sharing data among processors, and protocols to maintain coherence among private caches scale poorly.

In light of this, we have developed a hybrid architecture with two components: (1) a "heavy" serial processor with a private writable cache and (2) a number of "light" parallel processors, each without a private writable cache. The two components are tightly coupled such that switching from serial to parallel and back can accomplished in time on the order of 10-100 cycles.

To develop this hybrid architecture, we first considered what limits performance from the algorithm side. Under the PRAM algorithmic model, the relevant factors are (1) work, the total number of operations to be performed and (2) depth, the length of the critical path of execution. From the memory architecture point of view, work comprises the amount of data read from and written to memory, and depth is dominated by the length of the sequence of round trips to memory (LSRTM). We then asked, for a given choice of workload, what is the best LSRTM that can be achieved from the parallel algorithm side. After performing this optimization by hand, we set out to automate this task. Through iteration of the hardware as well as the software development toolchain, we refined the automation of the process of achieving a given LSRTM.

6.2.2 Our design choices

Here, we describe our lead design choices for the two components of our memory architecture and how they work together. See Fig. 6.1. This is just one possible set of choices we could have made; for a discussion and evaluation of design choices, see Section 6.4.

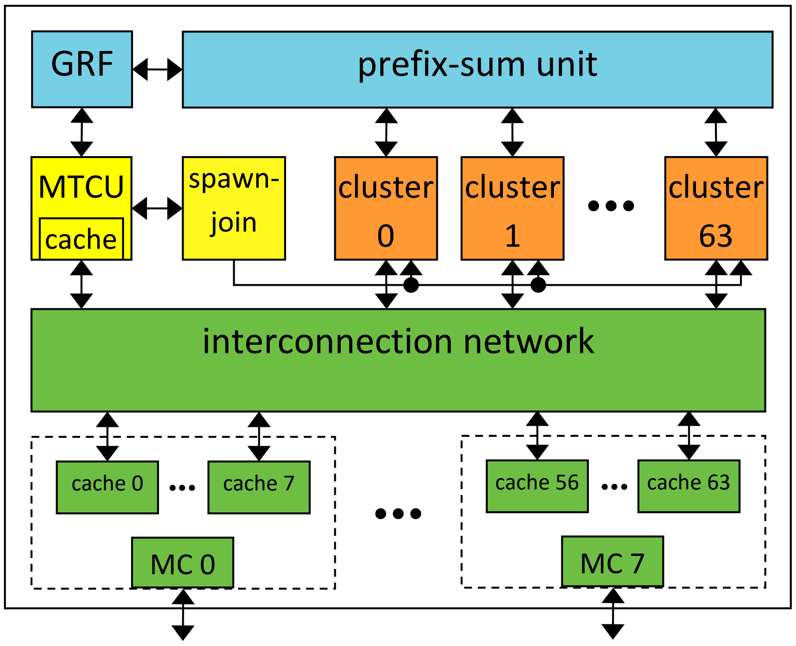


Fig. 6.1: Block diagram of hybrid memory architecture of XMT. The serial portion comprises the MTCU, which includes a local cache. The spawn-join unit (yellow) is used for transitioning between serial and parallel mode from the control side, which is a bit suppressed in this current memory-centered chapter. The parallel portion comprises the clusters (orange). The shared memory system comprises the interconnection network, shared caches, and memory controllers (green); it is used by both the serial and parallel portions. The global register file (GRF) and prefix-sum unit (blue) are used to coordinate concurrent execution of threads.

Serial mode

In serial mode, a single master thread control unit (MTCU) executes code. The MTCU is a standard serial processor core with its own private, writable cache. We choose a write-through no-write-allocate policy for the cache to reduce the potential for data to be brought into cache unnecessarily, which helps reduce the time needed to flush the cache. The MTCU private cache is connected to shared memory via a port on the interconnection network just like the TCUs.

Parallel mode

In parallel mode, a number of thread control units (TCUs) execute the program contained within the current parallel section of code (delimited by “spawn” and “join” instructions in XMT). TCUs are grouped into clusters (typically 16 TCUs per cluster) that share some resources including a single port to the shared cache.

TCUs lack private writable caches, instead writing directly to main memory via the shared cache. Several read-only memories are used to reduce the latency, and in some cases bandwidth, of accesses to shared memory:

- Each TCU stores a copy of the program in a local instruction buffer. This allows TCUs to run at their own pace rather than in lockstep.
- TCUs contain software-managed prefetch buffers, which reduce latency by allowing TCUs to send read requests to memory before they will be needed by the program and reduce LSRTM by allowing TCUs to issue reads back-to-back without waiting for them to complete one-by-one.
- Clusters contain read-only buffers, which are software-managed caches that allow TCUs to reuse data read by other TCUs in the cluster.

The shared cache is partitioned into cache modules, where each module is backed by a partition of the global memory space. Clusters communicate with the cache modules via an all-to-all interconnection network (ICN). For smaller configurations of XMT, the ICN is a mesh-of-trees network (MoT); for configurations where a pure MoT would be too large, a hybrid network is used instead where some of the middle layers of the MoT are replaced with layers of a butterfly network. All access by the TCUs to shared memory goes through the ICN.

Finally, the cache modules are connected to main memory (DRAM) via one or more memory controllers, which are evenly partitioned among the cache modules.

Requests by multiple TCUs in a cluster are queued, as are requests to the same cache module. Requests by the cache modules to the memory controllers are also queued. We hash memory addresses to spread memory accesses more evenly across the cache modules to reduce hot spots.

Transition from serial to parallel

When spawning threads, the MTCU first flushes its private cache to shared cache. This ensures that all data is available to the TCUs without the need for cache coherence protocols. Assuming that not too much data is brought into the local MTCU cache, the flush will be efficient. A possible optimization here would be to flush only those cache lines containing data that will be needed in parallel mode.

Then, starting immediately after the spawn instruction, the MTCU broadcasts the spawn block to the TCUs one instruction after another. Because each TCU has its own copy of the program and its own program counter, each thread can progress at its own pace.

Transition from parallel to serial

After all threads finish executing, TCUs wait for all outstanding requests to shared memory to complete, and then control returns to the MTCU. All local parallel memories (e.g., read-only buffer) are invalidated; no data needs to be written from local memory to shared memory since the local memories are read only.

6.3 Application: boosted decision trees

An increasingly-popular approach to machine learning is gradient boosted decision trees, as implemented by XGBoost [31]. XGBoost is designed to perform well on serial and parallel CPUs and has recently been extended to be supported by GPUs [122] as well. According to the authors of XGBoost, it has been used by many winners of machine learning competitions, including all of the top-10 winners of KDDCup 2015 as well as many top-3 winners on the popular machine learning competition website Kaggle (acquired by Google in 2017): 17 of the 29 challenge winning solutions published on Kaggle's blog during 2015 used XGBoost, compared with 11 that used deep neural networks. In some senses, Kaggle represents the marketplace for data scientists: companies often sponsor competitions on Kaggle to find solutions to problems of interest to them, and they also use Kaggle for recruiting data scientists, either by evaluating the Kaggle ranks of applicants to data scientist positions or by sponsoring competitions on Kaggle whose purpose is recruiting.

Reducing the training time would be beneficial to users of XGBoost. Indeed, speedups have been demonstrated using GPUs [122], and work continues on reducing times even on CPUs (e.g., the current beta version of the Intel Data Analytics Acceleration Library (DAAL) [92]). However, we conjecture that there is room for further speedups, and we have produced initial evidence to validate this conjecture. GPUs are tuned for approaches such as deep learning that consist mostly of regular operations with high computational intensity such as matrix multiplication and convolution. In contrast, XGBoost relies heavily on irregular operations with low computational intensity, such as sorting, compaction, and prefix sums with indirect addressing. Although support for irregular algorithms on GPUs appears to be improving, it still lags far behind support for regular algorithms.

6.3.1 High-level review of algorithm

A decision tree is a binary tree where internal nodes represent yes-or-no questions about an instance and leaf nodes represent the label to be reported for all instances that lead to that leaf. A simple type of decision tree is one in which each internal node asks whether a certain feature of the input is below or above some threshold, where the choice of feature and threshold are parameters of the model.

On their own, decision trees may be prone to overfitting. One approach to mitigate this is by limiting the depth, and thus the complexity, of the tree. The downside to this is that a single shallow decision tree is a fairly weak model. To compensate for this, multiple decision trees can be trained and their results averaged to produce a stronger model. XGBoost uses a boosted decision tree approach, in which trees are added one by one to refine the output produced by the trees in the model so far.

XGBoost uses a greedy approach to build each decision tree. To begin, XGBoost starts by creating a single leaf node and assigning all of the training examples to that leaf. XGBoost builds the tree by recursively splitting the examples at each leaf so as to produce the highest information gain, stopping when the gain falls below a specified threshold.

The majority of the time taken by XGBoost is spent searching for the best split point (a feature and its threshold) for each leaf. For each possible split, XGBoost looks at the left and right sides of the split and for each side computes a score representing how large of a refinement will be made by this split; the information gain is the sum of these two scores minus the score of the original, un-split node. A simple scoring function provided by XGBoost is to compute the square of the sum of the errors (signed differences) between the true output for each training example and the current prediction.

XGBoost makes use of the following insight: if the training examples are sorted in order of increasing value of a given feature, then all possible splits for that feature can be trivially found by walking through the list. Furthermore, the sums of errors can be updated while walking through the list simply by subtracting the error of the current element from the sum for the right side and adding it to the sum for the left side. This implies that the sums that are needed are the prefix-sums of the errors in this sorted order. Because a different sorted order is needed for each feature, the order in which the training examples are accessed is constantly changing, leading to an irregular memory access pattern.

Overview of parallel algorithm on XMT

Our parallel algorithm for XGBoost on XMT takes the serial algorithm and replaces each step with a corresponding parallel alternative:

- We sort the training examples using a shared-memory sample sort. In contrast to work on GPUs [122] that uses radix sort, this is less regular but provides more parallelism for some inputs. We do not rearrange the examples themselves in memory but instead maintain arrays of pointers to the examples in sorted order, one per feature. This results in more irregular memory access in later steps but saves work when splitting nodes.

- To compute the sums of errors, we use a parallel prefix-sums algorithm. This is similar to [122]; however, XMT can exploit more parallelism in this step than GPUs: on XMT, all TCUs can participate in computing the prefix-sums for a single feature whereas the GPU algorithm only uses a single thread block per feature.

- To find the split with the maximum score, we use a parallel reduction algorithm with maximum as the associative binary operator. Again, this is similar to [122] but without the limitation of one thread block per feature.

- To split each node and rearrange its associated examples accordingly, we apply parallel prefix-sums to perform compaction. In contrast, [122] employs two strategies to handle splitting: (a) for the first few levels of the tree, do not rearrange the examples. Instead, mark each example with the node it now belongs to after each split. (b) For deeper levels of the tree, rearrange the examples after each split using radix sort.

The XMT algorithm above and the GPU algorithm of [122] represent different trade-offs resulting from the memory architectures of the respective platforms. The XMT algorithm favors reducing algorithmic complexity (work and depth) at the expense of increased irregularity, maintaining pointers to examples and rearranging them as necessary to avoid idle threads in later steps. In contrast, the GPU algorithm maintains regularity as much as possible at the expense of increased work and depth by deferring irregular data movement until the overhead of skipping over examples that do not belong to the current node becomes prohibitive.

We also note that both the XMT and GPU algorithms involve numerous transitions between serial and parallel execution, as most of the above steps are executed many times and each execution of a step incurs multiple serial-to-parallel transitions. The impact of this is discussed in Sections 6.4.4 and 6.4.5.

6.3.2 Method

We compare XMT to commercial CPU and GPU platforms for the machine learning approach of gradient boosted decision trees and obtain significant speedups.

Software To facilitate a fair comparison, we compared our code against the following:

1. XGBoost, both serial and parallel CPU implementations [31]

2. the GPU-accelerated version of XGBoost [122].

Here, we focus on the training step, as it is more time consuming than inference and the parallelism is more difficult to exploit. This is not to exclude inference: although inference on decision forests is embarrassingly parallel across the trees, we still would expect some benefit on XMT since the problem is irregular.

XGBoost is written in C++ (with GPU kernels in CUDA), but there is currently no C++ compiler available for XMT. Therefore, we needed to rewrite the XGBoost algorithm in XMTC, with a focus on computing the same result as the original XGBoost code while exposing some parallelism. This starting point may prove to be a disadvantage here, as XGBoost was designed with the strengths and weaknesses of multi-core CPUs and GPUs in mind, and after more extensive work, we may be able to get better speedups.

Computing platforms To obtain serial and parallel CPU performance results, we ran XGBoost on a modern Linux machine with two 8-core Intel Xeon E5-2690 processors (16 cores in total). To obtain GPU results, we ran XGBoost on an Amazon EC2 p3.2xlarge instance, which includes eight cores of an Intel Xeon E5-2686 v4 CPU and a Tesla V100 GPU (Volta microarchitecture), NVIDIA's most advanced GPU to date.

Results for XMT were obtained using XMTSim, a cycle-accurate simulator of the XMT architecture derived from a commitment to silicon of XMT using FPGA. XMTSim was configured to simulate an XMT processor that would use silicon area comparable to the Tesla V100 (16,384 TCUs, 32 MB shared cache) and also provide nearly the same bandwidth to DRAM (768 GB/s).

XGBoost configuration and dataset XGBoost on all platforms was configured to generate 120 trees with a maximum depth of 6.

The dataset used in this experiment was the Higgs boson dataset taken from the Kaggle machine learning challenge website [93]. It consists of 250,000 training examples with 30 features each. This dataset was also used by the authors of XGBoost in their work.

6.3.3 Speedups

The results show that among the platforms above, XMT would outperform both the CPU and the GPU; see Fig. 6.2.

As far as we found out, other work on parallel implementations of decision forests does not report direct comparisons to the best serial implementation. Work on training tree ensembles using MapReduce [134] did not report any speedup versus best serial due to lack of memory on the serial machine.

Work on boosted trees [158] achieved a self-speedup of up to 42× on a 48-core shared memory machine and up to 25× on a 32-core distributed memory machine, but no results are reported relative to best serial.

6.4 Discussion of design choices

The above is one possible design choice for a hybrid memory architecture, which we made based on looking at various parallel workloads. Up to a point, the community has agreed on the transition from serial computing to parallel computing. Although no description is publicly available, NVIDIA GPUs appear to

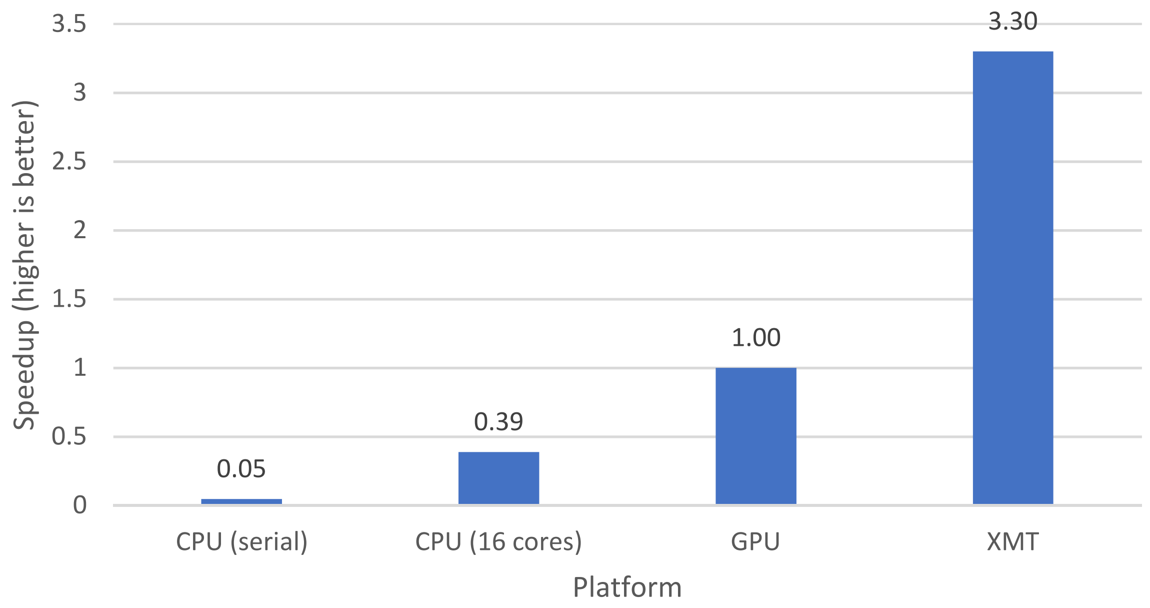


Fig. 6.2: Speedups of XGBoost on various platforms relative to the most powerful NVIDIA GPU. XMT has a speedup of 3.3X while the CPU platforms have slowdowns (speedup <1X).

have a high-performance all-to-all network connecting parallel processors to shared cache. However, the CPU and GPU each have some separate memories, where data shared between them must be copied from one to the other. In addition to discrete GPUs, AMD also produces Accelerated Processing Units (APUs), which combine a traditional CPU and GPU on a single die, and Intel also produces CPUs with an integrated GPU. However, the balance of silicon area between CPU cores and GPU cores on these chips has so far not favored GPU performance as in discrete GPUs.

6.4.1 *Evidence for advances in GPU memory architecture*

Our discovery of recent changes in modern NVIDIA GPU memory architecture is a bit anecdotal. It began with our earlier attempts to run cycle-accurate simulations of programs running on modern GPUs. We used FusionSim [174], based on GPGPU-Sim [12], as a starting point and adapted the included configuration, which was designed to match the NVIDIA GTX 480 GPU (Fermi architecture), to attempt to match the NVIDIA Tesla M40. We used FusionSim rather than GPGPU-Sim alone since we sought to model the transitions between serial and parallel execution in addition to the parallel kernels themselves.

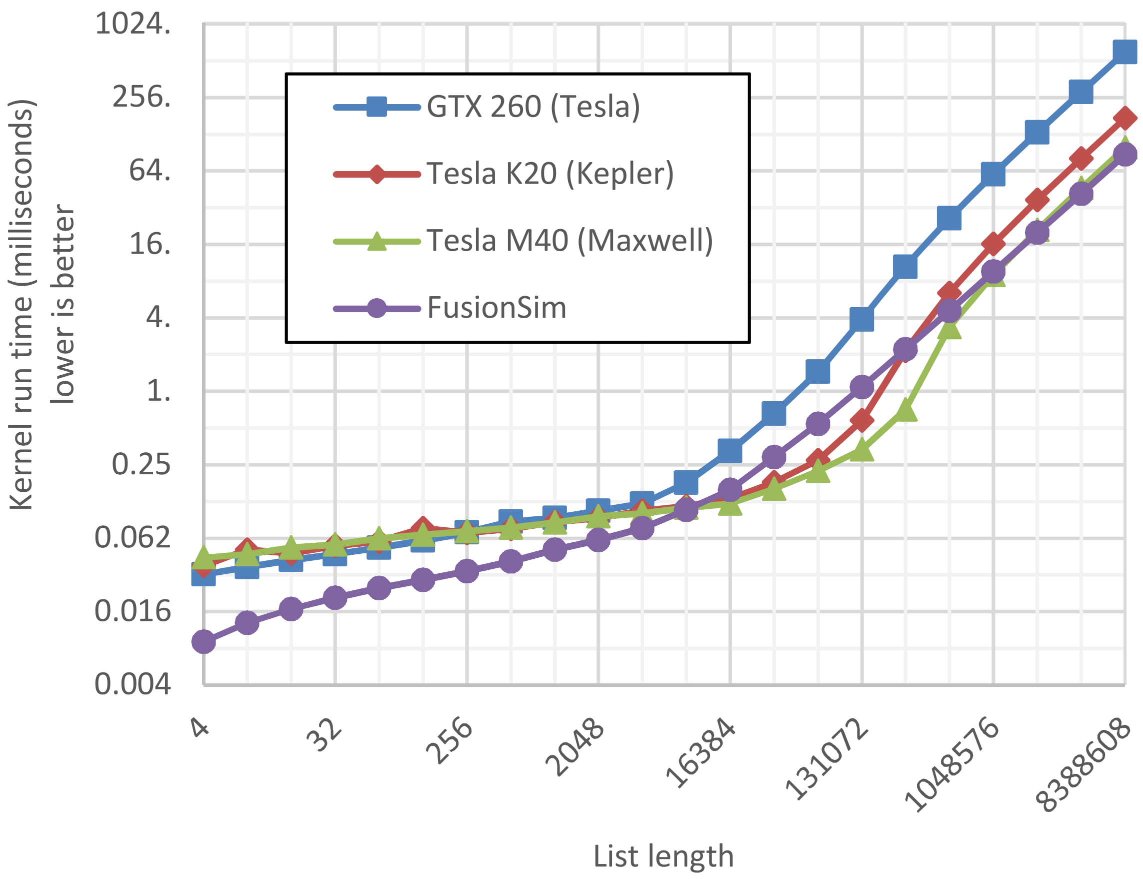


Fig. 6.3: Cycle accuracy of FusionSim (GPGPU-Sim) relative to three NVIDIA GPUs running a list ranking benchmark.

We ran a list ranking benchmark based on parallel pointer jumping as a (highly irregular) benchmark of three NVIDIA GPUs as well as FusionSim. Our goal was to develop a cycle-accurate simulation of the Tesla M40 GPU that we would then use for further work plans. However, we had to abandon our plans since we could not get FusionSim to match the actual performance of modern GPUs. The differences we observed can be seen in Fig. 6.3. For large inputs sizes of 1 million elements or more, FusionSim matches the Tesla M40. However, we point out two discrepancies for lists smaller than this.

First, for small input sizes (less than 8000 elements), FusionSim underestimates the run time of the benchmark relative to all three of the actual GPUs. This implies that there are additional overheads for launching kernels on the actual GPUs that are not reflected in FusionSim.

Second, the more recent Tesla K20 and M40 GPUs exhibit a steeper increase in runtime at around 250 thousand elements than at any other point, but FusionSim does not reflect this; FusionSim more closely follows the older GTX 260 in this respect.

In particular, the second observation above led us to suspect that NVIDIA made some improvements between the release of the GTX 260 in 2008 and the Tesla K20 in 2012. We could not make sense of the nature of this improvement based on published papers. In fact, we found it surprising given the well-cited keynote talk [42] with its claim: "locality equals efficiency"; how can parallel architectures that equate locality with efficiency (and minimizing reliance on non-local memories) provide such strong support for high rates of data movement? So, we felt that we need to dig deeper. To our surprise we found a patent [41] filed five years earlier, which went barely unnoticed in the literature suggesting that NVIDIA is indeed heading in a direction that seems a near opposite of [42]. That is, providing much better support for shared memory at the expense of local memories on its GPUs. Interestingly, [41] still suggests similar motivation to [42]; namely, that it would be better from an energy consumption point of view. However, we have not been able to find support in the literature for improved energy consumption as a result of trading local memories for shared ones. In fact, much of the architecture literature seems to continue being influenced by [42] and its call for limiting data movement. Indeed, when we then looked up information about the streaming multiprocessor in their P100 Volta, we didn't expect to find that even the register file is shared. It will be interesting to find out at the conference how representative is our anecdotal experience. Finally, the extent to which support for low-overhead transition between serial and parallel execution is being followed remains to be seen as GPUs continue to evolve.

6.4.2 Integrated vs. discrete GPUs

Some recent Intel processors have integrated GPUs that share their memory system with that of the CPU cores. Two examples are the Intel Core i5-4690K (with an Intel HD Graphics 4600 GPU) and the more recent Intel Xeon E3-1578L v5 (with an Intel Iris Pro Graphics P580 GPU). A notable difference between these two is that the P580 has 128 MB of eDRAM on the same package, which is used as a level 4 cache. The results we were able to get for the Intel Xeon E3-1578L v5 were a bit inconsistent, which we speculate may be due to the first generations of Intel GPUs with eDRAM not being fully optimized. Therefore, we discuss only the Intel Core i5-4690K in the following comparisons.

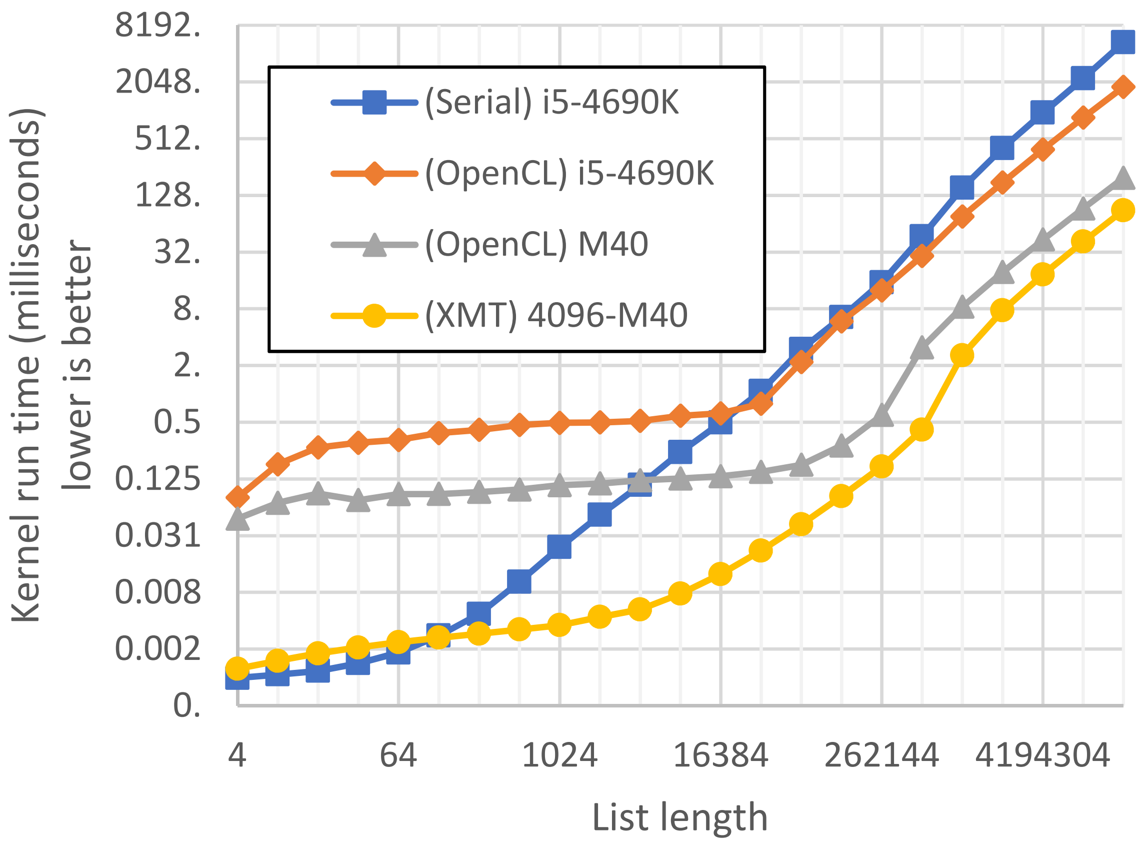


Fig. 6.4: Time taken by various processors on a list ranking benchmark, including a serial CPU (Intel Xeon E5-2686 v4), an integrated GPU (Intel Core i5-4690K), a discrete GPU (NVIDIA Tesla M40), and a simulated XMT system configured to match the Tesla M40.

6.4.3 Performance on irregular algorithms: list ranking

We use the same list ranking benchmark as before to determine whether the integration of the GPU provides an advantage here and to see whether we can detect any improvement due to more recent Intel GPUs being more tightly integrated. See Fig. 6.4. For the largest list (16 million elements), the i5-4690K achieves a speedup versus serial of 2.6X. The integrated GPU is outperformed by the more powerful discrete M40 GPU for all list sizes, which is expected since this benchmark involves little communication between the GPU and the CPU. Notably, XMT performs nearly as well as the serial CPU for small inputs while beating the M40 GPU even for large inputs.

6.4.4 Serial-parallel transition overhead

The GPU version of the boosted decision tree program as tested in Section 6.3 has over ten thousand kernel launches, and the XMT version has nearly as many parallel sections. Here, we examine the overhead of this more closely under two runtimes: OpenCL and OpenGL

OpenCL

Figure 6.5 shows the overhead of switching from serial execution to parallel and back in terms of the time taken to launch an empty OpenCL kernel. Surprisingly, the discrete K20 and M40 GPUs outperform the integrated GPU even for small numbers of threads. XMT spawns threads faster than any GPU by over an order of magnitude when the amount of parallelism is low and remains faster than the GPUs even when much parallelism is available.

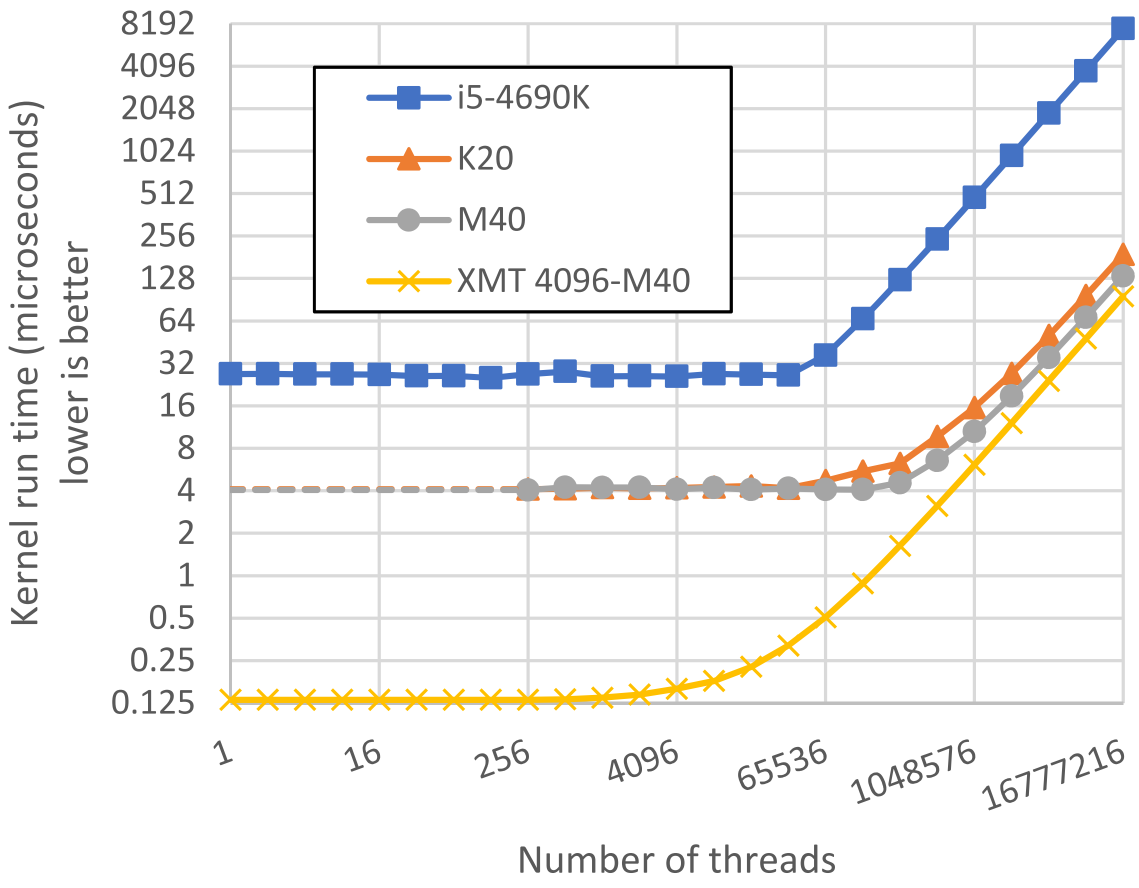


Fig. 6.5: Time taken to launch a single empty OpenCL kernel (spawn block on XMT) with respect to the number of threads launched. This is a measure of the time required to transition from serial to parallel and back. For each platform, the run time starts increasing once the number of threads reaches the maximum the platform can run at a time. For the K20 and M40, we use a minimum of 256 threads (indicated by the dotted line) since these GPUs are not designed for fewer threads; in our tests, running this benchmark with fewer than 256 threads was slower than with 256 threads.

OpenGL

We suspected that the poor performance of the integrated GPU relative to the discrete GPUs may be due in part to the NVIDIA OpenCL runtime being more optimized than its Intel counterpart. In an attempt to avoid the overhead of OpenCL, we tested a short OpenGL graphics benchmark consisting of a single OpenGL shader program that combines two textures (essentially arrays of pixels) using a simple arithmetic operation. Because this is a graphics benchmark, we were limited to running on computers that were configured to allow using the GPU for rendering graphics rather than only for GPGPU computation. Hence, we do not have results for the NVIDIA K20, M40, or V100 GPUs.

Figure 6.6 shows that for inputs up to 2048 pixels, the Intel i5 processor with HD Graphics 4600 is faster than the discrete NVIDIA GTX 1060 GPU. Possible explanation for this advantage can be found in the Memory section of [37].

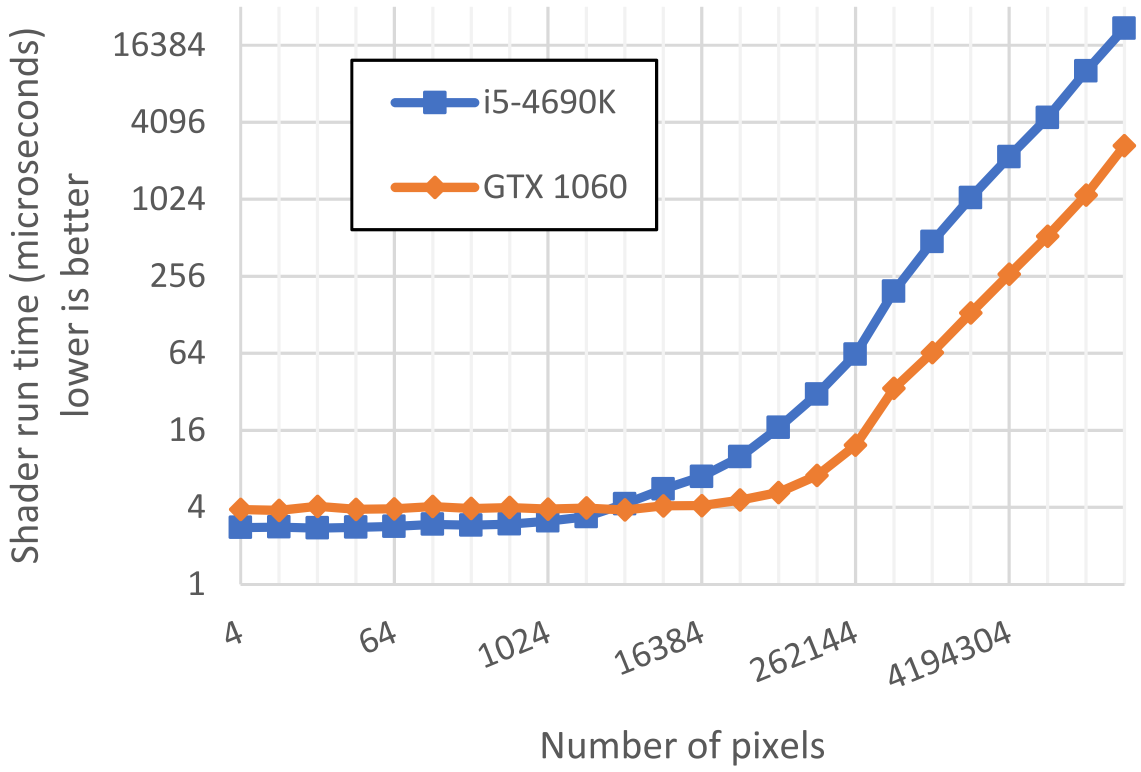


Fig. 6.6: Time to execute a short OpenGL shader that applies a SAXPY operation ($\vec{y} := \vec{y} + a\vec{x}$) to two textures $\vec{x}$ and $\vec{y}$ versus texture size in pixels (length times width). For inputs up to 2048 pixels, the integrated GPU of the Intel i5-4690K processor is faster, indicating lower overhead.

6.4.5 *Sensitivity to serial-parallel transition overhead*

To gain some understanding of the importance of low-overhead transition from serial to parallel in a complete application, we examine what would happen if this overhead were increased relative to baseline provided by the XGBoost results above. In Fig. 6.7, we show the effect of increasing the spawn latency, which is the hardware portion of the transition overhead, from its original value of 23 cycles to various values up to and including 50,000 cycles. Latencies up to about 1000 cycles have little effect on speedup, but performance falls off beyond that point. For comparison, the typical GPU kernel launch latency is around 10,000 cycles. If the overhead for serial-to-parallel transition on XMT were as high as it is for the GPU, then XMT would perform no better than the GPU.

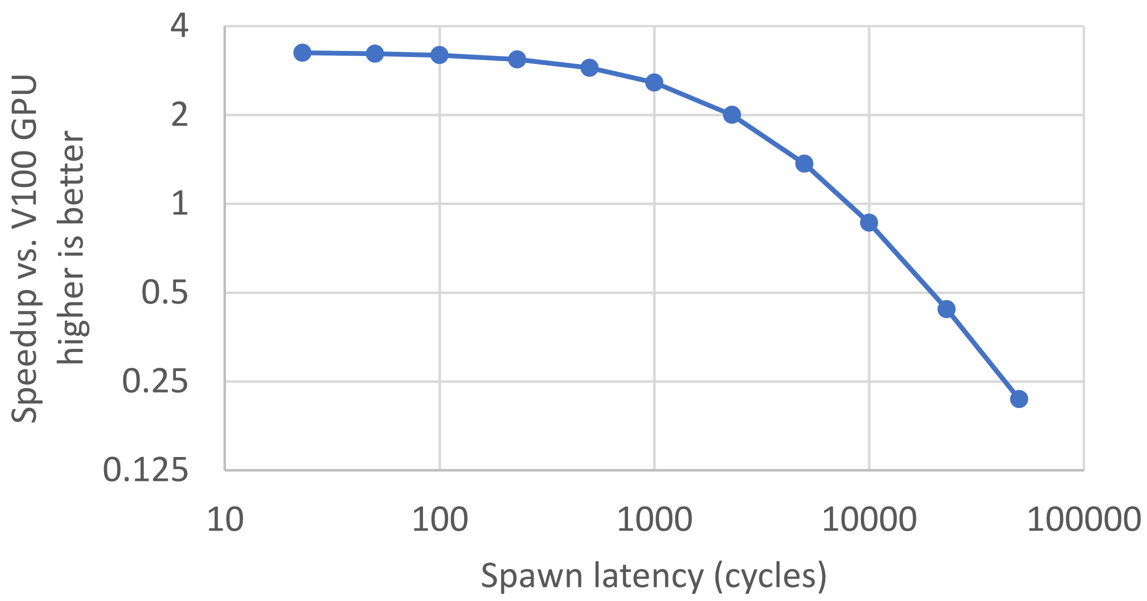


Fig. 6.7: Effect of serial-to-parallel transition (spawn) latency on speedup (log-log axes). The leftmost point is the speedup for the latency

6.5 Conclusion

As their name suggests, streaming multiprocessor memory organizations have long provided strong support for moving data in and out of execution units. However, as long advocated by our XMT/PRAM approach, the need to better support irregular parallel algorithms led some successful GPU designs to increasingly move towards reliance on shared memories, breaking away with their past emphasis on local memories and locality at all cost. While this has led to marked improvements, their limited ability to support down-scaling of parallelism, especially for discrete GPUs, is hurting them significantly for supporting some full applications. The emphasis of some sections of the machine learning market on methods such as stochastic gradient descent (SGD), and their reliance on full matrix multiplication, for deep learning, appears to take a toll for other prominent market success stories in machine learning, such as the boosted decision trees application discussed in this chapter.

However, our experience with XMT suggests that something bigger is at stake here. We demonstrated strong speedup on general-purpose applications, full support of the main theory of parallel algorithms and easy parallel programming; and, therefore, directions for finally providing simple multicore scaling and a cost-effective path to growing performance, finally overcoming what [85] and many other computer architects suggest is insurmountable.

Chapter 7: Boolean satisfiability (SAT)

7.1 Preview and introduction

Boolean Satisfiability (SAT) is a well-known NP-complete decision problem, and a widely used modeling framework for solving combinatorial problems. In spite of its widely assumed worst-case exponential time, modern SAT algorithms have been extremely effective at coping with large search spaces leading to their use in a broad range of practical applications. The overview sources [76, 77, 117] note such applications, including hardware verification, software verification, and, more generally, those that exploit the problem's structure when a structure exists; such approaches were explored over the years in many submissions to SAT competitions.

Additionally, SAT solvers leverage much broader forms of automated reasoning (e.g. [26, 65, 81, 106, 111, 124, 126, 127, 142, 168]), and other fields (e.g., bioinformatics [113], or computer security [121]).

The objective of this chapter is to explore the untapped parallelism in SAT solvers, with the goal of stimulating future research on (i) its scaling and (ii) realization of that potential through novel computing stacks comprising algorithms and architecture.

State-of-the-art approaches to parallel SAT solving focus on splitting the workload among CPU nodes as follows. The nodes handle mutually exclusive truth assignments. Each node runs a sequential SAT solver. The only coordination needed among nodes is the split of work and the load balancing needed in case a node runs out of work. The overall objective is to use parallelism for covering more search space at once, with little interaction between the solvers.

Such approaches map well to current high-end computer systems, with one or more sequential solvers per node. However, these approaches have so far only demonstrated a limited scalability horizon as simple strategies can result in redundant work and more complex ones can encounter high overheads due to frequent communications or significant sequential computation up front or both.

Using parallelism to speed up a state-of-the-art sequential SAT solver As sequential SAT solvers have greatly improved over the years, doing basically the same operations that such a solver does, but enhancing its speed by doing many more of these operations in parallel is an opportunity that we believe has received relatively little attention. If successful, this would have several benefits: 1. Problems that are not too large would fit on a single node and can be sped up without needing to spread the computation across multiple nodes. 2. Because current parallel approaches largely use sequential solvers as black boxes, speeding up those solvers would speed up the overall computation. 3. Parallelism can speed up the partitioning of same-input-different-truth-assignment among nodes, as in the cube-and-conquer method [87] discussed in Sec. 7.4.2, which we hope will spur further research into efficient partitioning algorithms.

The heuristics challenge presented by such work is that, when viewed from the traditional worst-case analysis point of view, SAT solving is as much of an art as it is a science. We must remember that the SAT problem is NP-complete. So, unless P turns to be equal to NP, we are not going to have an efficient SAT solver across all inputs. It was rather the trial-and-error heuristics development approach, guided by common sense and quantitatively evaluated by empirical benchmarking, that led to past successes. The best modern SAT solvers have built on decades of experience addressing the challenges posed by practical SAT problem domains. Resorting to quantitative empirical approaches is of course not alien to computer science; in fact, such approaches have come to dominate fields such as computer architecture, high performance programming and AI including machine learning, among others. For SAT solvers going forward, this suggests preserving as much as possible the "wisdom" that made a (serial) heuristic successful, and to expect similar development pains where the novel parallel SAT solver deviates from the successful serial ones.

Our work exposes an **architecture challenge** facing such work, articulating a gap between current computer architectures and parallel algorithms. Specifically, our initial investigation suggests that efficient handling by the hardware of very fine-grained nesting of parallelism would be critical for effective support of the parallel algorithms we need; in particular, how to minimize the overhead for the initiation of child threads. Traditional software nesting techniques have incurred relatively high overheads. While Vishkin and Wen [171] proposed low overhead hardware support for fine-grained nesting, it appears that most others were content with software techniques, that incur higher overheads. Alternative new hardware techniques for handling of fine-grained nesting may be required, but such techniques, and of course their performance, are still unknown at this time. Thus, the **modeling challenge** to be addressed is: given a parallel SAT solving algorithm, how to reason about its performance potential hypothesizing such future hardware support?

In this chapter, we explore how these challenges can stimulate future work in three inter-related ways:

The first is to study sequential SAT solver **algorithms** to discover opportunities for parallelism. In this chapter, we study fine-grained parallelizations of Glucose [9], a state-of-the-art general-purpose sequential SAT solver, that preserve its carefully developed heuristics. As Glucose uses the same algorithmic framework as most other modern SAT solvers, we expect that insights gained from working with Glucose can be applied to SAT solvers more broadly.

A second is to develop mathematical **models** to determine the potential speedups that can be obtained from new parallel algorithms. Because such work will consider the effect of both algorithms and architectures on performance, a way is needed to evaluate the behavior of the algorithm on real-world inputs apart from specific architectures in order to isolate the sources of bottlenecks. The model we consider here is extending **work-depth**, a widely-used abstraction in parallel algorithms (see e.g., [94, 104]), to allow nested parallelism. In the work-depth model, work is the total number of operations, and depth is the number of operations on the critical path assuming an unlimited number of processors and excluding the cost of nesting. However, we actually also consider an empirical variant of this model where operation counts are replaced with timings extracted from the execution of a serialized parallel algorithm on a commodity processor. These models, elaborated in Sec. 7.6, can provide a basis to compare new parallel SAT solvers in terms of bounds on performance potential.

For our parallel versions of Glucose, our preliminary findings suggest speedup potential of over 380X in some cases, as shown in Table 7.1 below. We explain Table 7.1 by an example: in the first row, author Biere provided inputs for SMT. The best performing input yielded potential speedups of 23.2X and 125.4X for the "Inner" and "Outer" (parallel) algorithms, respectively, and the worst yielded 7.8X and 62.3X (resp.).

The third is that we hope future work will study the challenges involved in bridging the gap between what our algorithms require and what **architectures** provide. For example, as parallelism in our algorithms is fine grained and irregular, including fine-grained nested parallelism, a computing platform designed for such features would be the best fit.

The remainder is organized as follows: Sec. 7.2 defines some terminology. Sec. 7.3 describes state-of-the-art sequential SAT solvers. Sec. 7.4 describes existing approaches to parallelizing solvers, followed by the new approaches we study here and some of the implementation challenges involved. Sec. 7.5 discusses architectural challenges. Sec. 7.6 discusses the model we use in this study. Sec. 7.7 reviews our results as shown in Table 7.1, explaining in detail their underlying assumptions. Sec. 7.8 reviews relevant literature. Finally, Sec. 7.9 concludes.

7.2 Preliminaries

A *Boolean variable* (e.g., x) can take one of two values: $TRUE$ or $FALSE$. A *literal* is a variable x or its negation (written $\neg x$). If x is $TRUE$ its negation $\neg x$ is $FALSE$ and vice versa. A *clause* is a disjunction of literals (e.g., $x \vee \neg y \vee z$); a clause is $TRUE$ (also called *satisfied*) if at least one literal in the clause is $TRUE$ and $FALSE$ (also called *unsatisfiable*) if all literals are $FALSE$. A *unit clause* is a clause with exactly one literal. A *formula in conjunctive normal form (CNF)* (henceforth, *formula*) is a conjunction of clauses (e.g., $(x \vee \neg y) \wedge (y \vee z)$); a clause is $TRUE$ (also called *satisfied*); a formula is $TRUE$ (satisfied) if all clauses in the formula are $TRUE$ and $FALSE$ (unsatisfiable) if at least one clause is $FALSE$. The Boolean satisfiability problem (SAT) is to decide whether a given formula is satisfiable.

7.3 Sequential SAT solver algorithms

The Boolean satisfiability problem (SAT) is NP-complete: there is no known way to solve SAT in polynomial-time, and unless P=NP there will not be one. Therefore, efficient SAT solvers rely on heuristics to provide good performance on practical inputs. The most common approach for SAT solving is the backtracking search algorithm. It traverses the search tree of all partial variable assignments in a depth-first manner until it finds a satisfying assignment or until it concludes that no such assignment exists and the formula is unsatisfiable. Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [44] (see Algorithm 1). It traverses the search trees, depth-first, one variable at a time, assigning it the values either TRUE or FALSE. The worst-case time complexity of all backtracking algorithms is exponential in the number of variables.

Algorithm 1: Davis-Putnam-Logemann-Loveland (DPLL) solver

```
Input: A CNF formula $\phi$
Output: A decision of whether $\phi$ is satisfiable.
1: function DPLL($\phi$)
2:     UNIT_PROPAGATE($\phi$)
3:     if the empty clause is generated then return $FALSE$
4:     else if all variables are assigned then return $TRUE$
5:     else
6:         $Q \leftarrow$ some unassigned variable ▷ Variable is selected based on a heuristic
7:         return DPLL($\phi \wedge Q$) $\vee$ DPLL($\phi \wedge \neg Q$) ▷ Value of $Q$ is selected based on a heuristic
8:     end if
9: end function
```

To improve performance, all SAT solvers are engaged in 3 main performance enhancing steps, aiming at pruning the search space explored.

- **Unit propagation.** After each assignment, the CNF formula is simplified by unit propagation (see Algorithm 2), which is a form of constraint propagation that prunes the search space while moving forward extending a partial solution by one more variables. If unit propagation generates an empty clause a dead end occurs and the current assignment is declared a conflict. DPLL backtracks to the last variable it assigned, flips its value (from $TRUE$ to $FALSE$ or vice versa), and tries again.

- **Variable and value ordering heuristics.** Unit propagation is also instrumental in facilitating look-ahead heuristics for selecting the next variable and its next value. These ordering decisions are known to have an immense impact on the size of the search tree explored, and thus on the efficiency of the algorithm.

- **Conflict-Directed Clause Learning (CDCL).** When a dead end occurs, rather than naively backtracking to a previous assignment, the algorithm analyses the reason for the conflict, and learns a new clause (or a *nogood*) that is added to the CNF formula, ensuring that the same conflict will not occur during the remainder of the search. Such newly-learned clauses are logical implications of the original CNF formula and can therefore augment it without changing its satisfiable models. In addition, instead of backtracking just one level, the algorithm can *backjump* a few layers back, allowing shortcutting parts of the search space that cannot lead to a solution.

Recent success of SAT solvers is attributed to this last step of clause-learning, hence the term CDCL algorithm [16, 45]. Many CDCL solvers, including Glucose, use a further refinement: Variable State Independent Decaying Sum (VSIDS) [110, 125], a heuristic for choosing decision literals that favors variables involved in recent conflicts.

Algorithm 2: Unit propagation

Input: A CNF formula ϕ

Output: An equivalent formula such that no unit clause appears in any non-unit clause.

```
 1: procedure UNIT_PROPAGATE($\phi$)
 2:     $Queue \leftarrow$ all unit clauses in $\phi$
 3:     while $Queue$ is not empty do                        ▷ "Outer" loop (Sec. 7.4.3)
 4:         $T \leftarrow$ next unit clause from $Queue$
 5:         for every clause $\beta$ containing $T$ or $\neg T$ do     ▷ "Inner" loop (Sec. 7.4.3)
 6:             if $\beta$ contains $T$ then
 7:                 Delete $\beta$ from $\phi$          ▷ Known as subsumption elimination [46]
 8:             else
 9:                 Delete $\neg T$ from $\beta$                    ▷ Known as resolution [46]
10:                 if $\beta$ is now a unit clause then add $\beta$ to $Queue$
11:             end if
12:         end for
13:     end while
14: end procedure
```

7.4 *Parallel SAT solver algorithms*

7.4.1 *Existing opportunities for parallelism exploited by Glucose*

Perhaps the simplest approach is to run multiple copies of the same solver on the same input problem with different initial conditions, sometimes call the Portfolio approach. The advantage of this approach is that it is embarrassingly parallel. The disadvantage is that, because there is no coordination among solver instances, work may be duplicated or wasted.

These disadvantages can be mitigated by sharing information between solvers. For solvers that learn new clauses at run time, such as CDCL solvers, a commonly-used approach is to share learned clauses between solver instances. Analogously to how clause learning in serial avoids redundant work by learning from its own failures, clause sharing in parallel avoids redundant work by learning from others' failures. This approach comes with its own disadvantages, however. First, sharing clauses requires communication between solvers, which may incur overhead depending on the computational platform. Second, there will still be redundant work if clauses are not useful or are shared too late to be useful.

The compromise taken by Glucose-Syrup [10], the parallel version of Glucose by the authors of Glucose, is to share clauses lazily using a shared-memory message-passing approach. Each solver places "useful" learned clauses (those that have been involved in conflicts several times) in a queue shared by all threads. Other solvers read new clauses from this queue when it is convenient for them. This is a form of asynchronous communication: each solver sends and receives clauses at its own pace.

7.4.2 Other existing opportunities not exploited by Glucose

Another approach is to partition the search space into disjoint sections and run one copy of the solver on each section. Some ways of partitioning the search space will be more effective than others, with more pre-partition computational work typically leading to a better partition. The cube-and-conquer method [87] uses a look-ahead solver to partition the original problem into many subproblems (called "cubes") and then solves each of the subproblems using a CDCL solver. A look-ahead solver is a DPLL solver where the variable selection heuristic (step 6 in Alg. 1) measures (based on some heuristic) the degree to which each possible variable selection simplifies the formula ϕ after unit propagation, balancing $TRUE$

and $FALSE$ assignments. One simple implementation is to choose the variable x with the greatest value of the product $R(x) \cdot R(\neg x)$, where $R(l)$ equals the number of clauses that are reduced (but not deleted) after performing unit propagation on the formula $\phi \wedge l$; typical implementations favor shorter clauses rather than giving each reduced clause equal weight. Note that this is a much more expensive heuristic than VSIDS.

7.4.3 Fine-grained parallelism extracted from Glucose

Glucose spends most of its time in the unit propagation step (Alg. 2), which is a doubly-nested loop akin to a breadth-first search (BFS) of a bipartite graph whose nodes are variables and clauses, with an edge connecting each clause to the variables it contains, as shown in Figure 7.1. Because the BFS frontier consists of unit clauses, which have a fanout of 1, we can compress the traversal from unit clauses to variables and back to clauses into a single step; this "compressed BFS" will henceforth be simply called "BFS".

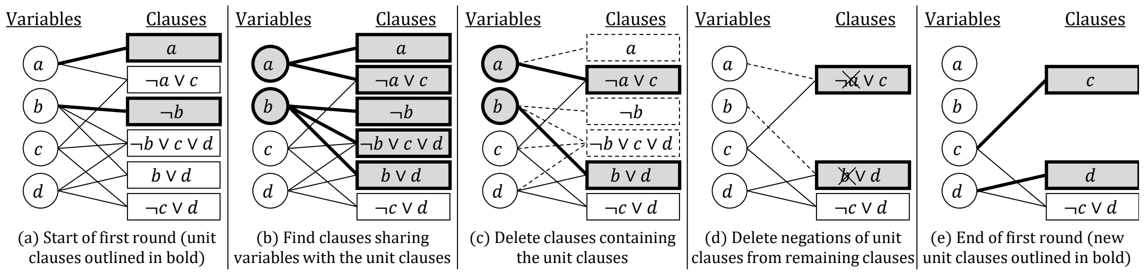

Variables
Clauses
a
b
c
d
a
$\neg a \vee c$
$\neg b$
$\neg b \vee c \vee d$
$b \vee d$
$\neg c \vee d$
(a) Start of first round (unit clauses outlined in bold)
(b) Find clauses sharing variables with the unit clauses
(c) Delete clauses containing the unit clauses
(d) Delete negations of unit clauses from remaining clauses
(e) End of first round (new unit clauses outlined in bold)

Fig. 7.1: Unit propagation viewed as a BFS graph traversal in parallel.

Each of the loops in Alg. 2 can be parallelized, the outer loop by processing all variables in the queue simultaneously and the inner loop by processing all clauses containing the unit clauses being processed simultaneously. This implies a nested parallel algorithm for unit propagation. However, we need to qualify this BFS analogy. Unlike the synchronous progress of BFS from one group of nodes in the parallel outer loop to the next, unit propagation does not mandate such synchrony; we will be on the lookout for exploiting such synchrony relaxation for improving performance.

7.4.4 Further opportunities for parallelism

Beyond this, multiple branches of the DPLL search tree can be explored in parallel, adding a third level of parallelism to the above. This can be applied in the selection of next variable (step 6 in Alg. 1) and/or in value selection (step 7) and be executed in parallel. This is likely to be more challenging to implement efficiently, as some solver state may need to be replicated for each branch, and there will be another level of nesting.

Finally, it may be possible to enhance the existing coarse-grained parallelism within Glucose-Syrup. One way may be to share with other solvers not only learned clauses but also variable assignments.

7.4.5 Implementation issues

One question is how to manage the parallelism of the outer loop. This can be done multiple ways. On one hand, it can be done synchronously: at each iteration, process in parallel all the variables in the queue that were added during the last iteration, in similar fashion to BFS. The advantage of this approach is that it is conceptually simpler to coordinate thread execution, while the disadvantage is that the longest thread must finish before beginning any new threads. On the other hand, an asynchronous approach can be used: spawn a new thread for each variable added to the queue as soon as it is discovered. This allows threads to start as soon as possible. However, the computing platform may not support efficient dynamic spawning of new threads within parallel code.

A related issue is that, as explained later, nesting is difficult to implement with good performance in practice. A possible alternative is to flatten parallelism to a single level. This presents the question of how to build the list of jobs to be executed by nested threads. One option is to statically reserve space for a fixed number of jobs per parent. In this case, a parent can add a job simply by writing to one of its reserved slots. Alternately, space for jobs can be allocated dynamically, which requires a parent to first request space for a job from a global pool, have its request granted and then write to it. The static approach has limited scalability (finite number of children per parent), but it is faster, since its saves the request for space allocation and its granting. Depending on the architecture there may also be a difference in cost even once space is allocated, but our modeling currently suppresses it.

7.4.6 Scaling across nodes

Because the fine-grained parallelism we study here is orthogonal to the coarse-grained parallelism of the portfolio approach in Glucose-Syrup (Alg. 3), it is possible to combine the two as follows (Alg. 4):

- Run one solver per node instead of one per core, with learned clauses shared via message passing instead of shared memory.

- Use a heuristic (to be determined) instead of random selection to choose each node's initial literal.
- Use a fine-grained parallel solver on each node instead of a serial one.

Algorithm 3: Original Glucose-Syrup

```
1: for each core in system, in parallel do
2:     $l \leftarrow$ random literal
3:     SerialCDCL($\phi \wedge l$)
4: end for
```

Algorithm 4: Combined fine-grained & distributed solver

```
1: for each node in cluster, in parallel do
2:     $l \leftarrow$ literal chosen by heuristic (TBD)
3:     ParallelCDCL($\phi \wedge l$)
4: end for
```

Furthermore, it is possible to extend the above by replacing the single-literal selection heuristic (line 2) with a split of the search space based on multiple literals, which we expect to result in a better balance of work across nodes at the expense of more work up front. One approach of this sort is cube-and-conquer, in which fine-grained parallelism can be exploited both in the cube phase (look-ahead solver) and in the conquer phase (CDCL solver on each node).

7.5 Architectural challenges

Here are some challenges posed by SAT algorithms to current platforms:

Granularity: For the new parallel algorithms studied, threads are short (10s-1000s of cycles), but multicores are optimized for longer threads.

Irregularity of threads: Thread length depends on clause length and satisfiability, but GPUs execute threads in lockstep and impose a performance penalty when threads diverge.

Irregularity of memory access: Different clauses are active at different times depending on current variable assignments and the variable being propagated. This means that accesses are scattered throughout memory, and memory access patterns can only be determined at run time.

Communication overhead: The fine-grained parallel algorithms probably share too much data to run efficiently across multiple nodes but may be practical within a node. Meanwhile, the coarse-grained algorithms require less communication, but scalability is limited due to overlap of work between nodes. Here, there is a tradeoff: the more nodes communicate, the less redundant work they do, but the more expensive the communication is, and the sweet spot depends on the specifics of the platform.

These challenges can be addressed from multiple angles. One is to examine what is possible with commodity computing platforms. For example, OpenMP can be used to parallelize one or both loops: (i) parallelize the inner loop only, thereby avoid nesting; (ii) parallelize the outer loop only and serialize the inner loop; and, (iii) parallelize both loops at once. Possible topics to study include: (a) the extent to which each approach may provide enough parallelism to enable some speedups; and (b) when the overhead of nesting may be too high to make this beneficial (see

Sec. 7.6). Our results (Table 7.1) suggest that parallelizing the inner loop only may provide enough parallelism to enable speedups on multi-core processors. However, we suspect that the overhead of nesting may be too high on multi-cores for taking advantages of parallelizing both loops (see Sec. 7.6), in spite of the fact that this exposes more parallelism.

Another angle is to explore possible architectural enhancements. One vehicle for doing so is the Explicit Multi-Threading (XMT) architecture [164], developed by at the University of Maryland as a general-purpose parallel computing platform. The XMT toolchain, consisting of the XMTC compiler and XMTSim cycle-accurate simulator as well as an FPGA prototype, allows exploring the effect of different architectural decisions on program performance. Experimental architectures such as XMT can be used to study what would be possible with hardware designed specifically to address the above challenges, showing the gap between current platforms and what is possible, which may provide insight into how current architectures may be improved.

7.6 Computational model

We need to develop a model for understanding the performance of SAT solvers in practice, but typical approaches are limited here. Asymptotic analysis of the SAT solver algorithms is not very informative, as all SAT solvers have an exponential worst-case run time while practical inputs will take much less time than this. Performance testing on existing platforms does not allow testing hypothetical architecture changes. Simulators run too slowly to allow complete runs of realistically large inputs in a reasonable time. The latter two approaches do not allow understanding the algorithm apart from an architecture.

Here, we extend the abstract **work-depth** (WD) model. In this model, popularized by the PRAM theory of algorithms (see e.g., [94, 104]), work is the total number of basic operations executed by the algorithm, and depth is the number of basic operations on the critical path of execution. The idea is that work translates to the total time taken to execute an algorithm serially, and depth translates to the shortest time to execute the algorithm assuming an unlimited number of processors, no resource (e.g., memory) contention, and no overhead for scheduling threads to processors. For example, the work for a parallel *for loop* is the time it takes to execute the loop serially, and the depth is the length of the longest iteration of the loop. Work divided by depth yields an upper bound on speedup.

We further extend this to consider an **empirical work-depth** model. In this model, we replace the asymptotic counts of basic operations as used in abstract WD with empirical timings on a state-of-the-art serial processor. Specifically, we execute the parallel algorithm sequentially on a single processor and record the time taken by (i) each serial section and (ii) each (serialized) iteration of each parallel section. For a serial section, work and depth both equal the measured time; for a parallel section, work is the sum of the times of the iterations, and depth is the time of the longest iteration. This allows us to account for the relative costs of different types of operations and the dependence of potential speedup on the input problem. The results we obtained in Sec. 7.7 using this model suggest promise, but it needs to be refined to account for implementation challenges.

A particularly interesting challenge is how to handle **nested parallelism**. Prior nesting-driven work (e.g., [159]) did not seek to optimize overhead for short and repeated spawning. A fundamental question in this case is how to efficiently synchronize a parent thread with its children to assign them work. Due to the limited success of software-only approaches to nesting, we believe it may be worth examining the potential benefits of adding low-overhead hardware primitives to support nesting, such as those in the Vishkin and Wen patent [171]. Factors that need to be modeled in order to test this assumption include 1. the overheads of various hardware approaches to nesting and 2. the effect of synchronization overhead on algorithmic performance.

7.7 *Empirical results*

Here, we examine the potential for speedups on an assortment of non-random SAT inputs, from a variety of problem domains, used in the SAT Competition 2018 [86]. There are 300 inputs in total, with 18 groups of authors each submitting 9 to 20 inputs. Each row of Table 7.1 corresponds to a different group of authors. Authors of input sets were supposed to submit descriptions of their sets to be included in the competition proceedings. For those who did, the problem domain of their inputs is also listed in Table 7.1; otherwise, the domain is listed as "(no description provided)".

We explored potential speedups for two new parallel versions of Glucose. In the "Inner" approach, we parallelize the inner loop of the unit propagation while leaving the outer loop serial. In the "Outer" approach, we parallelize both the inner and outer loop of the unit propagation. Our current study focuses on the unit propagation step of Glucose, which accounts for approximately 95% of the total run time of the solver; the majority of the remaining 5% is spent in the conflict analysis step, which would be a natural topic for future work.

For each input, we executed serial Glucose on an Intel Core i5-2500K CPU for up to 900 seconds. During each run, we measured the time t_{ijk} taken for every iteration of the inner loop executed during that run, where t_{ijk} is the time taken by the kth iteration of the inner loop within the jth iteration of the ith batch of the outer loop. A "batch" is a set of iterations of the outer loop corresponding to the set of variables enqueued during the previous batch, analogous to a BFS level. From these measurements, we computed the following according to the empirical work-depth model described in Sec. 7.6:

- "Empirical work": $W = \sum_i \sum_j \sum_k t_{ijk}$, the sum of the times for all iterations of the inner loop across all iterations of the outer loop

- "Empirical depth" ("Inner"): $D_I = \sum_i \sum_j (\max_k t_{ijk})$, the sum of the times for the longest iteration of the inner loop per iteration of the outer loop

- "Empirical depth" ("Outer"): $D_O = \sum_i (\max_j \max_k t_{ijk})$, the sum of the times for the longest iteration of the inner loop per batch of iterations of the outer loop

Using these measurements as inputs according to the empirical work-depth model described in Sec. 7.6, we computed the potential speedups on each input. The "Inner" speedup equals W/D_I, and the "Outer" speedup equals W/D_O. Table 7.1 shows the speedups for best and worst input for each group of inputs ("best" and "worst" are chosen based on "Outer" speedup).

Based on these results, we make the following observations:

- In many cases, it is necessary to parallelize the outer loop to gain strong speedups, especially for the worst inputs in each group.
- Potential parallelism varies greatly across problem groups, meaning that problem domain must be considered when evaluating algorithms.
- Potential parallelism can vary substantially within problem groups as well, making it difficult to characterize the difficulty of problem domains.

7.8 Literature review

7.8.1 Sequential SAT solvers

- MiniSat [60] is an open-source CDCL SAT solver that is often built upon by developers to test new heuristics.
- Glucose [9] (Sec. 7.3) builds on MiniSat by adding heuristics for periodically removing learned clauses that are unlikely to be useful in the future while keeping important learned clauses.
- An alternative to MiniSat and its derived solvers is Lingeling, which first appeared in SAT Race 2010 [17] and as recently as SAT Competition 2017

[19]. Lingeling augments the standard CDCL algorithm with inprocessing [95], which is the periodic application of satisfiability-preserving Boolean transformations.

7.8.2 Multi-core parallel SAT solvers

- Glucose-Syrup [10] (parallel Glucose) (Sec. 7.4)
- Plingeling [17] (parallel Lingeling), like Glucose-Syrup, implements a portfolio approach, but Plingeling uses a simpler clause sharing approach that only shares unit clauses.
- Treengeling [18] employs cube-and-conquer [87] (Sec. 7.4.2), which uses search-space splitting to generate sub-problems that can be solved in parallel.

None of the above papers discuss speedup versus the best serial algorithm.

7.8.3 GPU SAT solvers

Limited progress has been made on using GPUs to accelerate SAT solving.

- Some work [66, 118] considered solving 3-SAT (a restriction of SAT to clauses of three or fewer literals) on GPUs, with [66] achieving a speedup on random inputs of up to 6.7X versus the same algorithm on the CPU alone.
- The master's thesis [38] adapted MiniSat to use GPUs but was unable to achieve any speedup versus serial MiniSat or Glucose.

- CUD@SAT [40], exploits two sources of parallelism. The first is to perform unit propagation in parallel on the GPU; unlike the algorithm we studied, CUD@SAT deviates from the optimized serial unit propagation algorithm by examining all clauses each iteration rather than only those that were touched by the last iteration. The second is to parallelize the tail of the search by switching from CDCL to a GPU-only DPLL algorithm when only a few variables remain unassigned. CUD@SAT achieved up to a 9.4X speedup using a GPU versus the same algorithm on the CPU alone. Dal Palù et al. considered comparison to state-of-the-art serial SAT solvers to be beyond the scope of [40].

7.8.4 Nested parallelism in OpenMP

OpenMP allows programmers to express nested parallelism in their code, but efficient support for such is still a work in progress. Furthermore, the scheduling of nested parallel tasks to processors is challenging, as recent work has attempted to address.

- Dimakopoulos et al. [48] found that several OpenMP implementations have overheads of over an order of magnitude higher for nested parallelism than for single-level parallelism.

- Maroñas et al. [116] developed "worksharing tasks", which relax the fork-join execution model of the standard worksharing construct (parallel for loop) of OpenMP to allow overlapping the execution of loops whose dependencies do not overlap. They achieved a speedup of 1.5X-9X versus standard OpenMP, with the High Performance Computing Conjugate Gradient (HPCCG) benchmark yielding the best results.

- Sun et al. [153] proposed a hierarchical scheduling (HS) algorithm that defers the execution of nested parallel tasks until there are a reasonable number of idle processors available in order to improve the worst-case response time of OpenMP code on real-time systems. Although they did not implement their scheduling algorithm, they theoretically analyzed it under a directed acyclic graph (DAG) task model and derived formulas for upper bounds on response time (finishing time of the sink vertex of the DAG minus the starting time of the source vertex).

7.8.5 Other software approaches for nested parallelism

Some other software approaches proposed for explicit handling of nesting include: [20] and later work on the NESL language, and [22] and later work on the Cilk project, and quite a few "lazy" schedulers of coarse-grained nested parallelism including [74, 75, 123] and the fine-grained XMT work in [159]; for brevity, we refer the reader to a full page section entitled "Schedulers without Parallel Loop Support" in the latter paper. Our comment that all implementations known to us incur high overheads applies here as well.

7.8.6 Odds and ends

Handling limited nesting on GPUs is considered in [63]. The survey [77] reviews several projects seeking special-purpose FPGAs for instances of SAT solving. On the parallel algorithmic theory side, the randomized $O(\log^* n)$ dictionary data structure in [72] applies to nesting; however, constant factors hidden by the big oh notation may be an issue.

In our opinion, the main missing element in prior work has been a satisfactory solution for the extremely fine level of nesting granularity that parallelization of SAT solvers mandates. A secondary though significant issue is that, unlike some other works, we aspire to advancing SAT solving as a potential killer application for a more general form of scalable parallelism, ideally for a general-purpose platform.

A notable precedent for a computer architecture model abstracting not-yet implemented hardware capabilities in order to study their potential has been the extensive study of instruction level parallelism in works such as [107, 167].

7.9 Conclusion

We have begun to study the potential fine-grained parallelism in sequential SAT solvers in hopes that it will stimulate further research into parallel SAT solver algorithms, computing stacks that can execute them efficiently, and models for understanding how the former may perform on the latter. Topics for future study include understanding which among the various approaches noted here are the most promising as well as developing new approaches beyond these.

Input set	Potential speedups			
	Best input		Worst input	
Problem domain (Author)	Inner	Outer	Inner	Outer
SMT (Biere)	23.2	125.4	7.8	62.3
N.D.P. (Chen)	36.0	126.8	35.2	123.7
Puzzle (Chowdhury)	152.0	329.4	25.0	68.6
Graph k-colorability (Devriendt)	153.3	382.4	39.0	95.8
Tree decomposition/treewidth (Ehlers)	11.1	60.3	12.2	40.0
Cellular automata (Harder)	16.0	58.9	8.2	24.2
Graph coloring (Heule)	11.2	158.6	7.6	93.3
Blockchain (Bitcoin mining) (Heusser)	6.0	144.8	6.4	115.3
Combinatorics (Jingchao)	60.0	135.3	26.4	41.4
Scheduling (Konan)	29.4	103.9	3.1	10.9
Floating-point verification (Liang)	11.5	119.2	5.0	49.1
Software verification (Manthey)	10.4	112.9	1.7	14.3
N.D.P. (Mayer-Eichberger)	18.3	205.2	7.6	96.7
N.D.P. (Ofer)	2.1	54.0	2.6	29.5
N.D.P. (Porkhunov)	144.9	319.4	7.4	59.6
Cryptanalysis (Scheel)	22.1	66.1	8.8	27.3
Polynomial multiplication (Xiao)	8.6	35.2	2.6	9.3
Factorization (Zha)	57.6	290.6	16.7	183.2

(N.D.P. = no description provided by author)

Tab. 7.1: Potential speedups for the main nested loop of Glucose on a selection of inputs from the SAT Competition 2018 [86]. "Inner" and "outer" refer to parallelization of the two labeled loops in Alg. 2. By analogy to breadth-first search (BFS), "inner" is akin to exploring the edges from a node in parallel, and "outer" additionally explores multiple nodes in parallel. This BFS analogy is made precise in Sec. 7.4.3. We compute potential speedups by running serial Glucose, measuring its total run time. For the Inner result we also figure out the portion thereof corresponding to the critical path for each parallel algorithm and divide total runtime by the critical path (for elaboration and explanation of how the Outer result is derived, see Sec. 7.6).

Chapter 8: Characterization of Applications on XMT and GPUs

The general principles that led to the success of the work in the preceding chapters are (1) the existence of an efficient parallel algorithm (whose work is a close to the best serial algorithm as possible and whose depth is much less than its work) and (2) the possibility to hide the latency of memory operations, either through concurrent memory accesses or via prefetching. On XMT, the latter is made possible due to advantage both on the control path (independent progress of concurrent threads and immediate reuse of just-freed hardware) and in the memory system.

The contribution of this work can best be explained by considering three types of algorithms. The first type, with a significant amount of regular parallelism (see Sec. 8.1), already allows good speedups on both XMT and GPUs. Second, on the opposite end of the spectrum are irregular algorithms where depending on the specific input for a given run of a program parallelism can either be limited or there is no parallelism at all. These cases may provide little to no speedup on XMT and a slowdown on GPUs. Here, XMT still tends to outperform GPUs because XMT is engineered to allow seamless fallback on serial, efficient transitions between serial and parallel, and run-time adaptation to exploit as much parallelism as the input allows, as explained below. (A fundamental difference is that GPUs are engineered as accelerators, while XMT is engineered as a blueprint for a manycore CPU. The Intel integrated GPU is the closest so far to the XMT integrated vision. We already noted above the control path and memory system advantages of XMT.) The third type consists of algorithms with significant parallelism that is nevertheless difficult to exploit on traditional parallel architectures such as GPUs. The primary contribution of this dissertation is to establish that a number of important problems are of the third type rather than the second and that XMT broadens the set of problems for which parallelism can be exploited. A secondary, but important contribution is to make inroads into the second type, as demonstrated in the work on SAT solvers.

While a first generation of the parallel programming of such algorithms and programs require the type of skill demonstrated in the current dissertation, follow-up efforts can automate them, as demonstrated in [71]. Examples include automation of the parallel list ranking implementation developed and used in several places in this dissertation as well as the incorporation of the biconnectivity work of Chapter 2 into a class programming project.

8.1 Effects of data dependence

One major difference between XMT and GPUs is how well they handle data dependence in control flow (conditional execution) and in memory access (indirect memory addressing). These often arise when programs use pointer- or index-based data structures such as trees, graphs, linked lists, and sparse matrices. Due to the general-purpose design of XMT, XMT typically outperforms GPUs for programs with a non-trivial amount of data dependence ("irregular" parallelism), whereas GPUs tend to outperform XMT on programs without data dependence ("regular" parallelism) due to GPUs being optimized for such scenarios. This shows up in a number of ways:

Locality of reference versus predictability of reference GPUs rely heavily on locality of reference for efficient memory access. That is, memory elements that are accessed by concurrent threads should be adjacent in memory. For applications with a high degree of locality, GPUs can outperform XMT due to GPUs having a memory architecture tailored to such applications.

XMT also benefits from locality, but it is able to cope with a lack of locality as long as memory references are predictable and thus can be prefetched. Examples of predictable references include sequences of consecutive non-dependent loads and loads of array elements indexed by a loop counter. On XMT, prefetching is done in software to keep the processor cores simple and to allow the compiler or programmer to identify opportunities for prefetching that may be difficult for hardware to spot.

Number of threads GPUs are able to hide the latency of memory access as long as there are many more threads than cores. This is also true on XMT, but XMT tends to outperform GPUs when available parallelism is more modest.

Length of parallel sections GPUs are able to hide the latency of thread spawning as long as there is a large amount of work to do within a parallel section of code, but performance is greatly reduced when frequent serial-to-parallel transitions are required, say in response to the reorganization of a sparse data structure or the compaction of an array after removing elements. In contrast, the low serial-to-parallel transition overhead of XMT allows good performance to be obtained even for parallel sections consisting of only a few lines of C code.

Consistency of control flow Since GPUs execute code in lockstep within groups of threads (warps), control flow must be consistent across all threads in a warp for maximum performance. Conditional statements and loops whose behavior differs across threads incur a divergence penalty and also interfere with the ability of the GPU to merge memory accesses for locality. In contrast, on XMT, each thread executes at its own pace independent of other threads, and its control flow does not interfere with the execution of other threads.

Load balancing of threads The previous point implies in particular that, on GPUs, all threads in a parallel section should be the same length for optimum performance. This is possible but not as mandatory (for performance) on XMT, which only requires that no single thread should dominate the critical path of execution.

8.2 Tension between work efficiency and data dependence

The above considerations are reflected in the different approaches to algorithm design taken on XMT versus GPUs. On XMT, the choice of algorithm is primarily driven by its efficiency in the work-depth model, while regularity of parallelism is only a secondary concern. In contrast, algorithms for GPUs tend to favor regularity, both on the control side and for memory access, even at the cost of work efficiency.

A simple example is matrix-vector multiplication. If the matrix is stored in dense form, with all elements explicitly represented in memory, then any element of the matrix and its matching element in the vector can be readily located in memory without indirect addressing, and all threads follow the same control flow path. This is an ideal situation for GPUs, so the product can be computed very efficiently on a GPU. This will also work well on XMT, but since the peak floating-point performance of XMT is less than that of a comparable GPU, the GPU will outperform XMT in this case.

In cases where a matrix contains many elements that are zero, explicitly storing the zeros wastes memory, bandwidth, and work. While beyond the scope of this dissertation, this type of work done by the GPU led prior XMT work [101] to demonstrate that the energy requirement of XMT is generally on par with GPUs. In such cases, it is common to store the matrix in sparse form, with the non-zero elements and their column indices stored for each row. The downsides are that matching each matrix element to its corresponding vector element now requires indirect addressing into the vector, and the work done by each thread depends on the number of non-zero elements in its row. These factors adversely affect the GPU more than they do XMT, and in this case, XMT has been shown [29] to perform over 2X as fast as a comparable GPU.

8.3 Examples

Table 8.1 lists some applications in a number of domains and provides an educated guess, based on the foregoing criteria, about whether each is amenable to speedup on XMT and/or GPUs. This is not an exhaustive list, and it is not intended to prejudice the development of new parallel algorithms or architectural advancements that may enable efficient execution of existing parallel algorithms, but it provides some insight into how the results of the case studies in this dissertation may translate to other applications. As a rule, we expect applications with regular algorithms to perform well on both XMT and GPUs and those with irregular algorithms to perform better on XMT. The exceptions are "inherently sequential" algorithms and algorithms for P-complete problems, which may provide little to no parallelism in the worst case. We expect such algorithms to fall behind a serial CPU on GPUs, and they may or may not provide any speedup on XMT, though XMT will not fall behind serial due to incorporating a strong serial processor. Interestingly, Chapter 7 shows that SAT solvers are a potential exception to this rule as they show promise despite being NP- (and thus P-) complete.

XMT & GPU	Better on XMT
AES encryption	Boolean satisfiability (SAT)
Chirp Z-transform	Boosted decision trees
Convolution	Burrows-Wheeler compression
Dynamic Programming	De novo transcriptome assembly
Fast Fourier transform (FFT)	Enhanced dynamic programming
Gaussian elimination	Graph (1,2,3)-connectivity
Integer factorization	Lempel-Ziv (LZ77) compression
LU factorization	List ranking
Matrix inverse	Long short-term memory (LSTM) neural network
Matrix multiplication (dense)	Matrix multiplication (sparse)
Multilayer perceptrons	Maximum flow
Ordinary differential equations	Maximum SAT (MaxSAT)
Partial differential equations (rectangular mesh)	Molecular dynamics simulation
QR factorization	Partial differential equations (general mesh)
RSA encryption	Phylogeny inference
	Planarity testing
Some potential on XMT ($W \approx D$)	Random forests
Circuit value problem	Regular expressions
Context-free grammars	Sorting
Linear programming	Spanning tree
Newton's method	String matching
Neither	
Lempel-Ziv (LZ78) compression	
SHA-256 hash	

Tab. 8.1: List of applications that we speculate can potentially provide speedups on the listed platforms.

Chapter 9: Conclusion

The speedup results in this thesis show that it is possible to obtain strong speedups for PRAM algorithms for a variety of applications on XMT, a buildable computer architecture. In addition to potentially benefiting users in those application domains, the results provide some evidence to address criticism that the PRAM model is impractical and provide further validation of the approach taken by the XMT architecture.

This work also provides some support for the ease of programming of XMT. For applications where there was an existing PRAM algorithm, that algorithm provided good speedups without major modifications. For applications where a new algorithm needed to be developed, it sufficed to derive a PRAM algorithm from the best serial algorithm.

The results demonstrate the need for co-design of algorithms and architectures. To achieve strong speedup on these applications, it was necessary to consider both algorithm and architecture as variables, in contrast to prior research that considered the architecture as fixed.

This work also shows that XMT has room for further improvement. In particular, SAT solvers could benefit from efficient hardware support for very fine-grained nesting.

In some cases, the fine-grained parallelism studied here is orthogonal to existing coarse-grained parallelism. This means that even better performance could be obtained using a platform that supports efficiently exploiting both levels of parallelism simultaneously. XMT augmented with nesting support in hardware might be one such platform.

Published work incorporated into this thesis

The material in the following chapters has appeared in peer-reviewed publications: Chapter 2 [52], Chapter 3 [53], Chapter 4 [54, 56] (theory results only), Chapter 5 [57] and Chapter 6 [58]. The work in Chapter 7 is currently under review for publication as [59].

Bibliography

[1] QuickLZ: Fast compression library for C, C# and Java. `http://www.quicklz.com/index.php`.

[2] The ninth DIMACS implementation challenge: The shortest path problem. `http://www.dis.uniroma1.it/~challenge9/`, 2005.

[3] NVIDIA's next generation CUDA compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[4] Stanford network analysis platform. `http://snap.stanford.edu/index.html`, 2009.

[5] NSF/IEEE-TCPP curriculum initiative on parallel and distributed computing - core topics for undergraduates. `http://www.cs.gsu.edu/~tcpp/curriculum/index.php`, December 2010.

[6] Jürgen Abel. Improvements to the Burrows-Wheeler compression algorithm: After BWT stages. 2003. URL `http://www.juergen-abel.info/files/preprints/preprint_after_bwt_stages.pdf`.

[7] Jürgen Abel. Post BWT stages of the Burrows-Wheeler compression algorithm. *Software: Practice and Experience*, 40(9):751–777, 2010. ISSN 1097-024X. doi: 10.1002/spe.982. URL `http://dx.doi.org/10.1002/spe.982`.

[8] Peter M. G. Apers. Recursive samplesort. *BIT Numerical Mathematics*, 18: 125–132, 1978. ISSN 0006-3835.

[9] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009. URL `https://www.ijcai.org/Proceedings/09/Papers/074.pdf`.

[10] Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 197–205, 2014. ISBN 9783319092836. doi: 10.1007/978-3-319-09284-3_15. URL `http://link.springer.com/10.1007/978-3-319-09284-3{_}15`.

[11] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 547–556, June 2005. doi: 10.1109/ICPP.2005.55.

[12] Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009. doi: 10.1109/ISPASS.2009.4919648.

[13] A.O. Balkan, M.N. Horak, Gang Qu, and U. Vishkin. Layout-accurate design and implementation of a high-throughput interconnection network for single-chip parallel processing. In *Hot Interconnects 15*, pages 21–28, August 2007. doi: 10.1109/HOTI.2007.11.

[14] A.O. Balkan, Gang Qu, and U. Vishkin. An area-efficient high-throughput hybrid interconnection network for single-chip parallel processing. In *Proc. IEEE/ACM Design Automation Conf.*, pages 435–440, June 2008.

[15] Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing. In *Proc. FOCS*, pages 436–441, 1989. doi: 10.1109/SFCS.1989.63515.

[16] Roberto J. Bayardo and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI) - Volume 1*, pages 298–304, 1996. URL `https://www.aaai.org/Papers/AAAI/1996/AAAI96-045.pdf`.

[17] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Johannes Kepler University, Linz, Austria, 2010. URL `http://fmv.jku.at/papers/Biere-FMV-TR-10-1.pdf`.

[18] Armin Biere. Lingeling and friends entering the SAT challenge 2012. In *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, pages 33–34, 2012. URL `http://fmv.jku.at/papers/Biere-SAT-Challenge-2012.pdf`.

[19] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[20] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. ISSN 00189340. doi: 10.1109/12.42122. URL `http://ieeexplore.ieee.org/document/42122/`.

[21] Guy E. Blelloch, Bruce M. Maggs, and Gary L. Miller. The hidden cost of low bandwidth communication. In Uzi Vishkin, editor, *Developing a computer*

science agenda for high-performance computing, pages 22–25. ACM Press, New York, NY, USA, 1994. ISBN 0-89791-678-6. doi: http://doi.acm.org/10.1145/197912.197923.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216, Aug. 1995. ISSN 03621340. doi: 10.1145/209937.209958. URL `http://portal.acm.org/citation.cfm?doid=209937.209958`.

[23] R. Borkar, M. Bohr, and S. Jourdan. Advancing Moore's Law on 2014, August 2014. URL `http://www.intel.com/content/dam/www/public/us/en/documents/presentation/advancing-moores-law-in-2014-presentation.pdf`.

[24] Thomas Brunschwiler, B Michel, Hugo Rothuizen, U Kloter, B Wunderle, H Oppermann, and H Reichl. Forced convective interlayer cooling in vertically integrated packages. In *IEEE 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITHERM)*, pages 1114–1125, 2008.

[25] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994.

[26] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2), 2008. ISSN 10949224. doi: 10.1145/1455518.1455522.

[27] George Caragea and Uzi Vishkin. Better speedups for parallel max-flow. In *Proc. SPAA*, 2011.

[28] George C. Caragea and Uzi Vishkin. Brief Announcement: Better Speedups for Parallel Max-flow. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 131–134, 2011. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989511.

[29] George C. Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin. General-purpose vs. GPU: Comparison of many-cores on irregular workloads. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX, June 2010.

[30] Shuo Chen and Xiaoming Li. A hybrid GPU/CPU FFT library for large FFT problems. In *32nd International Performance Computing and Communications Conference (IPCCC)*, pages 1–10. IEEE, Dec 2013. ISBN 978-1-4799-3214-6. doi: 10.1109/PCCC.2013.6742796.

[31] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16*, pages 785–794, 2016. doi: 10.1145/2939672.2939785. URL `http://dl.acm.org/citation.cfm?id=2939672.2939785`.

[32] Markus Chimani, Carsten Gutwenger, Michael Jünger, Karsten Klein, Petra Mutzel, and Michael Schulz. The open graph drawing framework. Technische Universität Dortmund and University of Cologne, `http://ogdf.net/`, 2010.

[33] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, July 1986. ISSN 0019-9958. doi: 10.1016/S0019-9958(86)80023-7.

[34] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81(3):334 – 352, 1989. ISSN 0890-5401. doi: 10.1016/0890-5401(89)90036-9.

[35] Lasse Collin. XZ utils. `http://tukaani.org/xz/`.

[36] Guojing Cong and D.A. Bader. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium.*, page 45b, April 2005. doi: 10.1109/IPDPS.2005.100.

[37] Intel Corporation. The compute architecture of intel processor graphics gen9, 2015. URL `https://software.intel.com/sites/default/files/managed/c5/9a/The-Compute-Architecture-of-Intel-Processor-Graphics-Gen9-v1d0.pdf`.

[38] Carlos Filipe Costa. Parallelization of SAT algorithms on GPU. Master's thesis, Instituto Superior Técnico, University of Lisbon, 2014. URL `https://fenix.tecnico.ulisboa.pt/downloadFile/395146459368/thesis.pdf`.

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

[40] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, May 2015. ISSN 0952-813X. doi: 10.1080/0952813X.2014.954274. URL `http://www.tandfonline.com/doi/full/10.1080/0952813X.2014.954274`.

[41] William James Dally. Unified streaming multiprocessor memory, jun 2015. Patent No. US 9,069,664 B2, Filed Sep. 22nd., 2011, Issued June 30th., 2015.

[42] W.J. Dally. The end of denial architecture. Technical report, The International Symposium on Asynchronous Circuits and Systems (ASYNC), 2009. http://asyncsymposium.org/async2009/slides/dally-async2009.pdf.

[43] Shlomit Dascal and Uzi Vishkin. Experiments with list ranking for explicit multi-threaded (XMT) instruction parallelism. *J. Exp. Algorithmics*, 5, December 2000. ISSN 1084-6654. doi: http://doi.acm.org/10.1145/351827.384252.

[44] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, Jul. 1962. ISSN 00010782. doi: 10.1145/368273.368557. URL `http://portal.acm.org/citation.cfm?doid=368273.368557`.

[45] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, Jan. 1990. doi: 10.1016/0004-3702(90)90046-3. URL `https://doi.org/10.1016/0004-3702(90)90046-3`.

[46] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 9780080502953.

[47] J. Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 585–585, May 2013. doi: 10.1109/IPDPS.2013.123.

[48] Vassilios V. Dimakopoulos, Panagiotis E. Hadjidoukas, and Giorgos Ch Philos. A microbenchmark study of OpenMP overheads under nested parallelism. In *International Workshop on OpenMP (IWOMP)*, 2008. doi: 10.1007/978-3-540-79561-2_1. URL `http://link.springer.com/10.1007/978-3-540-79561-2{_}1`.

[49] Nicolas Dupuis, Benjamin G Lee, Jonathan E Proesel, Alexander Rylyakov, Renato Rimolo-Donadio, Christian W Baks, Abhijeet Ardey, Clint L Schow, Anand Ramaswamy, Jonathan E Roth, Robert S Guzzon, Brian Koch, Daniel K Sparacin, and Greg A Fish. 30-Gb/s optical link combining heterogeneously integrated III-V/Si photonics with 32-nm CMOS circuits. *Journal of Lightwave Technology*, 33(3):657–662, 2015. ISSN 0733-8724. doi: 10.1109/JLT.2014.2364551.

[50] Denise Marie Eckstein. *Parallel graph processing using depth-first search and breadth-first search*. PhD thesis, University of Iowa, 1977. AAI7728449.

[51] J. A. Edwards and U. Vishkin. Parallel algorithms for Burrows-Wheeler compression and decompression. Technical report, University of Maryland, College Park, MD, November 12, 2012. `http://hdl.handle.net/1903/13299`.

[52] James A. Edwards and Uzi 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 '12*, pages 103–114, 2012. doi: 10.1145/2141702.2141714.

[53] James A. Edwards and Uzi Vishkin. Brief announcement: Speedups for parallel graph triconnectivity. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architecture*, pages 190–192, 2012. ISBN 9781450312134. doi: 10.1145/2312005.2312042.

[54] James A. Edwards and Uzi Vishkin. Brief announcement: Truly parallel Burrows-Wheeler compression and decompression. In *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures - SPAA '13*, page 93, 2013. ISBN 9781450315722. doi: 10.1145/2486159.2486164.

[55] James A. Edwards and Uzi Vishkin. Empirical speedup study of truly parallel data compression. Technical report, University of Maryland, 2013. URL `http://hdl.handle.net/1903/13890`.

[56] James A. Edwards and Uzi Vishkin. Parallel algorithms for Burrows–Wheeler compression and decompression. *Theoretical Computer Science*, 525:10–22, March 2014. ISSN 03043975. doi: 10.1016/j.tcs.2013.10.009. URL `http://www.sciencedirect.com/science/article/pii/S0304397513007615`.

[57] James A. Edwards and Uzi Vishkin. FFT on XMT: Case study of a bandwidth-intensive regular algorithm on a highly-parallel many core. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 561–569. IEEE, May 2016. ISBN 978-1-5090-3682-0. doi: 10.1109/IPDPSW.2016.157. URL `http://ieeexplore.ieee.org/document/7529915/`.

[58] James A. Edwards and Uzi Vishkin. Linking parallel algorithmic thinking to many-core memory systems and speedups for boosted decision trees. In *Proceedings of the International Symposium on Memory Systems - MEMSYS '18*, pages 161–168, 2018. ISBN 9781450364751. doi: 10.1145/3240302.3240321. URL `http://dl.acm.org/citation.cfm?doid=3240302.3240321`.

[59] James A. Edwards and Uzi Vishkin. Study of fine-grained nested parallelism in CDCL SAT solvers, 2020. submitted.

[60] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3. doi: 10.1007/978-3-540-24605-3_37.

[61] Axel Eirola. Lossless data compression on GPGPU architectures. *ArXiv e-prints*, 2011.

[62] Venmugil Elango, Naser Sedaghati, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, Radu Teodorescu, and P. Sadayappan. On using the Roofline model with lower bounds on data movement. *ACM Transactions on Architecture and Code Optimization*, 11(4):1–23, 2015. doi: 10.1145/2693656.

[63] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. Parallel programming in Futhark, 2018. URL `https://futhark-book.readthedocs.io/en/latest/`.

[64] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840301.

[65] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2006. ISBN 3540372067. doi: 10.1007/11814948_25.

[66] Hironori Fujii and Noriyuki Fujimoto. GPU acceleration of BCP procedure for SAT algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2012. URL `http://worldcomp-proceedings.com/proc/p2012/PDP6166.pdf`.

[67] Karl Fürlinger and Michael Gerndt. Analyzing overheads and scalability characteristics of OpenMP applications. In *High Performance Computing for Computational Science - VECPAR*, 2006.

[68] Donald Fussell, Vijaya Ramachandran, and Ramakrishna Thurimella. Finding triconnected components by local replacement. *SIAM J. Computing*, 22(3): 587–616, 1993. doi: 10.1137/0222040. URL `http://link.aip.org/link/?SMJ/22/587/1`.

[69] Jean-Loup Gailly and Mark Adler. zlib compression library. `http://www.dspace.cam.ac.uk/handle/1810/3486`.

[70] Fady Ghanim, Uzi Vishkin, and Rajeev Barua. Easy PRAM-based high-performance parallel programming with ICE. *IEEE Transactions on Parallel and Distributed Systems*, 29:377–390, Feb 2018. doi: 10.1109/TPDS.2017.2754376.

[71] Fady A . Ghanim. *Easy PRAM-Based High-Performance Parallel Programming*. Phd thesis, University of Maryland, 2017. URL `http://hdl.handle.net/1903/19069`.

[72] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings 32nd Annual Symposium*

of Foundations of Computer Science, pages 698–710. IEEE Comput. Soc. Press, 1991. ISBN 0-8186-2445-0. doi: 10.1109/SFCS.1991.185438. URL `http://ieeexplore.ieee.org/document/185438/`.

[73] J. Gilchrist and A. Cuhadar. Parallel lossless data compression based on the Burrows-Wheeler transform. In *Proc. Advanced Information Networking and Applications*, pages 877 –884, May 2007. doi: 10.1109/AINA.2007.109.

[74] Seth Copen Goldstein. *Lazy Threads: Compiler and Runtime Stuctures for Fine-Grained Programming.* PhD thesis, University of California–Berkeley, Berkeley, CA, 1997. URL `https://www.cs.cmu.edu/{~}seth/papers/goldstein-phd97.pdf`.

[75] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996. ISSN 07437315. doi: 10.1006/jpdc.1996.0104. URL `https://linkinghub.elsevier.com/retrieve/pii/S0743731596901045`.

[76] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, volume 3, chapter 2, pages 89–134. Elsevier, 2008. ISBN 9780444522115. doi: 10.1016/S1574-6526(07)03002-7. URL `https://linkinghub.elsevier.com/retrieve/pii/S1574652607030027`.

[77] Weiwei Gong and Xu Zhou. A survey of SAT solver. In *AIP Conference Proceedings*, volume 1836, 2017. ISBN 9780735415065. doi: 10.1063/1.4981999. URL `http://aip.scitation.org/doi/abs/10.1063/1.4981999`.

[78] Google, Inc. Snappy, a fast compressor/decompressor. `http://code.google.com/p/snappy/`, Released March 2011.

[79] NK Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, November 2008.

[80] Pei Gu and Uzi Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocessor. *Journal of Embedded Computing*, 2(2):181–190, 2006.

[81] Aarti Gupta, Malay K. Ganai, and Chao Wang. SAT-based verification methods and applications in hardware verification. In *International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*, pages 108–143, 2006. ISBN 3540343040. doi: 10.1007/11757283_5. URL `http://link.springer.com/10.1007/11757283{_}5`.

[82] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In Joe Marks, editor, *Graph Drawing*, volume 1984 of *Lecture Notes in Computer Science*, pages 77–90. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-41554-1. doi: 10.1007/3-540-44541-2_8. URL `http://dx.doi.org/10.1007/3-540-44541-2_8`.

[83] Y. Han and X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM Journal on Computing*, 31(6):1852–1878, 2002. doi: 10.1137/S0097539799352449.

[84] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, fifth edition, 2011.

[85] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055, 9780128119051.

[86] Marijn Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2018 – Solver and Benchmark Descriptions*, volume B-2018-1, 2018. Department of Computer Science, University of Helsinki. URL `http://hdl.handle.net/10138/237063`.

[87] Marijn J.H. Heule, Oliver Kullmann, and Victor W. Marek. Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 4864–4868, Aug. 2017. doi: 10.24963/ijcai.2017/683. URL `https://www.ijcai.org/proceedings/2017/683`.

[88] Ariya Hidayat. Fastlz - lightning-fast compression library. `http://fastlz.org/`.

[89] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, 1981. doi: 10.1145/800076.802486.

[90] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. ISSN 0001-0782. doi: 10.1145/362248.362272.

[91] John E. Hopcroft and Robert E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2(3):135–158, 1973. doi: 10.1137/0202012. URL `http://link.aip.org/link/?SMJ/2/135/1`.

[92] Ying Hu, Oleg Kremnyov, and Ivan Kuzmin. Faster gradient-boosting decision trees. *The Parallel Universe*, 33:55–62, 2018. URL `https://techdecoded.intel.io/resources/faster-gradient-boosting-decision-trees/`.

[93] Kaggle Inc. Higgs boson machine learning challenge, 2014. URL `https://www.kaggle.com/c/higgs-boson`.

[94] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[95] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 355–370, 2012. doi: 10.1007/978-3-642-31365-3_28. URL `http://link.springer.com/10.1007/978-3-642-31365-3{_}28`.

[96] Jiho Joo, Ki-Seok Jang, Sang Hoon Kim, In Gyoo Kim, Jin Hyuk Oh, Sun Ae Kim, Gyu-Seob Jeong, Yoonsoo Kim, Jun-Eun Park, Sungwoo Kim, Hankyu Chi, Deog-Kyoon Jeong, and Gyungock Kim. Silicon photonic receiver and transmitter operating up to 36 Gb/s for $\lambda\tilde{}$1550 nm. *Optics Express*, 23(9): 12232, 2015. ISSN 1094-4087. doi: 10.1364/OE.23.012232.

[97] U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop, 2008.

[98] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, November 2006. ISSN 0004-5411. doi: 10.1145/1217856.1217858.

[99] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[100] F. Keceli, A. Tzannes, G.C. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the XMT many-core architecture. In *Proc. IPDPSW*, pages 1282–1291, 2011. doi: 10.1109/IPDPS.2011.270.

[101] Fuat Keceli. *Power and Performance Studies of the Explicit Multi-Threading (XMT) Architecture*. PhD thesis, University of Maryland, 2011. `http://hdl.handle.net/1903/11926`.

[102] Fuat Keceli, Tali Moreshet, and Uzi Vishkin. Power-performance comparison of single-task driven many-cores. *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 348–355, December 2011. doi: 10.1109/ICPADS.2011.101.

[103] Fuat Keceli, Tali Moreshet, and Uzi Vishkin. Power-performance comparison of single-task driven many-cores. In *Proc. ICPADS*, 2011.

[104] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM programming*. Wiley-Interscience, J. Wiley & Sons, Inc., 2001.

[105] S. T. Klein and Y. Wiseman. Parallel Huffman decoding with applications to JPEG files. *The Computer Journal*, 46(5):487–497, 2003. doi: 10.1093/comjnl/46.5.487.

[106] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-74104-6. doi: 10.1007/978-3-540-74105-3. URL `http://link.springer.com/10.1007/978-3-540-74105-3`.

[107] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. *ACM SIGARCH Computer Architecture News*, 20(2):46–57, Apr. 1992. ISSN 01635964. doi: 10.1145/146628.139702. URL `http://portal.acm.org/citation.cfm?doid=146628.139702`.

[108] Stephan T. Lavavej. bwtzip: A linear-time portable research-grade universal data compressor. `http://nuwen.net/bwtzip.html`.

[109] Marc Lehmann. liblzf. `http://software.schmorp.de/pkg/liblzf.html`.

[110] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *International Haifa Verification Conference (HVC)*, pages 225–241, Haifa, Israel, 2015. doi: 10.1007/978-3-319-26287-1_14. URL `http://link.springer.com/10.1007/978-3-319-26287-1{_}14`.

[111] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1): 1–33, Jan. 2008. ISSN 0168-7433. doi: 10.1007/s10817-007-9084-z. URL `http://link.springer.com/10.1007/s10817-007-9084-z`.

[112] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28:39–55, 2008. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2008.31.

[113] Salem Malikic, Simone Ciccolella, Farid Rashidi Mehrabadi, Camir Ricketts, Md. Khaledur Rahman, Ehsan Haghshenas, Daniel Seidman, Faraz Hach, Iman Hajirasouliha, and S. Cenk Sahinalp. PhISCS - a combinatorial approach for sub-perfect tumor phylogeny reconstruction via integrative use of single cell and bulk sequencing data. *bioRxiv*, page 376996, 2018. doi: 10.1101/376996. URL `https://www.biorxiv.org/content/early/2018/07/25/376996`.

[114] Yishay Mansour, Noam Nisan, and Uzi Vishkin. Trade-offs between communication throughput and parallel time. In *Proc. 26th Annual ACM Symp. on Theory of Computing*, pages 372–381, 1994.

[115] Yael Maon, Baruch Schieber, and Uzi Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Theor. Comput. Sci.*, 47:277–298, 1986. ISSN 0304-3975. URL `http://dl.acm.org/citation.cfm?id=41287.41291`.

[116] Marcos Maronas, Kevin Sala, Sergi Mateo, Eduard Ayguade, and Vicenc Beltran. Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 383–394, 2019. doi: 10.1109/HiPC.2019.00053. URL `https://ieeexplore.ieee.org/document/8990523/`.

[117] Joao Marques-Silva. Practical applications of boolean satisfiability. In *2008 9th International Workshop on Discrete Event Systems*, pages 74–80. IEEE, 2008. ISBN 978-1-4244-2592-1. doi: 10.1109/WODES.2008.4605925. URL `http://ieeexplore.ieee.org/document/4605925/`.

[118] Quirin Meyer, F Schönfeld, Marc Stamminger, and Rolf Wanka. 3-SAT on CUDA: Towards a massively parallel SAT solver. In *International Conference on High Performance Computing & Simulation*, pages 306–313, 2010. doi: 10.1109/HPCS.2010.5547116. URL `http://ieeexplore.ieee.org/document/5547116/`.

[119] Gary L. Miller and Vijaya Ramachandran. A new graph triconnectivity algorithm and its parallelization. *Combinatorica*, 12(1):53–76, 1992.

[120] Lynette I Millett, Samuel H Fuller, et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011.

[121] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 102–115, 2006. ISBN 3540372067. doi: 10.1007/11814948_13. URL `http://link.springer.com/10.1007/11814948{_}13`.

[122] Rory Mitchell and Eibe Frank. Accelerating the XGBoost algorithm using GPU computing. *PeerJ Computer Science*, 3:e127, July 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.127. URL `https://peerj.com/articles/cs-127`.

[123] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming - LFP '90*, pages 185–197, 1990. doi: 10.1145/91556.91631. URL `http://portal.acm.org/citation.cfm?doid=91556.91631`.

[124] Antonio Morgado, Federico Heras, and Joao Marques-Silva. Model-guided approaches for MaxSAT solving. In *International Conference on Tools with Artificial Intelligence*, pages 931–938. IEEE, Nov. 2013. ISBN 978-1-4799-2972-6. doi: 10.1109/ICTAI.2013.142. URL `http://ieeexplore.ieee.org/document/6735353/`.

[125] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th conference on Design automation - DAC '01*, pages 530–535, New York, New York, USA, 2001. ACM Press. ISBN 1581132972.

doi: 10.1145/378239.379017. URL http://portal.acm.org/citation.cfm?doid=378239.379017.

[126] Alexander Nadel. Solving MaxSAT with bit-vector optimization. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 54–72, 2018. doi: 10.1007/978-3-319-94144-8_4. URL http://link.springer.com/10.1007/978-3-319-94144-8{_}4.

[127] Alexander Nadel and Vadim Ryvchin. Bit-vector optimization. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 851–867, 2016. doi: 10.1007/978-3-662-49674-9_53. URL http://link.springer.com/10.1007/978-3-662-49674-9{_}53.

[128] J. Nickolls and W.J. Dally. The GPU computing era. *IEEE Micro*, 30(2): 56–69, March-April 2010.

[129] Vojtech Nikl and Jiri Jaros. Parallelisation of the 3D fast Fourier transform using the hybrid OpenMP/MPI decomposition. In *9th International Doctoral Workshop, MEMICS 2014, Revised Selected Papers*, pages 100–112, 2014. ISBN 978-3-319-14895-3. doi: 10.1007/978-3-319-14896-0{_}9.

[130] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. Data Compression Conference*, pages 193–202. IEEE, 2009. ISBN 978-0-7695-3592-0. doi: 10.1109/DCC.2009.42.

[131] Markus F.X.J. Oberhumer. LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[132] Sean O'Brien, Uzi Vishkin, James Edwards, Edo Waks, and Bao Yang. Can cooling technology save many-core parallel programming from its programming woes? In *Compiler, Architecture and Tools Conference (CATC)*, Intel Development Center, Haifa, Israel, November 23, 2015, or http://drum.lib.umd.edu/handle/1903/17153.

[133] D. Padua, U. Vishkin, and J. Carver. Joint UIUC/UMD parallel algorithms/programming course. In *First NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-11)* , in conjunction with IPDPS, Anchorage, Alaska, May 16, 2011.

[134] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, August 2009. doi: 10.14778/1687553.1687569. URL http://dl.acm.org/citation.cfm?doid=1687553.1687569.

[135] R.A. Patel, Yao Zhang, J. Mak, A. Davidson, and J.D. Owens. Parallel lossless data compression on the gpu. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, 2012. doi: 10.1109/InPar.2012.6339599.

[136] M. Powell. The Canterbury corpus. http://corpus.canterbury.ac.nz/, 2001.

[137] Mohamed A. Rabie, Premachandran C. S., Rakesh Ranjan, Mahadevan Iyer Natarajan, Sing Fui Yap, Daniel Smith, Sarasvathi Thangaraju, Ramakanth Alapati, and Francis Benistant. Novel stress-free keep out zone process development for via middle TSV in 20nm planar CMOS technology. In *IEEE International Interconnect Technology Conference*, pages 203–206, 2014. ISBN 978-1-4799-5018-8. doi: 10.1109/IITC.2014.6831870.

[138] Vijaya Ramachandran and Uzi Vishkin. Efficient parallel triconnectivity in logarithmic time. In *Proc. AWOC*, pages 33–42, 1988. ISBN 3-540-96818-0. URL http://dl.acm.org/citation.cfm?id=648246.753807.

[139] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985. ISSN 0020-0190. doi: DOI:10.1016/0020-0190(85)90024-9.

[140] Sühleyman Cenk Sahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 300–309, New York, NY, USA, 1994. ACM. ISBN 0-89791-663-8. doi: 10.1145/195058.195164.

[141] A. Beliz Saybasili, Alexandros Tzannes, Bernard R. Brooks, and Uzi Vishkin. Highly parallel multi-dimensional fast Fourier transform on fine- and coarse-grained many-core approaches. In *Proc. Parallel and Distributed Computing and Systems*, 2009.

[142] Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with LA(Q) cost functions. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 484–498, 2012. ISBN 9783642313653. doi: 10.1007/978-3-642-31365-3_38. URL http://link.springer.com/10.1007/978-3-642-31365-3{_}38.

[143] J. Seward. On the performance of BWT sorting algorithms. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 173 –182, 2000. doi: 10.1109/DCC.2000.838157.

[144] Julian Seward. bzip2, a program and library for data compression. http://www.bzip.org/.

[145] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[146] Yossi Shiloach and Uzi Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, February 1982. ISSN 0196-6774. doi: http://dx.doi.org/10.1016/0196-6774(82)90013-X.

[147] Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras. Power-efficient computer architectures: Recent advances. *Synthesis Lectures on Computer Architecture*, 9(3):1–96, 2014. doi: 10.2200/S00611ED1V01Y201411CAC030.

[148] J. Soman, K. Kishore, and P.J. Narayanan. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(4): 325–339, December 2010.

[149] J. Soman, K. Kishore, and P.J. Narayanan. A fast GPU algorithm for graph connectivity. In *Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010. doi: 10.1109/IPDPSW.2010.5470817.

[150] Sukhyun Song and Jeffrey K Hollingsworth. Scaling Parallel 3-D FFT with Non-Blocking MPI Collectives. In *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 1–8, 2014. doi: 10.1109/ScalA.2014.9.

[151] Mircea R Stan, Sudhanva Gurumurthi, Robert J Ribando, and Kevin Skadron. Interaction of scaling trends in processor architecture and cooling. *2010 26th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*, pages 198–204, 2010. ISSN 10652221. doi: 10.1109/STHERM.2010.5444290.

[152] Chen Sun et al. Single-chip microprocessor that communicates directly using light. *Nature*, 528(7583):534–538, 2015.

[153] Jinghao Sun, Nan Guan, Feng Li, Huimin Gao, Chang Shi, and Wang Yi. Real-time scheduling and analysis of OpenMP DAG tasks supporting nested parallelism. *IEEE Transactions on Computers*, 9340, 2020. ISSN 0018-9340. doi: 10.1109/TC.2020.2972385. URL `https://ieeexplore.ieee.org/document/8986583/`.

[154] X Sun, G. Van der Plas, M Detalle, and E Beyne. Analysis of 3D interconnect performance: Effect of the Si substrate resistivity. In *IEEE International Conference on 3D Systems Integration (3DIC)*, 2014. doi: 10.1109/3DIC.2014.7152156.

[155] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 12–20, 1984. ISBN 0-8186-0591-X. doi: 10.1109/SFCS.1984.715896.

[156] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985. doi: 10.1137/0214061.

[157] David B Tuckerman and RFW Pease. High-performance heat sinking for VLSI. *Electron Device Letters, IEEE*, 2(5):126–129, 1981.

[158] Stephen Tyree, Kilan Q. Weinberger, and Kunal Agrawal. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396, 2011. ISBN 9781450306324. doi: 10.1145/1963405.1963461. URL `http://portal.acm.org/citation.cfm?id=1963461`.

[159] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Transactions on Programming Languages and Systems*, 36(3):10:1–10:51, Aug. 2014. ISSN 01640925. doi: 10.1145/2629643. URL `http://dx.doi.org/10.1145/2629643`.

[160] G. Upadhya, J. Hom, K. Goodson, and M. Munch. Electro-kinetic microchannel cooling system for servers. In *The Ninth Intersociety Conference on Thermal and Thermomechanical Phenomena In Electronic Systems*, pages 367–371, 2004. ISBN 0-7803-8357-5. doi: 10.1109/ITHERM.2004.1319198.

[161] Dimitrios Velenis, Michele Stucchi, Erik Jan Marinissen, Bart Swinnen, and Eric Beyne. Impact of 3D design choices on manufacturing cost. In *IEEE International Conference on 3D System Integration (3DIC)*, 2009. doi: 10.1109/3DIC.2009.5306575.

[162] U. Vishkin, G. Caragea, and B. Lee. Models for advancing PRAM and other algorithms into parallel programs for a PRAM-on-chip platform. In S. Rajasekaran and J. Reif, editors, *Handbook on Parallel Computing: Models, Algorithms, and Applications*, chapter 5. Chapman and Hall/CRC Press, 2008.

[163] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. `http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf`, February 2009.

[164] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Communications of the ACM*, 54(1):75–85, Jan. 2011. ISSN 00010782. doi: 10.1145/1866739.1866757. URL `http://portal.acm.org/citation.cfm?doid=1866739.1866757`.

[165] Uzi Vishkin. Is multicore hardware for general-purpose parallel processing broken? *Commun. ACM*, 57(4):35–39, April 2014. ISSN 0001-0782.

[166] Uzi Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM J. Computing*, 14(2):303–314, 1985.

[167] David W. Wall. Limits of instruction-level parallelism. Technical report, DEC Western Research Laboratory, 1993. URL `https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-93-6.pdf`.

[168] Tjark Weber. Integrating a SAT solver with an LCF-style theorem prover. *Electronic Notes in Theoretical Computer Science*, 144(2):67–78, 2006. ISSN

15710661. doi: 10.1016/j.entcs.2005.12.007. URL `https://linkinghub.elsevier.com/retrieve/pii/S1571066106000089`.

[169] R. Weerasekera, M. Grange, D. Pamunuwa, and H. Tenhunen. On signalling over through-silicon via (TSV) interconnects in 3-D integrated circuits. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010. doi: 10.1109/DATE.2010.5457013.

[170] Xingzhi Wen and Uzi Vishkin. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 55–66, 2008. doi: http://doi.acm.org/10.1145/1366230.1366240.

[171] Xingzhi Wen and Uzi Yehoshua Vishkin. System and method for thread handling in multithreaded parallel computing of nested threads, U.S. Patent 8,209,690 B2, 2012.

[172] Samuel Webb Williams, Andrew Waterman, and David A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, University of California at Berkeley, 2008. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html`.

[173] Yair Wiseman. Burrows-wheeler based JPEG. *Data Science Journal*, 6, 2007.

[174] Vitaly Zakharenko, Tor Aamodt, and Andreas Moshovos. Characterizing the performance benefits of fused CPU/GPU systems using FusionSim. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013. doi: 10.7873/DATE.2013.148. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513594`.

[175] Lian Zhang, Kenneth E. Goodson, and Thomas W. Kenny. *Silicon Microchannel Heat Sinks: Theories and Phenomena*. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-662-09899-8_1.

[176] Xuezhe Zheng, Frankie Liu, Jon Lexau, Dinesh Patil, Guoliang Li, Ying Luo, Hiren D. Thacker, Ivan Shubin, Jin Yao, Kannan Raj, Ron Ho, John E. Cunningham, and Ashok V. Krishnamoorthy. Ultralow power 80 Gb/s arrayed CMOS silicon photonic transceivers for WDM optical links. *Journal of Lightwave Technology*, 30(4):641–650, 2012. ISSN 07338724. doi: 10.1109/JLT.2011.2179287.